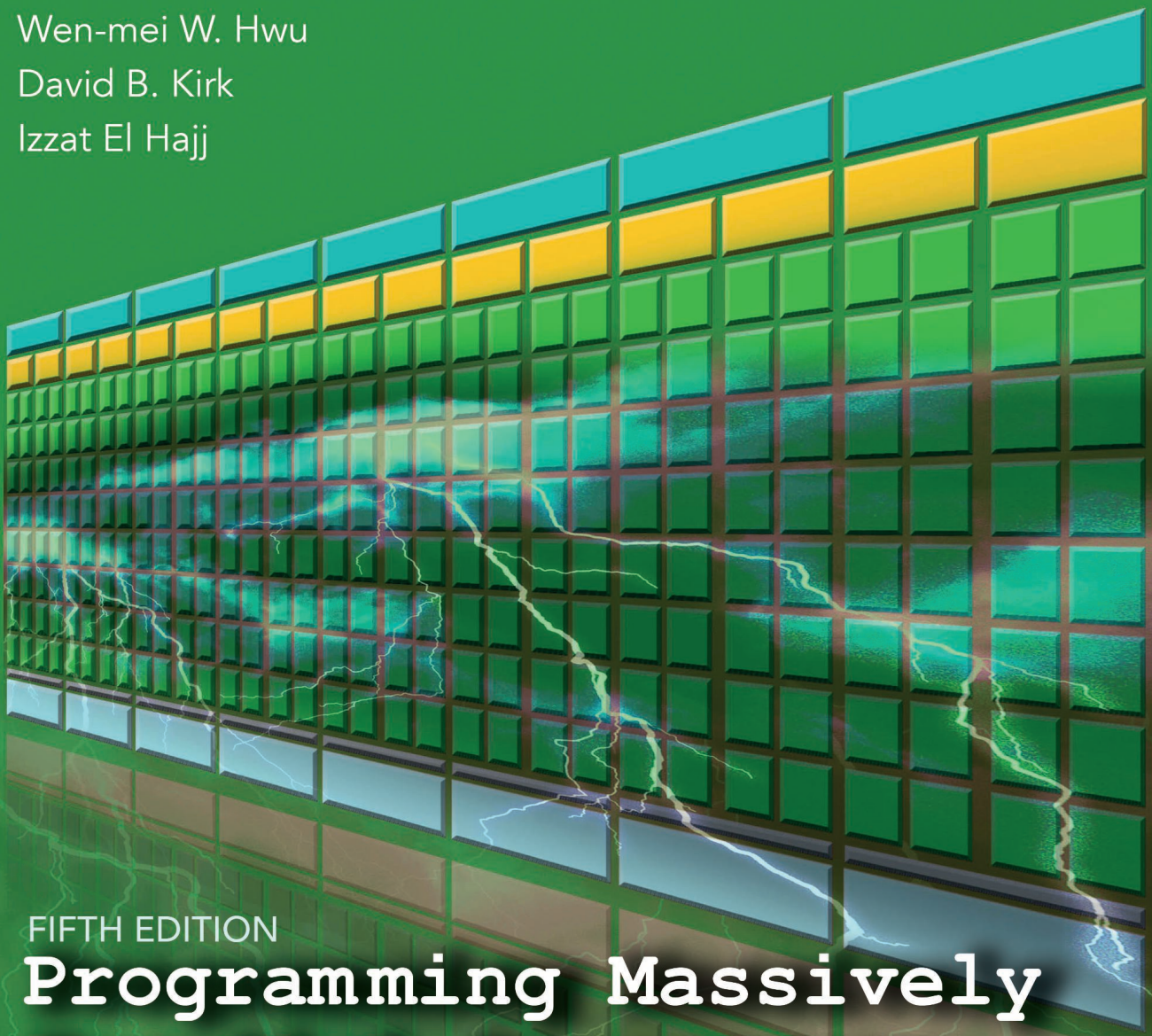
Wen-mei W. Hwu
David B. Kirk
Izzat El Hajj



FIFTH EDITION

Programming Massively Parallel Processors

A Hands-on Approach

MORGAN KAUFMANN

Programming Massively Parallel Processors

Programming Massively Parallel Processors

A Hands-on Approach

Fifth Edition

Wen-mei W. Hwu
NVIDIA
University of Illinois at Urbana-Champaign
Champaign, IL, United States

David B. Kirk
Formerly NVIDIA
Santa Clara, CA, United States

Izzat El Hajj
American University of Beirut
NVIDIA
Beirut, Lebanon



MK
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

For accessibility purposes, images in electronic versions of this book are accompanied by alt text descriptions provided by Elsevier. For more information, see https://www.elsevier.com/about/accessibility.

Books and Journals published by Elsevier comply with applicable product safety requirements. For any product safety concerns or queries, please contact our authorised representative, Elsevier B.V., at productsafety@elsevier.com.

Publisher's note: Elsevier takes a neutral position with respect to territorial disputes or jurisdictional claims in its published content, including in maps and institutional affiliations.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices
Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-443-43900-1

For information on all Morgan Kaufmann publications
visit our website at https://www.elsevier.com/books-and-journals

Publisher: Peter Linsley
Acquisitions Editor: Stephen Merken
Editorial Project Manager: Prashant Mahajan
Publishing Services Manager: Deepthi Unni
Production Project Manager: Surya Narayanan Jayachandran
Cover Designer: Mark Rogers

Typeset by VTeX

Printed in India

Last digit is the print number:
9 8 7 6 5 4 3 2 1



To Sabrina, Amanda, Bryan, and Carissa
To Caroline, Rose, and Leo
To Mona, Amal, and Ali

for enduring our absence while working on the course
and the book – once again!

Contents

Foreword xv
Preface xvii
Acknowledgments xxvii

CHAPTER 1 Introduction 1

1.1 Heterogeneous parallel computing 3
1.2 Why more speed or parallelism? 7
1.3 Speeding up real applications 9
1.4 Challenges in parallel programming 11
1.5 Related parallel programming interfaces 12
1.6 Overarching goals 14
1.7 Organization of the book 15
References 18

PART 1 Fundamental concepts

CHAPTER 2 Heterogeneous data-parallel computing 21

2.1 Data parallelism 21
2.2 CUDA C++ program structure 24
2.3 A vector addition example 26
2.4 Device global memory and data transfer 28
2.5 Kernel functions and threading 32
2.6 Calling kernel functions 37
2.7 Compilation 39
2.8 Summary 40
2.9 Exercises 41
References 43

CHAPTER 3 Multidimensional grids and data 45

3.1 Multidimensional grid organization 45
3.2 Mapping threads to MultiDimensional data 48
3.3 Image blur – a more complex kernel 55
3.4 Matrix multiplication 59
3.5 Summary 64

CHAPTER 4 Compute architecture and scheduling 67

4.1 Architecture of a modern GPU 68
4.2 Thread block scheduling 69
4.3 Synchronization and transparent scalability 70
4.4 Warps and SIMD hardware 74

4.5 Control divergence 79
4.6 Warp scheduling and latency tolerance 82
4.7 Resource partitioning and occupancy 85
4.8 Querying device properties 87
4.9 Summary 89
4.10 Exercises 90
References 92

CHAPTER 5 Memory architecture and data locality 93

5.1 Memory bandwidth as a performance limiter 93
5.2 CUDA memory types 98
5.3 Tiling for reduced memory traffic 104
5.4 A tiled matrix multiplication kernel 108
5.5 Boundary checks 112
5.6 Impact of memory usage on occupancy 115
5.7 Summary 118
5.8 Exercises 119
References 121

CHAPTER 6 Performance considerations 123

6.1 Global memory access coalescing 124
6.2 Hiding memory latency 132
6.3 Vector loads and stores 138
6.4 Shared memory bank conflicts 139
6.5 Thread coarsening 141
6.6 Loop unrolling 144
6.7 Double buffering 146
6.8 A checklist of optimizations 147
6.9 Optimization strategy 153
6.10 Summary 154
6.11 Exercises 154
References 155

PART 2 Parallel patterns

CHAPTER 7 Convolution 159

7.1 Background 159
7.2 Parallel convolution – a basic kernel 164
7.3 Memory bandwidth considerations 167
7.4 Constant memory and caching 168
7.5 Tiled convolution with halo cells 172
7.6 Tiled convolution using caches for halo cells 178
7.7 Summary 179
7.8 Exercises 180

CHAPTER 8 Stencil computation **183**
8.1 Background 184
8.2 Parallel stencil – a basic kernel 187
8.3 Memory bandwidth considerations 189
8.4 Shared-memory tiling for stencil sweep 189
8.5 Thread coarsening 193
8.6 Register tiling 196
8.7 Summary 198
8.8 Exercises 198

CHAPTER 9 Histogram **201**
9.1 Background 202
9.2 Atomic operations and a basic histogram kernel 203
9.3 Latency and throughput of atomic operations 209
9.4 Privatization 211
9.5 Thread coarsening 214
9.6 Thread-level privatization 217
9.7 Summary 219
9.8 Exercises 220
References 220

CHAPTER 10 Reduction **221**
10.1 Background 221
10.2 Reduction trees 223
10.3 A simple reduction kernel 227
10.4 Reducing control divergence 230
10.5 Reducing memory access divergence 234
10.6 Reducing global memory accesses 236
10.7 Reducing synchronization overhead with warp-level primitives 237
10.8 Further reducing synchronization overhead with two-stage warp-wide reduction 241
10.9 Reduction for arbitrary length inputs 244
10.10 Thread coarsening to reduce overhead 246
10.11 Summary 249
10.12 Exercises 249
References 250

CHAPTER 11 Scan **251**
11.1 Background 252
11.2 Parallel scan with the Kogge-Stone algorithm 254
11.3 Double-buffering to reduce synchronization 258
11.4 Warp-level primitives to reduce synchronization 261
11.5 Work efficiency considerations 265
11.6 Coarsening to improve work-efficiency 267

11.7 Register tiling to avoid shared memory access latency 270
11.8 Memory bandwidth considerations 272
11.9 Consolidating block segments for a global scan 273
11.10 Parallel scan with the Brent-Kung algorithm 282
11.11 Summary 285
11.12 Exercises 286
References 286

CHAPTER 12 Filter **289**

12.1 Background 289
12.2 A simple parallel unstable filter 290
12.3 Coalescing atomic operations with warp-level primitives 291
12.4 Privatization 295
12.5 A simple parallel stable filter 297
12.6 Improving memory coalescing with shared memory and thread coarsening 298
12.7 In-place stable filter 300
12.8 Related patterns 301
12.9 Summary 302
12.10 Exercises 302
References 302

CHAPTER 13 Merge **303**

13.1 Background 303
13.2 A sequential merge algorithm 305
13.3 A parallelization approach 306
13.4 Co-rank function implementation 308
13.5 A basic parallel merge kernel 313
13.6 A tiled merge kernel to improve coalescing 314
13.7 A circular-buffer merge kernel 321
13.8 Thread coarsening for merge 327
13.9 Summary 327
13.10 Exercises 327
References 328

CHAPTER 14 Sorting **329**

14.1 Background 330
14.2 Parallel odd-even sort 331
14.3 Parallel merge sort 333
14.4 Radix sort 334
14.5 Parallel radix sort 336
14.6 Optimizing for memory coalescing 339
14.7 Choice of radix value 342
14.8 Thread coarsening to improve coalescing 344

14.9 Other parallel sort methods 345
14.10 Summary 346
14.11 Exercises 347
References 347

CHAPTER 15 Advanced optimizations for matrix multiplication **349**

15.1 Background 349
15.2 Data reuse analysis 350
15.3 Using larger tiles with thread coarsening 352
15.4 Register tiling of the input tiles 357
15.5 Coalesced storing of the output tile 360
15.6 Eliminating bank conflicts 361
15.7 Occupancy considerations 363
15.8 Software pipelining 364
15.9 Specialized software and hardware support 368
15.10 Summary 370
15.11 Exercises 370
References 370

PART 3 Advanced patterns and applications

CHAPTER 16 Dynamic programming and wavefront parallelism **373**

16.1 Dynamic programming 373
16.2 Implementation approaches 374
16.3 Wavefront patterns 376
16.4 Floyd-Warshall algorithm 377
16.5 Genome sequence alignment and Smith-Waterman algorithm 382
16.6 Wavefront parallelization: block-level tiling 384
16.7 Hyperplane transformation 389
16.8 More optimizations 397
16.9 Summary 399
16.10 Exercises 399
References 400

CHAPTER 17 Sparse matrix computation **401**

17.1 Background 402
17.2 A simple SpMV kernel with the COO format 404
17.3 Grouping row non-zeros with the CSR format 407
17.4 Improving memory coalescing with the ELL format 410
17.5 Regulating padding with the hybrid ELL-COO format 414
17.6 Reducing control divergence with the JDS format 416
17.7 Column-wise accessibility with the CSC format 418
17.8 Summary 421
17.9 Exercises 422
References 423

CHAPTER 18 Graph traversal **425**

18.1 Background 425
18.2 Breadth-first search 429
18.3 Vertex-centric parallelization of BFS 431
18.4 Edge-centric parallelization of BFS 436
18.5 Improving work efficiency with frontiers 438
18.6 Reducing contention with privatization 442
18.7 Reducing launch overhead with cooperative groups 445
18.8 Other optimizations 448
18.9 Summary 449
18.10 Exercises 449
References 451

CHAPTER 19 Convolutional neural networks **453**

19.1 Convolutional neural networks 454
19.2 A CUDA convolutional layer kernel 460
19.3 Formulating convolutional layer as GEMM 463
19.4 CUDNN library 472
19.5 Summary 474
19.6 Exercises 474
References 475

CHAPTER 20 Large language models **477**

20.1 Transformer architecture 478
20.2 Multi-head attention 482
20.3 Implementing attention in CUDA 486
20.4 KV caching 488
20.5 Flash attention 492
20.6 KV cache arithmetic intensity and memory requirement 504
20.7 Alleviating the memory requirements of the attention mechanism 508
20.8 Summary 510
20.9 Exercises 511
References 511

CHAPTER 21 Electrostatic potential map **513**

21.1 Background 513
21.2 Scatter vs. gather in kernel design 515
21.3 Thread coarsening 519
21.4 Memory coalescing 521
21.5 Cutoff binning for data size scalability 523
21.6 Summary 527
21.7 Exercises 528
References 528

CHAPTER 22 Algorithm selection, problem decomposition, and problem formulation **529**

22.1 Algorithm selection 530
22.2 Problem decomposition 532
22.3 Application level considerations – Amdahl's law 537
22.4 Problem formulation 538
22.5 Batching: latency vs. throughput 539
22.6 Summary 540
References 540

CHAPTER 23 Multi-GPU programming **541**

23.1 Stencil as a running example 542
23.2 Multi-GPU stencil with MPI 545
23.3 Overlapping computation and communication 553
23.4 Multi-GPU stencil with NCCL 563
23.5 Multi-GPU stencil with NVSHMEM 568
23.6 Summary 574
23.7 Exercises 576
References 576

CHAPTER 24 Conclusion and future outlook **577**

24.1 Goals revisited 577
24.2 Future outlook 578

PART 4 Appendices

APPENDIX A Numerical considerations **583**

A.1 Floating-point data representation 583
A.2 Representable numbers 586
A.3 Precision and special bit patterns in IEEE format 590
A.4 Arithmetic accuracy and rounding 591
A.5 Lower-precision formats 592
A.6 Algorithm considerations 593
A.7 Linear solvers and numerical stability 594
A.8 Summary 599
A.9 Exercises 599
References 600

APPENDIX B Deep learning basics **601**

B.1 Classifiers 602
B.2 Fully connected and convolutional layers 605
B.2.1 Training models 606
B.3 Convolutional neural networks 612

B.4 CNN inference and training: sequential implementation 613
B.4.1 CNN training 616
B.5 Summary 621
References 621

APPENDIX C CUDA memories, address spaces, and pointers **623**

C.1 Unified device memory address space 624
C.2 Zero-copy access to host memory 624
C.3 Unified virtual address space 625
C.4 Large virtual and physical GPU address spaces 625
C.5 Unified physical address space 626
C.6 Unified memory 626
C.7 Access to full host virtual address space 627
C.8 Page fault handling 628
C.9 Virtual address space control 629
References 629

Index 631

Foreword

Thank you to Wen-mei, David, and Izzat. Your work has shaped how millions of developers learn to think in parallel and helped build the foundations of accelerated computing.

We are now at a major inflection point in computing. Accelerated computing and AI have set off a new industrial revolution, transforming every layer of technology and every industry.

The ideas in this book are essential for this moment. They prepare developers, researchers, and engineers to build the systems and applications driving this new era of intelligence.

NVIDIA invented accelerated computing to solve problems that CPUs could not. This breakthrough sparked the Big Bang of AI and launched a platform shift transforming every layer of technology – from chips and systems to networking, algorithms, and applications. This shift is reshaping science, industry, and the global economy.

GPU supercomputers have become AI factories. They turn data and energy into intelligence. They power every major domain: science, healthcare, manufacturing, climate modeling, autonomous machines, robotics, quantum research, 6G telecommunications, and gaming.

AI is essential infrastructure. Every company will use it. Every country will build it. These systems will become the new infrastructure that powers discovery, productivity, and growth.

This book gives you the fundamental ideas behind accelerated computing. Parallelism. Locality. Concurrency. Optimization.

These principles are the core of high-performance computing and modern AI. They will give you the ability to build applications that scale across GPUs, reason over massive datasets, and drive breakthroughs in science and industry.

You will use these concepts as AI models grow, as agentic systems understand context and plan, and as physical AI and robotics become part of every industry.

Generations of NVIDIANs learned from this book. Many contributed their expertise to refine these concepts and strengthen the foundations on which the global developer community builds every day. Their work reflects the dedication and craftsmanship that have advanced accelerated computing from its earliest days to this new industrial era. It remains one of the foundational texts for the builders moving our field forward.

Accelerated computing continues to open new frontiers.

AI models help researchers explore vast amounts of data, simulate complex physical systems, and discover insights once out of reach. Physical AI and robotics are transforming automation and design. Scientists, engineers, and students everywhere can now work at scales once impossible. And although the progress has been extraordinary, we are still in the early days of what accelerated computing and AI will unlock.

A scientist once told me, “Because of your GPU, I can do my life’s work in my lifetime.” That is what accelerated computing makes possible.

The knowledge in this book will help you build what comes next. I hope it helps you do your life’s work and contribute to the breakthroughs ahead.

Jensen Huang
Founder and CEO, NVIDIA

Preface

We are proud to present to you the 5th edition of "Programming Massively Parallel Processors – a hands-on approach."

Since the publication of the fourth edition in 2022, exascale computing using hundreds of thousands of GPUs interconnected with fast networks has powered the AI revolution to full gear. GPUs have played a front-and-center role in training the large language models behind $ChatGPT$, $DeepSeek$, and $Gemini$. GPUs have also become the workhorse of serving AI queries from users that span the entire age, race, and geography spectrum. The impact of GPU computing is now felt across all sectors of the industry and society.

We are also witnessing wide adoption of GPU computing in key industry vertical markets, such as transportation, e-commercial, manufacturing, health care, financial services, and logistics. Many will be able to achieve breakthroughs in these disciplines using digital twins of their physical infrastructure that are of unprecedented levels of scale, accuracy, safety, controllability, and observability. This book provides a critical ingredient for this vision: teaching parallel programming to millions of graduate and undergraduate students so that parallel thinking and programming skills will be as pervasive as calculus.

The target audience of this book starts with graduate and undergraduate students from all science and engineering disciplines where parallel computing is needed to achieve breakthroughs. From our experience, the book has also been successfully used by many industry professional developers who need to refresh their parallel computing skills and keep up-to-date with the ever increasing speed of technology evolution. These professional developers work in fields such as machine learning, network security, autonomous vehicles, computational financing, data analytics, cognitive computing, mechanical engineering, civil engineering, electrical engineering, bio-engineering, physics, chemistry, astronomy, and geography; they use computation to advance their field. We assume that the reader has at least some basic C++ programming experience. The book takes the approach of teaching parallel programming by building up an intuitive understanding of the techniques. We use CUDA, a parallel programming environment that is supported on NVIDIA GPUs. There are more than one billion of these processors in use and millions of programmers actively using CUDA.

Since the publication of the fourth edition, we have received numerous comments from our readers and instructors. Many told us about the existing features which they value. Others gave us ideas about how we should expand its contents to make the book even more valuable. Furthermore, the hardware and software technology for heterogeneous parallel computing has advanced tremendously since then. In the hardware arena, three more generations of GPU computing architectures, Ada Lovelace, Hopper, and Blackwell, have been introduced. In the software domain, CUDA 11 through CUDA 13 have allowed programmers to access new hardware

and system features. New algorithms have also been developed or adapted to take advantage of new GPU features. Accordingly, we added four new chapters and rewrote or expanded a substantial number of the existing chapters and appendices. We also removed three chapters to make room for the newly added contents.

The four new parallel patterns and applications chapters are Chapter 12 (Filtering), Chapter 15 (Advanced Optimizations for Matrix Multiplication), Chapter 16 (Wavefront Parallelism), and Chapter 20 (Large Language Models). Our motivations for adding these chapters are as follows:

- Chapter 12 (Filtering): Filtering is a classical parallel pattern that is often included in the same libraries that support the other patterns we cover. The newly added chapter covers both unstable and stable filtering. Unstable filtering is used as a vehicle for introducing the optimization of coalescing atomic operations, and the underlying warp voting functions needed by the optimization. It is also used to introduce coalesced groups from the cooperative groups API. Stable filter is used to demonstrate an important use case of the scan pattern. The chapter also showcases multiple optimizations from the checklist in Chapter 6, namely privatization, thread coarsening, and packing in shared memory to improve memory access coalescing.
- Chapter 15 (Advanced Optimizations for Matrix Multiplication): Matrix multiplication is without a doubt the most important computation that GPUs are used for today. In the previous edition, matrix multiplication was covered in multiple chapters. A basic implementation was introduced in Chapter 3 (Multi-dimensional Grids and Data) to showcase programming with 2D grids and blocks. It was covered again in Chapter 5 (Memory Architecture and Data Locality) to demonstrate how to apply the shared memory tiling optimization. It was also covered in Chapter 6 (Performance Considerations) to demonstrate a simple application of the thread coarsening optimization. In this edition, we introduce a new chapter that covers the advanced optimizations that make matrix multiplication execute as efficiently on GPUs as it does today. These include applying thread coarsening to enlarge the output tiles to improve arithmetic intensity, register tiling of the input and output tiles, leveraging vector loads and stores, coalesced storing of the output tiles, eliminating bank conflicts, and tolerating low occupancy execution with software pipelining. We also give an overview of important hardware support for further accelerating matrix multiplication such as tensor cores, tensor memory, asynchronous memory copies, and the tensor memory accelerator.
- Chapter 16 (Wavefront Parallelism): Wavefront parallelism is an algorithmic technique to extract data parallelism from dynamic programming algorithms. Dynamic programming problems are solved by recursively dividing them into smaller overlapping subproblems. Their nature makes challenging to partition the problems, and thus their parallelization. With the classical Smith-Waterman algorithm for genome sequence alignment as our running example, we first approach their parallelization by dividing the dynamic programming table into square (or rectangular) tiles and assigning their computation to thread blocks. However, due

to the changing dimensions of the wavefronts, these tiles cause warp underutilization and bad data locality. Then, we apply a hypertile transformation that tilts the tiles in a way that all threads remain active during the entire execution and data locality improves.

- Chapter 20 (Large Language Models): Large Language Models (LLMs) have become the keystone of Artificial Intelligence (AI), which represents arguably the most significant scientific, technological, and societal revolution since the appearance of the Internet. Chatbot applications (just an example of LLMs' multiple applications) like OpenAI ChatGPT reached millions of users in just a few days after their launch and their impact on the ways people work, learn, and communicate is being transformative and indelible. The high computation requirement for training and using LLM models have been the driving force behind the fast growth of hundred-thousand-GPU data centers around the world. This chapter introduces the architecture of Transformers, which are as of today the most successful LLM architecture. We put special focus on their attention mechanism, which identifies relationships (e.g., semantic, contextual, syntactic) between fragments of words called tokens. We present two key optimizations of the parallel implementation of attention. First, KV caching avoids recomputation during the generation of new tokens by the LLM. Second, flash attention proposes an algorithmic transformation that greatly reduces overheads derived from multiple kernel execution and repeated memory accesses. We also use the attention mechanism to illustrate the use of batching to improve the throughput of processing queries and resource utilization at the cost of increased latency experienced by each individual query.

In addition to the newly added chapters, three chapters have gone through major expansion and revision. The rationales for rewriting these chapters are as follows:

- Chapter 6 (Performance Considerations): In the previous edition, this chapter included a checklist of six optimizations: occupancy tuning, reducing control divergence, and shared memory and register tiling which were covered in previous chapters, privatization which was covered in later chapters, and coalescing of memory accesses and thread coarsening which were introduced in the chapter itself. In the current edition, the checklist is significantly expanded with additional optimizations including warp-level primitives which are covered in later chapters, and loop unrolling, vector loads and stores, avoiding shared memory bank conflicts, and double buffering which are introduced in the chapter itself. Moreover, thread coarsening was previously demonstrated with matrix multiplication. This example is replaced with a more abstract coverage since the application of thread coarsening to matrix multiplication is discussed in-depth in the new Chapter 15.
- Chapter 11 (Scan): In the previous edition, this chapter focused on comparing the Kogge-Stone and Brent-Kung approaches to parallel scan with emphasis on work efficiency. It also covered thread coarsening as a means for improving work efficiency, a scan-scan-add decomposition for parallelizing scan across multiple blocks, and a single-kernel scan based on single-lookback. In the current edition, the chapter was significantly revised to put less emphasis on the Brent-Kung approach, discuss work efficiency more thoroughly, and apply more optimizations in

the context of the Kogge-Stone approach. In particular, the revised chapter covers double-buffering and warp-level primitives as a means for reducing synchronization overhead, and register tiling as a means for reducing shared memory access latency. It also covers the reduce-scan-scan decomposition and compares it to the scan-scan-add decomposition from the perspective of memory bandwidth utilization. It fixes the implementation of single-lookback with C++ atomics that have release and acquire semantics. It also introduces multiple lookback and decoupled lookback that overcome the long latency experienced by the single lookback implementation.

- Chapter 23 (Multi-GPU Programming): In the previous edition, this chapter used a 3D stencil computation as a running example to introduce multi-GPU programming with MPI without CUDA-awareness. In the current edition, the running example was revised to use 2D stencil with residual computation. Moreover, the running example was used to introduce not just one but three different multi-GPU programming models: CUDA-aware MPI, NCCL, and NVSHMEM.

The eight chapters and appendix that were subject to substantial expansion or updates are:

- Chapter 4 (Compute Architecture and Scheduling): This chapter was revised to use the Hopper H100 GPU specifications in its examples. Moreover, additional text was added to introduce thread block clusters, a new level of the CUDA thread grid hierarch that enables more threads to synchronize with each other and access each other's shared memory. Warp-level primitives are also briefly introduced, but are more thoroughly covered in later chapters.
- Chapter 5 (Memory Architecture and Data Locality): This chapter was also revised to use the Hopper H100 GPU specifications in its examples. Moreover, additional text was added to introduce distributed shared memory. The chapter also more thoroughly defines and discusses the concepts of arithmetic intensity, compute- and memory-boundedness, and speed of light analysis, setting the stage for more thorough analyses of memory bandwidth considerations in Chapter 7 (Convolution), Chapter 8 (Stencil), and Chapter 11 (Scan).
- Chapter 9 (Histogram): In the previous edition, this chapter used a text histogram as a running example to introduce atomic operations, focusing on CUDA's low-level atomic intrinsics. In the current edition, the revised chapter uses an image histogram as a more intuitive running example. It also uses the higher-level CUDA C++ atomics API which exposes the scope of atomic operations (block, device, system) and their memory ordering semantics (relaxed, release, acquire, sequential). In fact, the code examples in other chapters were also revised to use the higher-level API.
- Chapter 10 (Reduction): This chapter was expanded to introduce warp-level primitives, especially warp shuffle operations, and use these primitives to reduce the synchronization overhead of the reduction kernel.
- Chapter 14 (Sorting): This chapter was expanded to cover odd-even sort as a simpler pedagogical introduction to sorting on GPUs. It also observes that radix sort is built on the newly added stable filter pattern in the new Chapter 12 on filtering.

- Chapter 17 (Sparse Matrix Computation): This chapter was expanded to cover the Compressed Sparse Column (CSC) format. While CSC is not typically used for SpMV, it is useful in other contexts where column-wise traversal of the sparse matrix is desired. For this reason, it was added to the chapter.
- Chapter 18 (Graph Traversal): This chapter was expanded to use BFS as an example for demonstrating how the cooperative groups API can be used to launch kernels that can perform grid-wide barrier synchronizations. In particular, instead of launching a new grid for every BFS level, a new kernel was introduced that iterated across all levels while performing a grid-wide barrier synchronization within the kernel.
- Chapter 19 (Convolutional Neural Networks): When we wrote the chapter for the fourth edition, we felt that we needed to offer a more extensive background section on machine learning and deep learning to help readers understand the main contents of the chapter. As a result, the chapter became the longest chapter by far. Since then, students in all majors have increasingly taken courses in these areas. Therefore, for the fifth edition, we feel that much of the background material is no longer needed by most of our readers. We decided to remove the material from Chapter 19 into a new Appendix B (Deep Learning Basics). This way, readers who miss some background in these areas can still have easy access to the background material that is curated to be directly relevant to the chapter.
- Appendix A (Numerical Considerations): Since the publication of the fourth edition, there has been rapid progress in defining and adopting low-precision floating-point format standard across industry. This movement is in response to the insatiable demand for very high floating-point throughput and advances in quantization techniques for large language model parameters. With the help from expert reviewers, we expanded the appendix to cover the new formats and the improvements in the floating-point implementation in recent GPU architecture generations. We also added more insights into the original sections.

In order to make room for the newly added contents, three chapters were removed. It is never easy to let a chapter go. We carefully decided on the following chapters because of the recent changes in GPU architecture trends and application landscape.

- *Iterative MRI Reconstruction*: The main objective of this old chapter was to contrast an input-centric scatter-based implementation of the iterative MRI reconstruction application with an output-centric gather-based implementation. The same contrast is also made in Chapter 21 (Electrostatic Potential Map), but in the context of another application. Since the two chapters carried the same message, we decided to omit one of them and keep the other. We kept the electrostatic potential map chapter because the application is representative of a wider range of physical modeling and simulation use cases and because it presented other optimizations as well. We moved some of the fundamental discussions on the scatter-based and gather-based implementations into Chapter 21. Readers who are particularly interested in the Iterative MRI Reconstruction application can still refer to older editions of the book.

- *Dynamic Parallelism*: The main objective of this old chapter was to introduce dynamic parallelism, a feature of CUDA for capturing nested parallelism in computations by allowing threads running on the GPU to call kernel functions and launch new grids as they discover more work during their execution. Over a decade has passed since the first implementation of dynamic parallelism in the Kepler architecture. Unfortunately, the feature has not seen wide adoption, in part due to its high overhead. For this reason, we decided to remove the chapter from the book. Readers who are interested in learning about and using dynamic parallelism can still refer to the CUDA programming guide, or to old editions of the book. We plan to revise and include the spatial sorting pattern covered by this chapter in a future edition.
- *Other Advanced Practices*: The main objective of this old chapter was to briefly present some promising, advanced CUDA features that were not mature enough at the time and/or not illustrated with code examples. Over time, some of these features, such as cooperative groups and thread block clusters, have matured and are adopted into revised/new code examples of this edition. As a result, we decided to remove the old chapter. The materials in this chapter addressing the evolution of CUDA memory types, address spaces, and CUDA pointers' reach have been moved into the new Appendix C. These topics have been confusing to many CUDA developers, particularly as they continue to evolve with time. A clear understanding of the history and nuances help avoid performance pitfalls that can trip even experienced developers.

While making all these improvements, we tried to preserve the features that seem to contribute most to the book's popularity. First, we keep our explanations as intuitive as possible. While it is tempting to formalize some of the concepts, especially when we cover the basic parallel algorithms, we strive to keep all our explanations intuitive and practical. Second, we keep the book as concise as possible. While it is tempting to keep adding new material, we want to minimize the number of pages a reader needs to go through to learn all the key concepts. We accomplished this by removing some of the chapters mentioned earlier.

In addition to adding new chapters and substantially rewriting others since the previous edition, we have also organized the book into four major parts. The updated organization is illustrated in Fig. P.1. The first part introduces the fundamental concepts behind parallel programming, the GPU architecture, and performance analysis and optimization. This part has remained the same compared to the previous edition. The second part applies these concepts by covering common computation patterns and showing how they can be parallelized and optimized. Each parallel pattern also introduces a new programming feature or technique. Compared to the previous edition, the new Chapter 12 (Filtering) and Chapter 15 (Advanced optimizations for Matrix Multiplication) were added to this part and Chapter 14 (Sort) was moved from the third part to this part. The third part introduces additional advanced patterns and applications and continues to apply the optimizations that are practiced in the second part. However, it puts more emphasis on exploring alternative forms of problem

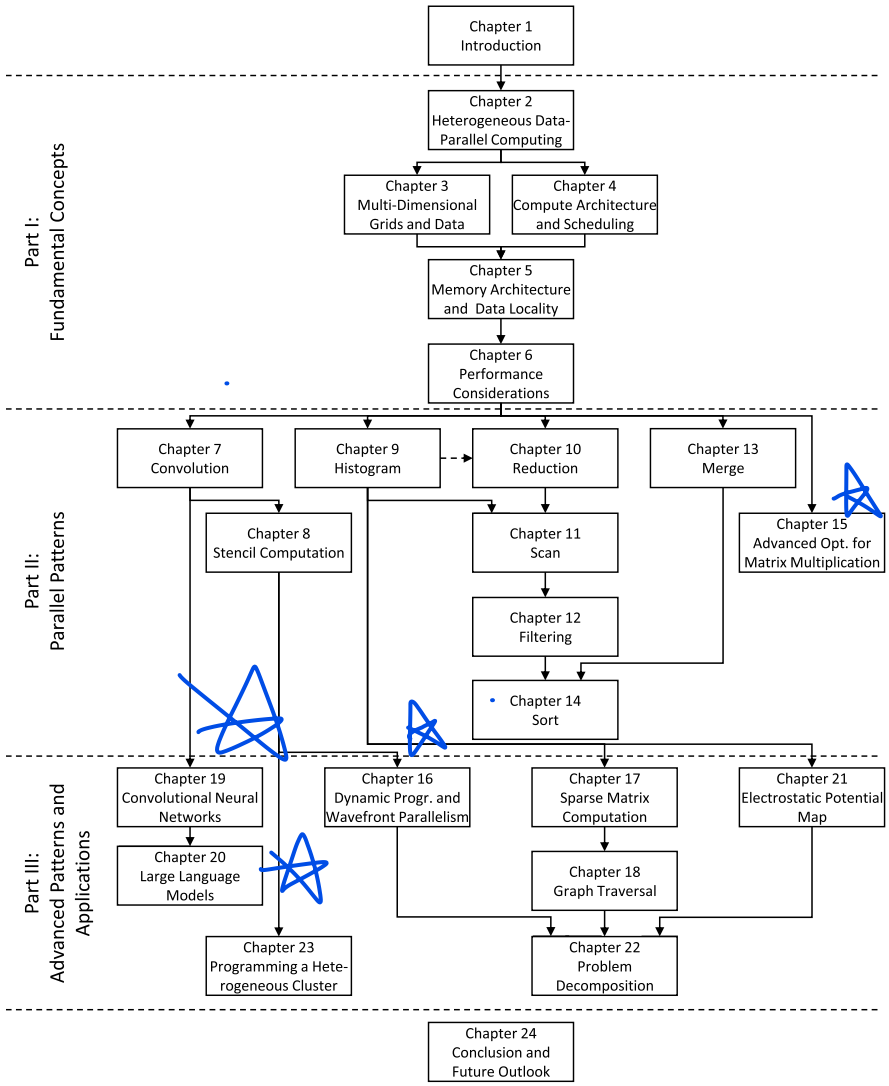


FIGURE P.1

Organization of the book.

decomposition to parallelize a computation and analyzes the trade-offs between different decompositions and their associated data structures. Compared to the previous edition, this part was expanded by the addition of Chapter 16 (Wavefront Parallelism) and Chapter 20 (Large Language Models). Moreover, Chapter 23 (Multi-GPU Programming) was moved to this part from the old fourth part. On the other hand, the old chapter on Iterative MRI Reconstruction was removed from this part, and as men-

tioned earlier, Chapter 14 (Sort) was moved from this part to the second part. Finally, the old fourth part on advanced practices and programming features was removed because the two chapters on *Dynamic Parallelism* and *Other Advanced Practices* were removed from the book, and the remaining Chapter 23 (Multi-GPU Programming) was moved to the third part.

How to use the book

We would like to offer some of our experience in teaching courses with this book. Since 2006, the original ECE498AL course has become a permanent course known as ECE408 or CS483 at the University of Illinois at Urbana-Champaign. We started writing some of the early chapters of this book when we offered ECE498AL the second time. Since then, we have used the book for numerous offerings of ECE408 as well as the Coursera Heterogeneous Parallel Programming course, and the VSCSE and PUMPS summer schools.

A two-phased approach

Most of the chapters in the book are designed to be given in approximately a single 75 min lecture each. The chapters that may need two 75 min lectures to be fully covered are Chapter 11 (Scan), Chapter 17 (Sparse Matrix Computation), Chapter 18 (Graph Traversal) and Chapter 19. In ECE408, the lectures, programming assignments, and final project are paced with each other and are organized into two phases.

In the first phase, which consists of Parts 1 and 2 of this book, students learn the fundamentals and basic patterns, and they practice the skills they learn via guided programming assignments. This phase consists of 14 chapters and typically takes around eight weeks, where each week students work on a programming assignment corresponding to the lectures covered that week. For example, in the first week, a lecture based on Chapter 2 is dedicated to teaching the basic CUDA memory/threading model, the CUDA extensions to the C++ language, and the basic programming tools. After that lecture, students can write a simple vector addition code in a couple of hours.

The following two weeks include a series of four lectures based on Chapters 3 through Chapter 6 that give students conceptual understanding of the CUDA memory model, the CUDA thread execution model, GPU hardware performance features, and modern computer system architecture. During these two weeks, students work on different implementations of matrix-matrix multiplication where they see the performance of their implementations increases throughout this period. In the remaining five weeks, the lectures cover common data-parallel programming patterns needed to develop a high-performance parallel application based on Chapters 7 through Chapter 14. Throughout these weeks, students complete assignments on convolution, histogram, reduction, and scan. By the end of the first phase, students should be quite comfortable with parallel programming and should be ready to implement more advanced code with less handholding.

In the second phase, which is based on Part 3, students learn advanced patterns and applications while they themselves work on a final project that involves accelerating an advanced pattern or application. Although we do not usually assign weekly programming assignments during this phase, the project typically has a weekly milestone to help students pace themselves. Depending on the duration and format of the course, instructors may be able to cover only a small subset of the chapters in this phase and may need to skip some. Instructors may also choose to replace some lectures with guest lectures, paper discussion sessions, or lectures that support the final project. For this reason, Fig. P.1 uses arrows to indicate the dependencies between chapters to assist instructors in selecting which chapters they can skip or reorder to customize the course for their local context.

Tying it all together: the final project

While the lectures, labs, and chapters of this book help lay the intellectual foundation for the students, what brings the learning experience together is the final project. The final project is so important to the full-semester course that it is prominently positioned in the course and commands nearly two months' focus. It incorporates five innovative aspects: mentoring, workshop, clinic, final report, and symposium. While much of the information about the final project is available in the NVIDIA Accelerated Computing Teaching Kit, we would like to offer the reasoning that was behind the design of these aspects.

Students are encouraged to base their final projects on problems that represent current challenges in the research community. To seed the process, the instructors should recruit several computational science research groups to propose problems and serve as mentors. The mentors are asked to contribute a one-to-two-page project specification sheet that briefly describes the significance of the application, what the mentor would like to accomplish with the student teams on the application, the technical skills (particular type of Math, Physics, Chemistry courses) required to understand and work on the application, and a list of web and traditional resources that students can draw upon for technical background, general information, and building blocks, along with specific URLs or ftp paths to particular implementations and coding examples. These project specification sheets also provide students with learning experiences in defining their own research projects later in their careers. Several examples are available in the Teaching Kit.

The design document

Once the students decide on a project and form a team, they submit a design document for the project. This helps them to think through the project steps before they jump into it. The ability to do such planning will be important to their career success. The design document should discuss the background and motivation for the project, application-level objectives and potential impact, main features of the end application, an overview of their design, an implementation plan, their performance goals, a verification plan and acceptance test, and a project schedule.

The project report and symposium

Students are required to submit a project report on their team's key findings. We also recommend a whole-day class symposium. During the symposium, students use presentation slots proportional to the size of the teams. During the presentation, the students highlight the best parts of their project report for the benefit of the whole class. The presentation accounts for a significant part of the students' grades. Each student must answer questions directed to the student individually, so that different grades can be assigned to individuals in the same team. The symposium is an opportunity for students to learn to produce a concise presentation that motivates their peers to read a full paper.

Class competition

In 2016, the enrollment level of ECE408 far exceeded the level that can be accommodated by the final project process. As a result, we moved from the final project to a class competition. At the middle of the semester, we announce a competition challenge problem. We use one lecture to explain the competition challenge problem and the rules that will be used for ranking the teams. All student submissions are autograded and ranked. The final ranking of each team is determined by the execution time, correctness, and clarity of their parallel code. The students do a demo of their solution at the end of the semester and submit a final report. This compromise preserves some of the benefits of final projects when the class size makes final projects infeasible.

Course resources

The NVIDIA Accelerated Computing Teaching Kit is a publicly available resource that contains lecture slides, recordings of past lectures, lab assignments, final project guidelines, and sample project specifications for instructors who use this book for their classes. The lecture slides and lab assignments are also available at the companion site of the textbook, and the recordings of past course offerings are available on the book's YouTube channel. We are also in the process of making the sample codes from the book's figures available with their associated host codes for instructors who would like to demo the code in class, or general readers who would like to experiment with the code themselves. While this book provides the intellectual contents for these classes, the additional material is crucial in achieving the overall education goals.

Finally, we would like to hear from you if you have any ideas for improving this book. We would also like to know how we can improve the supplementary on-line material. Of course, we always love to know what you like about the book. We hope to hear from you soon.

Wen-mei W. Hwu
David B. Kirk
Izzat El Hajj

Acknowledgments

There are so many people who have made special contributions to the fifth edition. We would like to first thank the contributing chapter co-authors. Their names are listed in the chapters to which they made special contributions. Their expertise made a tremendous difference in the technical contents of this new edition. Without the expertise and contribution of these individuals, we would not have been able to cover the topics with the level of insight that we wanted to provide to our readers.

We would like to especially thank Juan Gómez-Luna who played a major role in the writing of the fifth edition. Juan has co-authored chapters in previous editions, co-authored two new chapters in the fifth edition, developed slides and code examples for these chapters, assisted in reviewing other chapters, and engaged with us in many fruitful discussions. We thank Juan for all his tireless efforts in making this edition possible.

We would like to especially acknowledge Ian Buck, the father of CUDA, and John Nickolls, the lead architect of the original Tesla GPU Computing Architecture. Their teams created the initial hardware and software infrastructure for this book and its associated course. Many engineers and researchers at NVIDIA have also contributed to the rapid advancement of CUDA that supports the efficient implementation of advanced parallel patterns. John passed away while we were working on the second edition. We miss him dearly.

Our external reviewers spent numerous hours of their precious time to give us insightful feedback. Our reviewers in the fifth edition include: Aart Bik (NVIDIA), Federico Busato (NVIDIA), Cris Cecka (NVIDIA), Guillaume Thomas Collignon (NVIDIA), Abdul Dakkak (Modular), Georgii Evtushenko (NVIDIA), Charles Frye (Modal), Jake Hemstad (NVIDIA), CJ Newburn (NVIDIA), and Jay Shah (Colfax International). In addition, Anshuman Bhat (NVIDIA) and Katrina Riehl (NVIDIA) provided valuable insights and support. Our reviewers since the third edition include: Sonia Lopez Alarcon (Rochester Institute of Technology), Bedrich Benes (Purdue University), Bryan Chin (University of California San Diego), Samuel Cho (Wake Forest University), Kevin Farrell (Institute of Technology Blanchardstown, Dublin Ireland), Lahouari Ghouti (King Fahd University of Petroleum and Minerals, Saudi Arabia), Marisa Gil (Universitat Politecnica de Catalunya, Barcelona, Spain), Karen L. Karavanic (Portland State University), Steve Lumetta (University of Illinois at Urbana-Champaign), Vikram Mailthody (NVIDIA), Dejan Milojici (Hewlet-Packard Labs), Pinar Muyan Ozcelik (California State University, Sacramento), Greg Peterson (The University of Tennessee-Knoxville), Zaid Qureshi (NVIDIA), José L. Sánchez (University of Castilla-La Mancha), Janche Sang (Cleveland State University), and Jan Verschelde (University of Illinois at Chicago). The comments of all our reviewers helped us to significantly improve the content and readability of the book.

Steve Merken, Prashant Mahajan, Surya Narayanan Jayachandran, and their staff at Elsevier worked tirelessly on this project.

We would like to especially thank Jensen Huang for providing a great amount of financial and human resources for developing the course that laid the foundation for this book.

We would like to acknowledge Dick Blahut and Dan Vivoli who challenged David to come to Illinois and create the original ECE498AL course with Wen-mei. We would like to especially thank our colleagues Kurt Akeley, Al Aho, Arvind, Dick Blahut, Randy Bryant, Bob Colwell, Bill Dally, Ed Davidson, Joel Emer, Steve Fields, Mike Flynn, Michael Garland, John Hennessy, Pat Hanrahan, Nick Holonyak, Stephen Jones, Dick Karp, Kurt Keutzer, Arvind Krishna, Chris Lamb, Dave Liu, David Luebke, Dave Kuck, Nacho Navarro, Sanjay Patel, Yale Patt, David Patterson, Bob Rao, Burton Smith, Jim Smith, Guri Sohi, and Mateo Valero who have taken the time to share their insight with us over the years. We are humbled by the generosity and enthusiasm of all the great people who contributed to the course and the book.

Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj

CHAPTER

Introduction 1

CONTENTS

1.1 Heterogeneous parallel computing 3
1.2 Why more speed or parallelism? 7
1.3 Speeding up real applications 9
1.4 Challenges in parallel programming 11
1.5 Related parallel programming interfaces 12
1.6 Overarching goals 14
1.7 Organization of the book 15
References 18

Ever since the beginning of computing, many high-valued applications have demanded more execution speed and resources than the computing devices can offer. Early applications relied on the advancement of processor speed, memory speed, and memory capacity to enhance application-level capabilities such as the timeliness of weather forecast, the accuracy of engineering structural analysis, the realism of computer-generated graphics, the number of airline reservations processed per second, or the number of fund transfers processed per second. More recently, new applications such as deep learning have demanded even more execution speed and resources than what the best computing devices can offer. These application demands have fueled fast advancement in computing device capabilities in the past five decades and will continue to do so in the foreseeable future.

Microprocessors based on a single central processing unit (CPU) that appear to execute instructions in sequential steps, such as those in the x86 processors from Intel and AMD, armed with fast increasing clock frequency and hardware resources, drove rapid performance increases and cost reductions in computer applications in the 1980s and 1990s. During these two decades of growth, single-CPU microprocessors brought GFLOPS, or giga (10^9) floating-point operations per second, to the desktop computers and TFLOPS, or tera (10^{12}) floating-point operations per second to datacenters. This relentless drive for performance improvement has allowed application software to provide more functionality, have better user interfaces, and generate more useful results. The users, in turn, demand even more improvements once they become accustomed to these improvements, creating a positive (virtuous) cycle for the computer industry.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00009-3
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

This drive, however, slowed down since 2003 due to energy consumption and heat dissipation issues. These issues limited the increase of the clock frequency and the productive activities that could be performed in each clock period within a single CPU while maintaining the appearance of executing instructions in sequential steps. Since then, virtually all microprocessor vendors switched to a model where multiple physical CPUs, referred to as processor cores, are used in each chip to increase its processing power. A traditional CPU can be viewed as a single-core CPU in this model. In order to benefit from the multiple processor cores, users must have multiple instruction sequences, whether from the same application or different applications, that can simultaneously execute on these processor cores. For a particular application to benefit from multiple processor cores, its work must be divided into multiple instruction sequences that can simultaneously execute on these processor cores. This switch from a single CPU executing instructions in sequential steps to multiple cores executing multiple instruction sequences in parallel had a tremendous impact on the software developer community.

Traditionally, the vast majority of software applications were written as sequential programs that were executed by processors whose design was envisioned by von Neumann in his seminal report in 1945 [1]. The execution of these programs can be understood by a human as sequentially stepping through the program code based on the concept of a program counter, also known as an instruction pointer in the literature. The program counter contains the memory address of the next instruction that will be executed by the processor. The sequence of instruction execution activities resulting from this sequential stepwise execution of an application is referred to as a **thread** of execution, or simply a thread, in the literature. The concept of threads is so important that it will be more formally defined and used extensively in the rest of this book.

Historically, most software developers relied on the advances in hardware, such as increased clock speed and executing multiple instructions under the hood, to increase the speed of their sequential applications; the same software simply ran faster as each new processor generation was introduced. Computer users were also accustomed to the expectation that these programs ran faster with each new generation of microprocessors. This expectation has not been valid for over a decade. A sequential program only runs on one of the processor cores, which is no longer becoming significantly faster from generation to generation. Without performance improvement, sequential application developers were no longer able to introduce new features and capabilities into their software as new microprocessors were introduced, reducing the growth opportunities of the entire computer industry.

Rather, the application software that continued to enjoy significant performance improvement with each new generation of microprocessors was **parallel** software, in which multiple threads of execution cooperate to complete the work faster. This dramatically escalated advantage of parallel programs over sequential programs was referred to as the concurrency revolution [2]. The practice of parallel programming was by no means new. The high-performance computing community had been developing parallel programs for decades. These parallel programs typically ran on large

scale, expensive computers. Only a few elite applications could justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers. However, when all microprocessors became parallel computers, the number of applications that needed to be developed as parallel programs increased dramatically. There is a greater need for software developers to learn about parallel programming, which is the focus of this book.

1.1 Heterogeneous parallel computing

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessors [3]: multi-core and many-thread. The *multi-core* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. Multi-core processors began with two-core processors with the number of cores increasing with each semiconductor process generation. For example, Intel and AMD multi-core server microprocessor can have over 100 processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full x86 instruction set, supporting hyper-threading with two hardware threads, designed to maximize the execution speed of sequential programs. Another example is the ARM Ampere multicore server processor with up to 128 processor cores.

In contrast, the *many-thread* trajectory focuses more on the execution throughput of parallel applications. The many-thread trajectory began with a large number of threads and once again, the number of threads has increased with each generation. A recent exemplar is the NVIDIA Hopper H100 Graphics Processing Unit (GPU) with hundreds of thousands of threads, executing in a large number of simple in-order pipelines. Many-thread processors, especially GPUs, have led the race of floating-point performance since 2003. As of 2022, the peak floating-point throughput of the H100 GPU is 34 TFLOPS for 64-bit double precision, 67 TFLOPS for 32-bit single-precision, and 1979 TFLOPS for 16-bit half-precision. In comparison, the typical peak floating-point throughput of a server grade CPU in the same year is only a few TFLOPS. The ratio of peak floating-point calculation throughput between many-thread GPUs and multi-core CPUs has been increasing for the past several years. These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips.

Such a large gap of peak performance between multi-core and many-thread processors has amounted to a significant "electrical potential" build-up, and at some point, something will have to give. We have reached that point. To date, this large peak performance gap has already motivated many applications developers to move the computationally intensive parts of their software to GPUs for execution. Perhaps even more importantly, the drastically elevated performance of parallel execution has enabled revolutionary new applications such as deep learning that are intrinsically composed of computationally intensive parts. Not surprisingly, these computationally intensive parts are also the prime target of parallel programming – when there

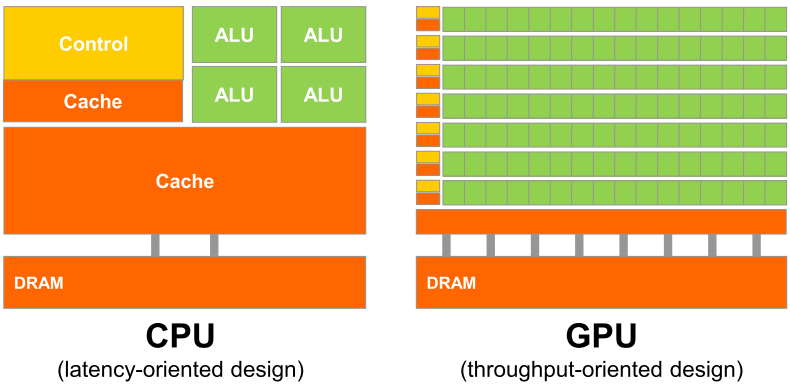


FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies.

is more work to do, there is more opportunity to divide the work among cooperating parallel workers, i.e., threads.

One might ask why there is such a large peak performance gap between many-threaded GPUs and multi-core CPUs. The answer lies in the differences in the fundamental design philosophies between the two types of processors, as illustrated in Fig. 1.1. The design of a CPU, as shown in Fig. 1.1(a), is optimized for sequential code performance. The arithmetic units and operand data delivery logic are designed to minimize the effective latency of arithmetic operations at the cost of increased use of chip area and power per unit. Large last-level on-chip caches are designed to capture frequently accessed data and convert some of the long-latency memory accesses into short-latency cache accesses. Sophisticated branch prediction logic and execution control logic are used to mitigate the latency of conditional branch instructions. By reducing the latency of operations, the CPU hardware reduces the execution latency of each individual thread. However, the low-latency arithmetic units, sophisticated operand delivery logic, large cache memory, and complex control logic consume chip area and power that could be otherwise used to provide more arithmetic execution units and memory access channels. This design approach is commonly referred to as latency-oriented design.

The design philosophy of the GPUs, on the other hand, has been shaped by the fast-growing video game industry that exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations and memory accesses per video frame in advanced games. This demand motivates GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations and memory access throughput.

The need for performing a massive number of floating-point calculations per second in graphics applications for tasks such as viewpoint transformations and object rendering is quite intuitive. Additionally, the need for performing a massive number of memory accesses per second is just as and perhaps even more important. The speed

of many graphics applications is limited by the rate at which data can be delivered from the memory system into the processors and vice versa. A GPU must be capable of moving extremely large amounts of data in and out of graphics frame buffers in its DRAM (Dynamic Random-Access Memory) because such movement is what makes video displays rich and satisfying to gamers. The relaxed memory model (the way various system software, applications, and I/O devices expect how their memory accesses work) commonly accepted by game applications also makes it easier for the GPUs to support massive parallelism in accessing memory.

In contrast, general-purpose processors must satisfy requirements from legacy operating systems, applications, and I/O devices that present more challenges to supporting parallel memory accesses and thus make it more difficult to increase the throughput of memory accesses, commonly referred to as **memory bandwidth**. As a result, graphics chips have been operating at approximately 10x the memory bandwidth of contemporaneously available CPU chips and we expect that GPUs will continue to be at an advantage in terms of memory bandwidth for some time.

An important observation is that reducing latency is much more expensive than increasing throughput in terms of power and chip area. For example, one can double the arithmetic throughput by doubling the number of arithmetic units at the cost of doubling the chip area and power consumption. However, reducing the arithmetic latency by half may require doubling the current at the cost of more than doubling the chip area used and quadrupling the power consumption. Therefore, the prevailing solution in GPUs is to optimize for the execution throughput of massive numbers of threads, rather than reducing the latency of individual threads. This design approach saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency. The reduction in area and power of the memory access hardware and arithmetic units allows the GPU designers to have more of them on a chip and thus increase the total execution throughput. Fig. 1.1 visually illustrates the difference in the design approaches by showing a smaller number of larger arithmetic units and a smaller number of memory channels in the CPU design in Fig. 1.1(a), in contrast to the larger number of smaller arithmetic units and a larger number of memory channels in Fig. 1.1(b).

The application software for these GPUs is expected to be written with a large number of parallel threads. The hardware takes advantage of the large number of threads to find work to do when some of them are waiting for long-latency memory accesses or arithmetic operations. Small cache memories in Fig. 1.1(b) are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not all need to go to the DRAM. This design style is commonly referred to as throughput-oriented design as it strives to maximize the total execution throughput of a large number of threads while allowing individual threads to take a potentially much longer time to execute.

It should be clear that GPUs are designed as parallel, throughput-oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well. For programs that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs.

When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs. Therefore, one should expect that many applications use both CPUs and GPUs, executing sequential parts of the application on the CPU and numerically intensive parts on the GPUs. This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPU-GPU execution of an application.

It is also important to note that speed is not the only decision factor when application developers choose the processors for running their applications. Several other factors can be even more important. First and foremost, the processors of choice must have a very large presence in the market place, referred to as the *installed base* of the processor. The reason is very simple. The cost of software development is best justified by a very large customer population. Applications that run on a processor with a small market presence will not have a large customer base. This has been a major problem with traditional parallel computing systems that have negligible market presence compared to general-purpose microprocessors. Only a few elite applications funded by the government and large corporations have been successfully developed on these traditional parallel computing systems. This has changed with many-thread GPUs. Due to their popularity in the PC market, GPUs have been sold by the hundreds of millions. Virtually all desktop PCs and high-end laptops have GPUs in them. There are more than 1 billion CUDA enabled GPUs in use to date. Such a large market presence has made these GPUs economically attractive targets for application developers.

Another important decision factor is practical form factors and easy accessibility. Until 2006, parallel software applications ran on data center servers or departmental clusters. But such execution environments tend to limit the use of these applications. For example, in an application such as medical imaging, it is fine to publish a paper based on a 64-node cluster machine. But actual clinical applications on MRI machines have been based on some combination of a PC and special hardware accelerators. The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks of compute server boxes into clinical settings, while this is common in academic departmental settings. In fact, the National Institute of Health (NIH) refused to fund parallel programming projects for some time: they felt that the impact of parallel software would be limited because huge cluster-based machines would not work in the clinical setting. Today, many companies ship MRI products with GPUs and NIH funds research using GPU computing.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics API (Application Programming Interface) functions to access the processing units, meaning that OpenGL or Direct3D techniques were needed to program these chips. Stated more simply, a computation must be expressed as a function that paints a pixel in some way in order to execute on these early GPUs. This technique was called GPGPU, for General Purpose Programming using a Graphics Processing Unit. Even with a higher level programming environment, the underlying code still needs to fit into the APIs that are designed to paint pixels. These APIs limit the kinds of applications that one can actually write for early GPUs.

Consequently, GPGPU did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent research results.

But everything changed in 2007 with the release of CUDA [4]. CUDA did not represent software changes alone, but additional hardware was added to the chip. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming. In the G80 and its successor chips for parallel computing, GPGPU programs no longer go through the graphics interface at all. Instead, a new general-purpose parallel programming interface on the silicon chip serves the requests of CUDA programs. The general-purpose programming interface greatly expands the types of applications that one can easily develop for GPUs. Moreover, all the other software layers were redone as well, so that the programmers can use the familiar C/C++ programming tools.

While GPUs are an important class of computing devices in heterogeneous parallel computing, there are other important types of computing devices used as accelerators in heterogeneous computing systems. For example, Field Programmable Gate Array (FPGAs) have been widely used to accelerate networking applications. The techniques covered in this book using GPUs as the learning vehicle also apply to the programming tasks for such accelerators.

1.2 Why more speed or parallelism?

As we stated in Section 1.1, the main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations. As we will discuss in the Parallel Patterns, Advanced Patterns, and Applications chapters (Parts 2 and 3, Chapters 7 through Chapter 23), when an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100× (100 times) speedup over sequential execution on a single CPU core. If the application includes what we call "data parallelism", it's often possible to achieve a 10× speedup with just a few hours of work.

One might ask why applications will continue to demand increased speed. Many applications that we have today seem to be running fast enough. Despite the myriad of computing applications in today's world, many exciting mass market applications of the future are what we previously consider "supercomputing applications", or super-applications. For example, the biology research community is moving more and more into the molecular level. Microscopes, arguably the most important instrument in molecular biology, used to rely on optics or electronic instrumentation. But there are limitations to the molecular-level observations that we can make with these instruments. These limitations can be effectively addressed by incorporating a computational model to simulate the underlying molecular activities with boundary conditions set by traditional instrumentation. With simulation we can measure even more details and test more hypotheses than can ever be imagined with traditional instrumentation alone. These simulations will continue to benefit from the increasing

computing speed in the foreseeable future in terms of the size of the biological system that can be modeled and the length of reaction time that can be simulated within a tolerable response time. These enhancements will have tremendous implications to science and medicine.

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition (HD) TV vs. older NTSC TV. Once we experience the level of details in an HDTV, it is very hard to go back to older technology. But consider all the processing that is needed for that HDTV. It is a highly parallel process, as are 3D imaging and visualization. In the future, new functionalities such as view synthesis and high-resolution display of low resolution videos will demand more computing power in the TV. At the consumer level, we are seeing an increasing number of video and image processing applications that improve the focus, lighting, and other key aspects of the pictures and videos.

Among the benefits offered by more computing speed are much better user interfaces. Modern smart phone users enjoy a much more natural interface with high-resolution touch screens that rival a large-screen TV. Undoubtedly, future versions of these devices will incorporate sensors and displays with three-dimensional perspectives, applications that combine virtual and physical space information for enhanced usability, and voice and computer vision based interfaces, requiring even more computing speed.

Similar developments are underway in consumer electronic gaming. In the past, driving a car in a game was in fact simply a prearranged set of scenes. If your car bumped into an obstacle, the course of your vehicle did not change; only the game score changed. Your wheels were not bent or damaged, and it was no more difficult to drive, regardless of whether you bumped your wheels or even lost a wheel. With increased computing speed, the games can be based on dynamic simulation rather than pre-arranged scenes. We can expect to experience more of these realistic effects in the future: accidents will damage your wheels and your online driving experience will be much more realistic. The ability to accurately model physical phenomena has already inspired the concept of *digital twins*, where physical objects have accurate models in the simulated space so that stress testing and deterioration prediction can be thoroughly conducted at much lower cost. Realistic modeling and simulation of physics effects are known to demand very large amounts of computing power.

An important example of new applications enabled by drastically increased computing throughput is deep learning based on artificial neural networks. While neural networks have been actively researched since the 1970s, they were ineffective in practical applications because it took too much labeled data and too much computation to train them. The rise of internet offered a tremendous number of labeled pictures and the rise of GPUs offered a surge of computing throughput. As a result, there has been a fast adoption of neural network based applications in computer vision and natural language processing since 2012. This adoption revolutionized computer vision and natural language processing applications and triggered fast development of self-driving cars and home assistant devices.

All the new applications that we mentioned involve simulating and/or representing a physical and concurrent world in different ways and at different levels, with

tremendous amounts of data being processed. With this huge quantity of data, much of the computation can be done on different parts of the data in parallel, although they will have to be reconciled at some point. In most cases, effective management of data delivery can have a major impact on the achievable speed of a parallel application. While techniques for doing so are often well known to a few experts who work with such applications on a daily basis, the vast majority of application developers can benefit from more intuitive understanding and practical working knowledge of these techniques.

We aim to present the data management techniques in an intuitive way to application developers whose formal education may not be in computer science or computer engineering. We also aim to provide many practical code examples and hands-on exercises that help the reader to acquire working knowledge, which requires a practical programming model that facilitates parallel implementation and supports proper management of data delivery. CUDA offers such a programming model and has been well tested by a large developer community.

1.3 Speeding up real applications

How much speedup can we expect from parallelizing an application? The definition of **speedup** for an application by computing system A over computing system B is the ratio of the time used to execute the application in B over the time used to execute the same application in A. For example, if an application takes 10 seconds to execute in system A but takes 200 seconds to execute in System B, the speedup for the execution by system A over system B would be $(200s)/(10s) = 20$, which is referred to as a $20\times$ (20 times) speedup.

The speed up achievable by a parallel computing system over a serial computing system depends on the portion of the application that can be parallelized. For example, if the percentage of time spent in the part that can be parallelized is 30%, a 100x speedup of the parallel portion will reduce the total execution time of the application by no more than 29.7%. That is, the speedup for the entire application will be only about $1/(1-0.297) = 1.42\times$. In fact, even an infinite amount of speedup in the parallel portion can only slash 30% off execution time, achieving no more than $1.43\times$ speedup. On the other hand, if 99% of the execution time is in the parallel portion, a $100\times$ speedup of the parallel portion will reduce the application execution to 1.99% of the original time. This gives the entire application a $50\times$ speedup. Therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speedup its execution. The fact that the level of speedup one can achieve through parallel execution can be severely limited by the size of the parallelizable portion of the application is referred to as **Amdahl's Law** [5].

Researchers have achieved speedups of more than $100\times$ for some applications. However, this is typically achieved only after extensive optimization and tuning after

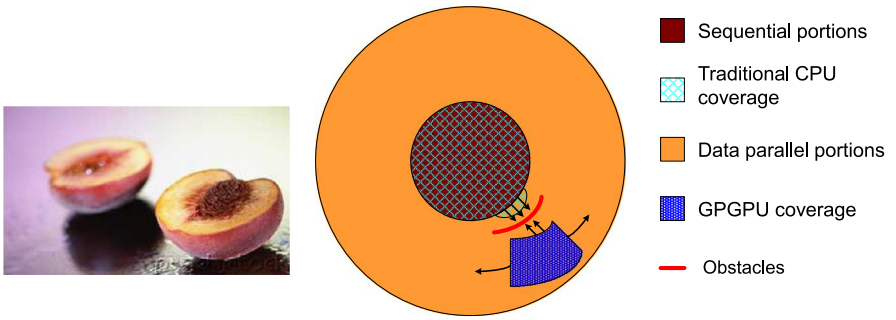


FIGURE 1.2

Coverage of sequential and parallel application portions. The sequential portions and the traditional (single-core) CPU coverage portions overlap with each other. The previous GPGPU technique offers very limited coverage of the data parallel portions since it is limited to computations that can be formulated into painting pixels. The obstacles refer to the power constraints that make it hard to extend single-core CPUs to cover more of the data parallel portions.

the algorithms have been enhanced so that more than 99.9% of the application work is in the parallel portion.

Another important factor for the achievable level of speedup for applications is how fast data can be accessed from and written to the memory. In practice, straightforward parallelization of applications often saturates the memory (DRAM) bandwidth, resulting in only about a 10× speedup. The trick is to figure out how to get around memory bandwidth limitations, which involves doing one of many transformations to utilize specialized GPU on-chip memories to drastically reduce the number of accesses to the DRAM. One must, however, further optimize the code to get around limitations such as limited on-chip memory capacity. An important goal of this book is to help the reader to fully understand these optimizations and become skilled at them.

Keep in mind that the level of speedup achieved over single core CPU execution can also reflect the suitability of the CPU to the application. In some applications, CPUs perform very well, making it harder to speed up performance using a GPU. Most applications have portions that can be much better executed by the CPU. One must give the CPU a fair chance to perform and make sure that code is written so that GPUs complement CPU execution, thus properly exploiting the heterogeneous parallel computing capabilities of the combined CPU/GPU system. As of today, mass-market computing systems that combine multi-core CPUs and many-core GPUs have brought terascale computing to the desktops and exascale computing to clusters.

Fig. 1.2 illustrates the main parts of a typical application. Much of a real application's code tends to be sequential. These sequential parts are illustrated as the "pit" area of the peach: trying to apply parallel computing techniques to these portions is like biting into the peach pit – it is not a good feeling! These portions are very hard to

parallelize. CPUs tend to do a very good job on these portions. The good news is that these portions, although they can take up a large portion of the code, tend to account for only a small portion of the execution time of super-applications.

Then come what we call the "peach meat" portions. These portions are easy to parallelize, as are some early graphics applications. Parallel programming in heterogeneous computing systems can drastically improve the speed of these applications. As illustrated in Fig. 1.2, early GPGPUs programming interfaces cover only a small portion of the meat section, which is analogous to a small portion of the most exciting applications. As we will see, the CUDA programming interface is designed to cover a much larger section of the peach meat of exciting applications. Parallel programming models and their underlying hardware are still evolving at a fast pace in order to enable efficient parallelization of even larger sections of applications.

1.4 Challenges in parallel programming

What makes parallel programming hard? Someone once said that if you do not care about performance, parallel programming is very easy. You can literally write a parallel program in an hour. But then why bother to write a parallel program if you do not care about performance?

This book addresses several challenges in achieving high performance in parallel programming. First and foremost, it can be challenging to design parallel algorithms with the same level of algorithmic (computational) complexity as sequential algorithms. Many parallel algorithms perform the same amount of work as their sequential counterpart. However, some parallel algorithms do more work than their sequential counter parts. In fact, sometimes they may do so much more work that they could end up running slower for large input datasets. This is especially a problem because fast processing of large input datasets is an important motivation for parallel programming.

For example, many real-world problems are most naturally described with mathematical recurrences. Parallelizing these problems often require non-intuitive ways of thinking about the problem and may require redundant work during execution. There are important algorithm primitives such as prefix-sum or scan that can facilitate the conversion of sequential, recursive formulations of problems into more parallel forms. We will more formally introduce the concept of *work efficiency* and will illustrate the methods and tradeoffs involved in designing parallel algorithms that achieve the same level of computational complexity as their sequential counter parts, using important parallel patterns such as prefix-sum in Chapter 11.

Second, the execution speed of many applications is limited by memory access latency and/or throughput. We refer to these applications as memory-bound, as opposed to compute bound, which are limited by the rate at which the hardware can perform arithmetic operations. Achieving high-performance parallel execution in memory-bound applications often requires methods for improving memory access speed. We will introduce optimization techniques for memory accesses in Chapter 5

and Chapter 6, and apply these techniques in several chapters on parallel patterns and applications.

Third, the execution speed of parallel programs is often more sensitive to the input data characteristics than their sequential counter parts. Many real-world applications need to deal with inputs with widely varying characteristics, such as erratic or unpredictable data sizes, and uneven data distributions. These variations of sizes and distributions can cause uneven amount of work to be assigned to the parallel threads and significantly reduce the effectiveness of parallel execution. The performance of parallel programs can sometimes vary dramatically with these characteristics. We will introduce techniques for regularizing data distributions or tailoring the assignment of data to threads to address these challenges in the chapters that introduce parallel patterns and applications.

Fourth, some applications can be parallelized while requiring little collaboration across different threads. These applications are often referred to as **embarrassingly parallel**. On the other hand, other applications require threads to collaborate with each other, which requires using *synchronization operations* such as barriers or atomic operations. These synchronization operations impose overhead on the application because threads often find themselves waiting for other threads instead of performing useful work. We will discuss various strategies for reducing this synchronization overhead throughout this textbook.

Fortunately, most of these challenges have been addressed by researchers in the past. There are also common patterns across application domains that allow us to apply solutions derived in one domain to others. This is the primary reason why we will be presenting key techniques for addressing these challenges in the context of important parallel computation patterns and applications.

1.5 Related parallel programming interfaces

Many parallel programming languages and models have been developed in the past several decades [6]. The ones that are the most widely used are OpenMP [7] for shared memory multiprocessor systems and the Message Passing Interface (MPI) [8] for scalable cluster computing. Both have become standardized programming interfaces supported by major computer vendors.

An OpenMP implementation consists of a compiler and a runtime. A programmer specifies directives (commands) and pragmas (hints) about a loop to the OpenMP compiler. With these directives and pragmas, OpenMP compilers generate parallel code. The runtime system supports the execution of the parallel code by managing parallel threads and resources. OpenMP was originally designed for CPU execution and has been extended to support GPU execution. The major advantage of OpenMP is that it provides compiler automation and runtime support for abstracting away many parallel programming details from programmers. Such automation and abstraction can help make the application code more portable across systems produced by different vendors as well as different generations of systems from the same vendor. We

refer to this property as "performance portability". However, effective programming in OpenMP still requires the programmers to understand all the detailed parallel programming concepts involved. Because CUDA gives programmers explicit control of these parallel programming details, it is an excellent learning vehicle even for someone who would like to use OpenMP as their primary programming interface. Furthermore, from our experience, OpenMP compilers are still evolving and improving. Many programmers will likely need to use CUDA style interfaces for parts where OpenMP compilers fall short.

MPI is a programming interface where computing nodes in a cluster do not share memory. All data sharing and interaction must be done through explicit message passing. MPI has been widely used in high-performance computing (HPC). Applications written in MPI have run successfully on cluster computing systems with more than 100,000 nodes. Today, many HPC clusters employ heterogeneous CPU/GPU nodes. The amount of effort needed to port an application into MPI can be quite high due to the lack of shared memory across computing nodes. The programmer needs to perform **domain decomposition** to partition the input and output data across individual nodes. Based on the domain decomposition, the programmer also needs to call message sending and receiving functions to manage the data exchange between nodes. CUDA, on the other hand, provides shared memory for parallel execution in the GPU to address this difficulty. While CUDA is an effective interface within each node, most application developers need to use an interface like MPI to program at the cluster level. There has been increasing support for multi-GPU programming in CUDA with MPI itself becoming CUDA-aware and with other APIs like NCCL and NVSHMEM. It is therefore important that a parallel programmer in HPC understands how to do CUDA programming in the context of these other multi-GPU programming models in modern computing clusters employing multi-GPU nodes, which is presented in Chapter 23.

In 2009, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA jointly developed a standardized programming model called Open Compute Language (OpenCL) [9]. Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors. In comparison to CUDA, OpenCL relies more on APIs and less on language extensions. This allows vendors to quickly adapt their existing compilers and tools to handle OpenCL programs. OpenCL is a standardized programming model in that applications developed in OpenCL can run correctly without modification on all processors that support the OpenCL language extensions and API. However, one will likely need to modify the applications in order to achieve high performance for a new processor.

Those who are familiar with both OpenCL and CUDA know that there is a remarkable similarity between the key concepts and features of OpenCL and those of CUDA. That is, a CUDA programmer can learn OpenCL programming with minimal effort. More importantly, virtually all techniques learned using CUDA can be easily applied to OpenCL programming.

1.6 Overarching goals

Our primary goal is to teach you, the reader, how to program massively parallel processors to achieve high performance. Thus, much of the book is dedicated to the techniques for developing high-performance parallel code. Our approach will not require a great deal of hardware expertise. Nevertheless, you will need to have a good conceptual understanding of the parallel hardware architectures to be able to reason about the performance behavior of your code. Therefore, we are going to dedicate some pages to the intuitive understanding of essential hardware architecture features and many pages to techniques for developing *high-performance* parallel programs. In particular, we will focus on *computational thinking* [10] techniques that will enable you to think about problems in ways that are amenable to high-performance execution on massively parallel processors.

High-performance parallel programming on most processors requires some knowledge of how the hardware works. It will probably take many years to build tools and machines so that programmers can develop high-performance code without this knowledge. Even if we have such tools, we suspect that programmers who have knowledge of the hardware will be able to use the tools much more effectively than those who do not. For this reason, we dedicate Chapter 4 to introduce the fundamentals of the GPU architecture. We also discuss more specialized architecture concepts as part of our discussions on high-performance parallel programming techniques.

Our second goal is to teach parallel programming for correct functionality and reliability, which constitutes a subtle issue in parallel computing. Those who have worked on parallel systems in the past know that achieving initial performance is not enough. The challenge is to achieve it in such a way that you can debug the code and support users. The CUDA programming model encourages the use of simple forms of barrier synchronization, atomicity, and memory consistency for managing parallelism. In addition, it provides an array of powerful tools that allow one to debug not only the functional aspects but also the performance bottlenecks.

Our third goal is scalability across future hardware generations by exploring approaches to parallel programming such that future machines, which will be more and more parallel, can run your code faster than today's machines. We want to help you to master parallel programming so that your programs can scale up to the level of performance of new generations of machines. The key to such scalability is to regularize and localize memory data accesses to minimize consumption of critical resources and conflicts in updating data structures. Therefore, the techniques for developing high-performance parallel code are also important for ensuring future scalability of applications.

Much technical knowledge will be required to achieve these goals, so we will cover quite a few principles and patterns [6] of parallel programming in this book. We will not be teaching these principles and patterns on their own. We will teach them in the context of parallelizing useful applications. We cannot cover all of them however, so we have selected the most useful and well-proven techniques to cover in detail. In fact, the current edition has a significantly expanded number of chapters on parallel patterns and applications. We are now ready to give you a quick overview of the rest of the book.

1.7 Organization of the book

This book is organized into three parts. Part 1 covers fundamental concepts in parallel programming, data parallelism, GPUs, and performance optimization. These foundational chapters equip the reader with the basic knowledge and skills necessary for becoming a GPU programmer. Part 2 covers primitive parallel patterns and Part 3 covers more advanced parallel patterns and applications. These two parts apply the knowledge and skills learned in the first part, and also introduce other GPU architecture features and optimization techniques as the need for them arises.

Part 1 on fundamental concepts consists of Chapter 2 to Chapter 6. Chapter 2 introduces data parallelism and the CUDA programming model. This chapter relies on the fact that the reader has had previous experience with C++ programming. It first introduces CUDA as a simple, small extension to C++ that supports heterogeneous CPU/GPU computing and the widely used Single Program Multiple Data (SPMD) parallel programming model. It then covers the thought process involved in (1) identifying the part of application programs to be parallelized, (2) isolating the data to be used by the parallelized code, using an API function to allocate memory on the parallel computing device, (3) using an API function to transfer data to the parallel computing device, (4) developing the parallel part into a kernel function that will be executed by parallel threads, (5) launching a kernel function for execution by parallel threads, and (6) eventually transferring the data back to the host processor with an API function call. We use a running example of vector addition to illustrate these concepts. While the objective of Chapter 2 is to teach enough concepts of the CUDA programming model so that the reader can write a simple parallel CUDA program, it covers several basic skills needed to develop a parallel application based on any parallel programming interface.

Chapter 3 presents more details of the parallel execution model of CUDA, particularly as it relates to handling multi-dimensional data using multi-dimensional organizations of threads. It gives enough insight into the creation, organization, resource binding, and data binding of threads to enable the reader to implement sophisticated computations using CUDA.

Chapter 4 introduces the GPU compute architecture, with a focus on how the computational cores are organized and how threads are scheduled to execute on these cores. Various architecture considerations are discussed with their implications on the performance of code that is executed on the GPU architecture. These include concepts such as transparent scalability, SIMD execution and control divergence, multi-threading and latency tolerance, and occupancy, all of which are defined and discussed in the chapter.

Chapter 5 extends Chapter 4 by discussing the memory architecture of a GPU. It also discusses the special memories that can be used to hold CUDA variables for managing data delivery and improving program execution speed. We introduce the CUDA language features that allocate and use these memories. Appropriate use of these memories can drastically improve the data access throughput and help to alleviate the traffic congestion in the memory system.

Chapter 6 presents several important performance considerations in current CUDA hardware. In particular, it gives more details in desirable patterns of thread execution and memory accesses. These details form the conceptual basis for programmers to reason about the consequence of their decisions on organizing their computation and data. The chapter concludes with a checklist of common optimization strategies that GPU programmers often use to optimize any computation pattern. This checklist will be used throughout the next two parts of the book to optimize various parallel patterns and applications.

Part 2 on primitive parallel patterns consists of Chapter 7 to Chapter 15. Chapter 7 presents convolution, a frequently used parallel computing pattern that is rooted in digital signal processing and computer vision and requires careful management of data access locality. We also use this pattern to introduce constant memory and caching in modern GPUs. Chapter 8 presents stencil, a pattern similar to convolution, but that is rooted in solving differential equations and has specific features that present unique opportunities for further optimization of data access locality. We also use this pattern to introduce three-dimensional organizations of threads and data, and to showcase an optimization introduced in Chapter 6 that targets thread granularity.

Chapter 9 covers histogram, a pattern widely used in statistical data analysis as well as pattern recognition in large data sets. We also use this pattern to introduce atomic operations as a means for coordinating concurrent updates to shared data, and the privatization optimization which reduces the overhead of these operations. Chapter 10 introduces the reduction tree pattern which is used to summarize a collection of input data. We also use this pattern to demonstrate the impact of control divergence and excessive barrier synchronization on performance, and show techniques for how this impact can be mitigated. Chapter 11 presents prefix sum, or scan, an important parallel computing pattern that coverts inherently sequential computation into parallel computation. We also use this pattern to introduce the concept of work-efficiency in parallel algorithms. Chapter 12 presents the filter pattern in two forms: unstable filter (where the original order of filtered elements can be changed) or stable filter (where the original order needs to be retained). The unstable filter pattern is used to demonstrate how threads can work together to reduce the atomic operations they perform, and the stable filter patter demonstrates an important application of the scan pattern previously introduced in Chapter 11. Chapter 13 covers parallel merge, a widely used pattern in divide-and-concur work partitioning strategies. We also use this chapter to introduce dynamic input data identification and organization. Finally, Chapter 14 introduces sorting on GPUs, covering three different parallel sorting algorithms: odd-even sort, merge sort (which relies on the merge pattern in Chapter 13), and radix sort (which relies on the scan pattern covered in Chapter 14). Chapter 15 revisits matrix multiplication in much more depth, applying a large number of advanced optimizations to it from the checklist in Chapter 6.

Part 3 on advanced parallel patterns and applications is similar in spirit to Part 2, but the patterns covered are more elaborate in nature, and often include more application context. As such, these chapters are less focused on introducing new techniques or features and more focused on application-specific considerations. For each application, we start by identifying alternative ways of formulating the basic structure of

the parallel execution and follow up with reasoning about the advantages and disadvantages of each alternative. We then go through the steps of code transformation needed to achieve high performance. These chapters help the readers put all the materials from the previous chapters together and support them as they take on their own application development projects.

Part 3 consists of Chapters 16 to Chapter 23. Chapter 16 uses the classical Floyd-Warshall algorithm from graph processing and Smith-Waterman algorithm for genome sequence alignment as running examples to demonstrate dynamic programming and wavefront parallelism. Chapter 17 presents sparse matrix computation which is widely used for processing very large data sets. This chapter introduces the reader to the concepts of rearranging data for more efficient parallel access: data compression, padding, sorting, transposition, and regularization. Chapter 18 introduces graph algorithms and how graph search can be efficiently implemented in GPU programming. Many different strategies are presented for parallelizing graph algorithms, and the impact of the graph structure on the choice of best algorithm is also discussed. These strategies build on the more primitive patterns such as histogram and merge.

Chapter 19 and Chapter 20 cover deep learning, which has become an extremely important area for GPU computing. Chapter 19 introduces an efficient implementation of convolutional neural networks which leverages techniques such as tiling and patterns like convolution. Chapter 20 introduces large language models and covers state-of-the-art optimizations for training and inferring with these models on GPUs, such as KV caching and batching. Chapter 21 covers the electrostatic potential map computation, and how to leverage techniques scatter-to-gather transformations to enhance parallelism and reduce synchronization overhead.

Chapter 22 introduces computational thinking, the art of formulating and solving computational problems in ways that are more amenable to high-performance computing. It does so by covering the concept of organizing the computation tasks of a program so that they can be done in parallel. We start by discussing the translational process of organizing abstract scientific, problem-specific concepts into computational tasks, which is an important first step in producing quality application software, serial or parallel. It then discusses parallel algorithm structures and their effects on application performance, which is grounded in the performance tuning experience with CUDA. Although we do not go into the implementation details of these alternative parallel programming styles, we expect that the readers will be able to learn to program in any of them with the foundation they gain in this book. The chapter ends by distinguishing between two broad strategies for parallelizing computations, input-centric and output-centric, and circles back to the patterns and applications covered in the book to study the strategies used for each retrospectively.

The last chapter in Part 3, Chapter 23 covers CUDA programming on heterogeneous clusters, where each compute node consists of both CPUs and GPUs. We discuss the use of three different programming models – MPI, NCCL, and NVSHMEM – alongside CUDA to orchestrate the distribution of a computation across many GPUs in a cluster, while mitigating the communication overhead.

While the chapters throughout this book are based on CUDA, they help the readers build up the foundation for parallel programming in general. We believe that

humans understand best when we learn from concrete examples. That is, we must first learn the concepts in the context of a particular programming model, which provides us with solid footing when we generalize our knowledge to other programming models. As we do so, we can draw on our concrete experience from the CUDA examples. An in-depth experience with CUDA also enables us to gain maturity, which will help us learn concepts that may not even be pertinent to the CUDA model.

Chapter 24 offers concluding remarks and an outlook for the future of massively parallel programming. We first revisit our goals and summarize how the chapters fit together to help achieve the goals. We then conclude with a prediction that these fast advances in massively parallel computing will make it one of the most exciting areas in the coming decade.

References

[1] J. Von Neumann, First draft of a report on the edvac, IEEE Annals of the History of Computing 15 (4) (1993) 27–75.
[2] H. Sutter, J. Larus, Software and the concurrency revolution: leveraging the full power of multicore processors demands new tools and new thinking from the software industry, ACM Queue 3 (7) (2005) 54–62.
[3] W.-m. Hwu, K. Keutzer, T.G. Mattson, The concurrency challenge, IEEE Design & Test of Computers 25 (4) (2008) 312–320.
[4] NVIDIA, CUDA C++ programming guide. Release 13.0, https://docs.nvidia.com/cuda/cuda-c-programming-guide, 2025.
[5] G.M. Amdahl, Computer architecture and Amdahl's law, Computer 46 (12) (2013) 38–46.
[6] T.G. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Pearson Education, 2004.
[7] OpenMP.org, OpenMP: the OpenMP API specification for parallel programming, https://www.openmp.org, 2025.
[8] M.P. Forum, Mpi: a message-passing interface standard, 1994.
[9] K.O.W. Group, The opencl specification, version 1.0, 2008.
[10] J.M. Wing, Computational thinking, Communications of the ACM 49 (3) (2006) 33–35.

PART

Fundamental concepts

1

This page intentionally left blank

CHAPTER

Heterogeneous data-parallel computing

2

Data parallelism refers to the phenomenon that the computations to be performed on different parts of a dataset can be done independently of each other and thus can be done in parallel with each other. Many applications exhibit a rich amount of data parallelism that makes them amenable to scalable parallel execution where the speed of execution improves proportionally to the amount of execution resources in the system. It is therefore important for parallel programmers to be familiar with the concept of data parallelism and the parallel programming language constructs for writing code that exploit data parallelism. In this chapter, we will use the CUDA C++ language constructs to develop a simple data parallel program.

2.1 Data parallelism

When modern software applications run slowly, the problem is usually data – too much data to process. Image processing applications manipulate images or videos with millions to trillions of pixels. Scientific applications model fluid dynamics using billions of grid points. Molecular dynamics applications simulate interactions between thousands to billions of atoms. Airline scheduling must deal with thousands of flights, crews, and airport gates. Most of these pixels, particles, grid points, interactions, flights, and so on can usually be dealt with largely independently. For example, in image processing, converting a color pixel to grayscale requires only the data of that pixel. Blurring an image averages each pixel's color with the colors of nearby pixels, requiring only the data of that small neighborhood of pixels. Even a seemingly global operation, such as finding the average brightness of all pixels in an image, can be broken down into many smaller computations that can be executed independently. Such independent evaluation of different pieces of data is the basis of data parallelism. Writing data parallel code entails (re)organizing the computation around the data such that one can execute the resulting independent computations in parallel to complete the overall job faster – often much faster.

Let us illustrate the concept of data parallelism with a color-to-grayscale conversion example. Fig. 2.1 shows a color image (left side) consisting of many pixels, each

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00011-1
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.



FIGURE 2.1

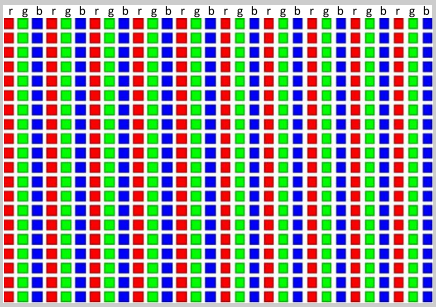
Conversion of a color image to a grayscale image.

containing a red, green, and blue fractional value (r, g, b) varying from 0 (black) to 1 (full intensity). To convert the color image (left side of Fig. 2.1) to a grayscale image (right side), we compute the luminance value L for each pixel by applying the following weighted sum formula:

$$L = r * 0.299 + g * 0.587 + b * 0.114 \tag{2.1}$$

RGB Color Image Representation

In an RGB representation, each pixel of an image is specified as a tuple of (r, g, b) values. The format of an image's row is (r g b) (r g b) ... (r g b), as illustrated in the following conceptual picture.



Each tuple specifies a mixture of red (R), green (G), and blue (B). That is, for each pixel, the r, g, and b values represent the intensity (0 being dark and 1 being full intensity) of the red, green, and blue light sources when the pixel is rendered.

If we consider the input to be an image organized as an array `I` of RGB values and the output to be a corresponding array `O` of luminance values, we get the simple

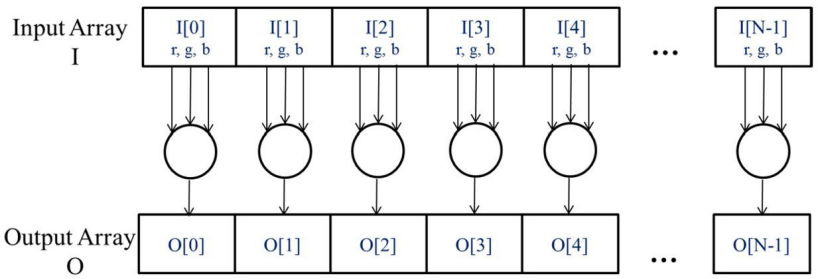


FIGURE 2.2

Data parallelism in color-image-to-grayscale conversion – pixels can be calculated independently of each other.

computation structure shown in Fig. 2.2. For example, `O[0]` is generated by calculating the weighted sum the RGB values in `I[0]` according to the formula above; `O[1]` by calculating the weighted sum of the RGB values in `I[1]`; `O[2]` by calculating the weighted sum of the RGB values in `I[2]`; and so on. None of these per-pixel computations depends on each other. All of them can be performed independently. Clearly, color-to-grayscale conversion exhibits a rich amount of data parallelism as modern images commonly consist of millions of pixels. Of course, data parallelism in complete applications can be more complex and much of this book is devoted to teaching the "parallel thinking" necessary to find and exploit data parallelism.

Data Parallelism vs. Task Parallelism

Data parallelism is not the only type of parallelism used in parallel programming. Task Parallelism has also been used extensively in parallel programming. Task parallelism is typically exposed through task decomposition of applications. For example, a simple application may need to do a vector addition and a matrix-vector multiplication. Each of these would be a task. Task parallelism exists if the two tasks can be done independently. I/O and data transfers are also common sources of tasks.

In large applications, there is usually a larger number of independent tasks and therefore a larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks include vibrational force calculation, rotational force calculation, neighbor identification for non-bonding forces, non-bonding force calculation, velocity and position calculation, etc.

In general, data parallelism is the main source of performance improvement for parallel programs. With large data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources, a desirable property referred to as *scalability*. Nevertheless, task parallelism can also play an important role in achieving performance goals.

2.2 CUDA C++ program structure

Our next step is to learn how to write a program to exploit data parallelism for faster execution in heterogeneous computing systems. Without loss of generality, we will use code examples of CUDA C++ programs to illustrate the algorithms and techniques in this book. From our experience, concepts learned based on CUDA C++ can be easily extended to other programming languages and platforms.

CUDA C++[1] extends the popular C++ programming language with minimal new syntax and with library functions to let programmers target heterogeneous computing systems containing both CPUs and GPUs. As the name implies, CUDA C++ is built on NVIDIA's CUDA platform. CUDA is currently the most mature framework for massively parallel computing. It is broadly used in the high-performance computing industry, with essential tools such as compilers, debuggers, and profilers available on the most common operating systems.

The structure of a CUDA C++ program reflects the co-existence of a host (CPU) and one or more devices (GPUs) in the computer. Each CUDA C++ source file can have a mixture of both *host code* and *device code*. By default, any traditional C++ program is a CUDA C++ program that contains only host code. One can add device code into any source file. The device code is clearly marked with special CUDA C++ keywords. The device code includes functions, or kernels, whose code is executed in a data-parallel manner.

The execution of a CUDA C++ program is illustrated in Fig. 2.3. The execution starts with host code (CPU serial code) whose execution is illustrated as a single curly arrow that represents a thread. When a kernel, which is a piece of device code (device parallel kernel in Fig. 2.3), is called, a large number of threads are launched on a device to execute the kernel. All the threads that are launched by a kernel call are collectively called a *grid*. These threads are the primary vehicle of parallel execution in a CUDA platform. Fig. 2.3 shows the execution of two grids of threads. Each grid is shown as multiple *blocks* of threads. We will discuss how these grids and blocks are organized soon. When all threads of a grid complete their execution, the grid terminates and the execution continues on the host until another grid is launched. As is implied in Fig. 2.3, heterogeneous computing applications can manage overlapped CPU and GPU execution to take advantage of both CPUs and GPUs.

Threads

A thread is a simplified view of how a processor executes a sequential program in modern computers. A thread consists of the code of the program, the point in the code that is being executed, and the values of its variables and data structures. The execution of a thread is sequential as far as a user is concerned. One

[1] In the code examples in this book, we use C-style programming for simplicity and introduce C++ features where needed. However, readers should be aware that CUDA programming has evolved towards C++ to support modern generic programming and other object-oriented programming practices.

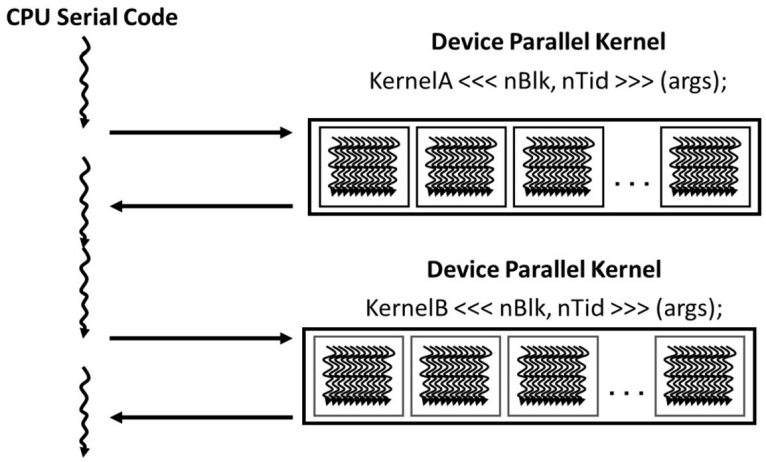


FIGURE 2.3

Execution of a CUDA program.

can use a source-level debugger to monitor the progress of a thread by executing one statement at a time, looking at the statement that will be executed next and checking the values of the variables and data structures as the execution progresses. Threads have been used in programming for many years. If a programmer wants to start parallel execution in an application, the programmer creates and manages multiple threads using thread libraries or special languages. In CUDA, the execution of each device thread is sequential as well. A CUDA program initiates parallel execution by calling kernel functions, which causes the underlying runtime mechanisms to launch a grid of device threads that process different parts of the data in parallel.

Launching a grid typically generates many threads to exploit data parallelism. In the color-to-grayscale conversion example, each thread could be used to compute one pixel of the output array. In this case, the number of threads that ought to be generated by the grid launch is equal to the number of pixels in the image. For large images, a large number of threads, i.e., millions, could be generated. In practice, one might use each thread to compute multiple output pixels to enhance execution and data access efficiency, as we will discuss later in this book. CUDA programmers can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support. This assumption contrasts with traditional CPU threads that typically take thousands of clock cycles to generate and schedule. In Chapter 3, we show how to implement color-to-grayscale conversion and image blur kernels. In this chapter, we will use vector addition as a running example for simplicity.

```
01    // Compute vector sum C_h = A_h + B_h
02    void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
03        for (int i = 0; i < n; ++i) {
04            C_h[i] = A_h[i] + B_h[i];
05        }
06    }
07    int main() {
08        // Memory allocation for arrays A, B, and C
09        // I/O to read A and B, N elements each
10        ...
11        vecAdd(A, B, C, N);
12    }
```

FIGURE 2.4

A simple traditional vector addition C++ code example.

2.3 A vector addition example

Vector addition is arguably the simplest possible data parallel computation – the parallel equivalent of "Hello World" from sequential programming. Before we show the kernel code for vector addition, it is helpful to first review how a conventional vector addition (host code) function works. Fig. 2.4 shows a simple C++ program that consists of a main function and a vector addition function. In all our examples, whenever there is a need to distinguish between host and device data, we will suffix the names of variables that are used by the host with "_h" and those of variables that are used by a device with "_d" to remind ourselves of the intended usage of these variables. Since we only have host code in Fig. 2.4, we only see variables suffixed with "_h".

Assume that the vectors to be added are stored in arrays `A` and `B` that are allocated and initialized in the main function. The output vector is in array `C`, which is also allocated in the main function. For brevity, we do not show the details of how `A`, `B`, and `C` are allocated or initialized in the main function. The pointers to these arrays are passed to the `vecAdd` function, along with the variable `N` that contains the length of the vectors. Note that the parameters of the `vecAdd` function are suffixed with "_h" to emphasize that they are used by the host.

Pointers in the C++ Language

The function arguments `A`, `B`, and `C` in Fig. 2.4 are pointers. In the C and C++ languages, a pointer can be used to access variables and data structures. While a floating-point variable `V` can be declared with:

```
float V;
```

a *pointer* variable `P` can be declared with:

```
float *P;
```

By assigning the address of `V` to `P` with the statement $P = \&V$, we make `P` "point to" `V`. `*P` becomes a synonym for `V`. For example, `U = *P` assigns the value of `V` to `U`. For another example, `*P = 3` changes the value of `V` to 3.

An array in a C or C++ program can be accessed through a pointer that points to its 0th element. For example, the statement `P = &(A[0])` makes `P` point to the 0th

```
01   void vecAdd(float* A, float* B, float* C, int n) {
02       int  size = n* sizeof(float);
03       float  *d_A *d_B, *d_C;
04
05       // Part 1: Allocate device memory for A, B, and C
06       // Copy A and B to device memory
07       ...
08
09       // Part 2: Call kernel – to launch a grid of threads
10       // to perform the actual vector addition
11       ...
12
13       // Part 3: Copy C from the device memory
14       // Free device vectors
15       ...
16   }
```

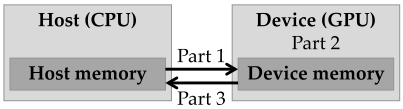


FIGURE 2.5

Outline of a revised vecAdd (host code) function that moves the work to a device.

element of array `A`. `P[i]` becomes a synonym for `A[i]`. In fact, the array name `A` is in itself a pointer to its 0th element.
In Fig. 2.4, passing an array name `A` as the first argument in the function call to `vecAdd` makes the function's first parameter `A_h` point to the 0th element of `A`. As a result, `A_h[i]` in the function body can be used to access `A[i]` for the array `A` defined in the main function. See [1] for an easy-to-follow explanation of the detailed usage of pointers in C.

The `vecAdd` function in Fig. 2.4 uses a for-loop to iterate through the vector elements. In the ith iteration, output element `C_h[i]` receives the sum of `A_h[i]` and `B_h[i]`. The vector length parameter `n` is used to control the loop so that the number of iterations matches the length of the vectors. The function reads the elements of `A` and `B` and writes the elements of `C` through the pointers `A_h`, `B_h`, and `C_h`, respectively. When the `vecAdd` function returns, the subsequent statements in the main function can access the new contents of `C`.

A straightforward way to execute vector addition in parallel is to modify the `vecAdd` function and move its calculations to a device. The structure of such a modified `vecAdd` function is shown in Fig. 2.5. Part 1 of the function allocates space in the device (GPU) memory to hold copies of the `A`, `B`, and `C` vectors, and copies the `A` and `B` vectors from the host memory to the device memory. Part 2 calls a vector addition kernel to launch a grid of threads on the device. Part 3 copies the sum vector `C` from the device memory to the host memory and frees the three arrays from the device memory.

Note that the revised `vecAdd` function is essentially an outsourcing agent that ships input data to a device, activates the calculation on the device, and collects the results from the device. The agent does so in such a way that the main program does not need to even be aware that the vector addition is now actually done on a device. In practice, such "transparent" outsourcing model can be very inefficient because of all the copying of data back and forth. One would often keep large and important data structures on the device and simply invoke device functions that operate on the data structures from the host code. For now, however, we will use the simplified transparent model to introduce the basic CUDA C++ program structure. The details of the revised function, as well as the way to compose the kernel function, will be the topic of the rest of this chapter.

2.4 Device global memory and data transfer

In current CUDA systems, devices are often hardware cards that come with their own Dynamic Random-Access Memory (DRAM) called *device global memory*, or simply *global memory*. For example, the NVIDIA Hopper H100 GPU comes with 80 GB or 94 GB of global memory. Calling it "global" memory distinguishes it from other types of device memory that are also accessible to programmers. Details about the CUDA memory model and the different types of device memory are discussed in 5.

For the vector addition kernel, before calling the kernel, the programmer needs to allocate space in the device global memory and transfer data from the host memory to the allocated space in the device global memory. These activities correspond to Part 1 of Fig. 2.5. Similarly, after device execution, the programmer needs to transfer the output data from the device global memory back to the host memory and free up the allocated space in the device global memory that is no longer needed. These activities correspond to Part 3 of Fig. 2.5. The CUDA runtime system (typically running on the host) provides API functions to perform these activities on behalf of the programmer. From this point on, we will simply say that data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device global memory. The same holds for the opposite direction.

In Fig. 2.5, Part 1 and Part 3 of the `vecAdd` function need to use the CUDA API functions to allocate device global memory for `A`, `B`, and `C`, transfer `A` and `B` from host to device, transfer `C` from device to host after the vector addition, and free the device global memory for `A`, `B`, and `C`. We will explain the memory allocation and free functions first.

Fig. 2.6 shows two API functions for allocating and freeing device global memory. The `cudaMalloc` function can be called from the host code to allocate a piece of device global memory for an object. The reader should notice the striking similarity between `cudaMalloc` and the `malloc` function from the standard C runtime library, which is also part of C++. This similarity is intentional; CUDA C++ is C++ with minimal extensions. CUDA C++ uses the standard C and C++ runtime library `malloc` function to manage the host memory and adds `cudaMalloc` as an extension to the C++

cudaMalloc(): allocates object in the device global memory and requires two parameters
- **Address of a pointer** to the allocated object
- **Size** of allocated object in terms of bytes

cudaFree(): frees object from device global memory and requires one parameter
- **Pointer** to freed object

FIGURE 2.6

CUDA API functions for managing device global memory.

runtime library. By keeping the interface as close to the original C and C++ runtime libraries as possible, CUDA C++ minimizes the time that a C or C++ programmer spends to relearn the use of these extensions.

The first parameter to the `cudaMalloc` function is the address of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to `(void**)` because the function expects a generic pointer; the memory allocation function is a generic function that is not restricted to any particular type of objects.[2] This parameter allows the `cudaMalloc` function to write the address of the allocated memory into the provided pointer variable regardless of its type. The host code that calls kernels passes this pointer value to the kernels that need to access the allocated memory object. The second parameter to the `cudaMalloc` function gives the size of the data to be allocated, in number of bytes. The usage of this second parameter is consistent with the size parameter to the C `malloc` function.

Note that `cudaMalloc` has a slightly different format from the C `malloc` function. The C `malloc` function returns a pointer to the allocated object. It takes only one parameter that specifies the size of the allocated object. The `cudaMalloc` function writes to the pointer variable whose address is given as the first parameter. As a result, the `cudaMalloc` function takes two parameters. The two-parameter format of `cudaMalloc` allows it to use the return value to report any errors in the same way as other CUDA API functions.

The `cudaFree` function strongly resembles the C `free` function. It takes one input parameter, the pointer to the memory to be freed, and frees the allocated space from the device memory.

CUDA C++ also has more advanced library functions for allocating and deallocating space in the host and device memories. These alternative methods allow for faster copying of data, or eliminate the need to copy the data explicitly and allow it to be copied automatically on-demand. Such methods provide programs with improved performance or convenience when managing the device memory. We discuss these other forms of memory allocation in Chapter 23 and in Appendix C.

We now use the following simple code example to illustrate the use of `cudaMalloc` and `cudaFree`:

[2] The fact that `cudaMalloc` returns a generic object makes the use of dynamically allocated multidimensional arrays more complex. We address this issue in Chapter 3.

```
float *A_d
int size = n*sizeof(float);
cudaMalloc((void**)&A_d, size);
...
cudaFree(A_d);
```

This code is a continuation of the example in Fig. 2.5. For clarity, we suffix a pointer variable with "_d" to indicate that it points to an object in the device global memory. The first argument passed to `cudaMalloc` is the address of pointer `A_d` (i.e., `&A_d`) cast to a void pointer. When `cudaMalloc` returns, `A_d` will point to the device global memory region allocated for the `A` vector. The second argument passed to `cudaMalloc` is the size of the region to be allocated. Since size is in number of bytes, the programmer needs to translate from the number of elements in an array to the number of bytes when determining the value of size. For example, when allocating space for an array of `n` single-precision floating-point elements, the value of size would be `n` times the size of a single-precision floating number, which is 4 bytes in computers today. Therefore, the value of size would be `n*4`. After the computation, `cudaFree` is called with pointer `A_d` passed as an argument to free the storage space for the `A` vector from the device global memory. Note that `cudaFree` does not need to change the value of `A_d`; it only needs to use the value of `A_d` to return the allocated memory back to the available pool. Thus, only the value and not the address of `A_d` is passed as an argument.

The addresses in `A_d`, `B_d`, and `C_d` point to locations in the device global memory. These addresses should not be dereferenced in the host code. They should only be used in calling API functions and kernel functions. Dereferencing a device global memory pointer in host code can cause exceptions or other types of run-time errors since these addresses are located in the device global memory and are not accessible by the user host code.

Now that we have discussed the declaration of the `A_d` pointer variable, and its use in `cudaMalloc` and `cudaFree` calls, the reader should be able to complete Part 1 of the `vecAdd` example in Fig. 2.5 with similar declarations of the `B_d` and `C_d` pointer variables as well as their corresponding `cudaMalloc` calls. Furthermore, Part 3 in Fig. 2.5 can be completed with the `cudaFree` calls for `B_d` and `C_d`.

Once the host code has allocated space in the device global memory for the data objects, it can request that data be transferred from host to device and vice versa. This transfer is accomplished by calling one of the CUDA API functions. Fig. 2.7 shows such an API function, `cudaMemcpy`. The `cudaMemcpy` function takes four parameters. The first parameter is a pointer to the destination location for the data object to be copied. The second parameter points to the source location. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the location of the memory involved in the copy: from host to host, from host to device, from device to host, and from device to device. For example, the memory copy function can be used to copy data from one location in the device global memory to another location of the device global memory.

The `vecAdd` function calls the `cudaMemcpy` function to copy the host memory arrays pointed to by `A_h` and `B_h` to the device memory arrays pointed to by `A_d` and `B_d`

cudaMemcpy(): performs memory data transfer and requires four parameters
- o **Pointer** to destination
- o **Pointer** to source
- o **Size** in number of bytes copied
- o **Type/Direction** of transfer

FIGURE 2.7

CUDA API function for data transfer between host and device.

```
01    void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
02        int size = n * sizeof(float);
03        float *A_d, *B_d, *C_d;
04
05        cudaMalloc((void **) &A_d, size);
06        cudaMalloc((void **) &B_d, size);
07        cudaMalloc((void **) &C_d, size);
08
09        cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
10        cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
11
12        // Kernel invocation code - to be shown later
13        ...
14
15        cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);
16
17        cudaFree(A_d);
18        cudaFree(B_d);
19        cudaFree(C_d);
20    }
```

FIGURE 2.8

A more complete version of the `vecAdd` function.

before adding them. The `vecAdd` function also calls the `cudaMemcpy` function to copy the device memory array pointed to by `C_d` to the host memory array pointed to by `C_h` after the vector addition is done. Assume that the values of `A_h`, `B_h`, `A_d`, `B_d`, and size have already been set as we discussed before, the three `cudaMemcpy` calls are shown below.

```
cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
...
cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);
```

The two symbolic constants, `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`, are recognized, predefined constants of the CUDA programming environment. Note that the same `cudaMemcpy` function can be used to transfer data in both directions by properly ordering the source and destination pointers and using the appropriate constant for the transfer type.

To summarize, the main function in Fig. 2.4 calls `vecAdd`, which is also executed on the host. The `vecAdd` function, outlined in Fig. 2.5, allocates space in device global memory, requests data transfers, and calls the kernel that performs the actual vector addition. We refer to this type of host code as a *stub* for calling a kernel. We show a more complete version of the `vecAdd` function in Fig. 2.8.

Compared to Fig. 2.5, the `vecAdd` function in Fig. 2.8 is complete for Part 1 and Part 3. Part 1 allocates device global memory for `A_d`, `B_d`, and `C_d` and transfers `A_h` to `A_d` and `B_h` to `B_d`. This is done by calling the `cudaMalloc` and `cudaMemcpy` functions. The readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Fig. 2.8. Part 2 calls the kernel and will be described in the following subsection. Part 3 copies the vector sum data from device to host so that the values will be available in the main function. This is accomplished with a call to the `cudaMemcpy` function. It then frees the memory for `A_d`, `B_d`, and `C_d` from the device global memory, which is done by calls to the `cudaFree` function.

Error Checking and Handling in CUDA C++ Programs

It is important for a program to check for and handle errors. With good error checking and handling, one can catch the errors when they happen and detect them close to the root cause of the problems. CUDA API functions return flags that indicate whether an error has occurred when they served the request. Most errors are due to inappropriate argument values used in the call. For brevity, we will not show error checking code in our examples. For example, Fig. 2.8 shows a call to `cudaMalloc` as follows:

```
cudaMalloc((void **) &A_d, size);
```

In practice, CUDA programmers surround the call with code that checks the returned error condition and prints out a relevant error message so that the user can be aware of the fact that an error has occurred.
A simple version of such checking code is as follows:

```
cudaError_t err = cudaMalloc((void **) &A_d, size);
if (error != cudaSuccess) {
    printf("%s in %s at line %d \n", cudaGetErrorString(err),
        __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```

This way, if the system is out of device memory, the user will be informed about the situation. Proper error checking can save many hours of debugging time. One could define a C++ macro to make the checking code more concise in the source code.

2.5 Kernel functions and threading

We are now ready to discuss more about the CUDA C++ kernel functions and the effect of calling these kernel functions. In CUDA, a kernel function specifies the

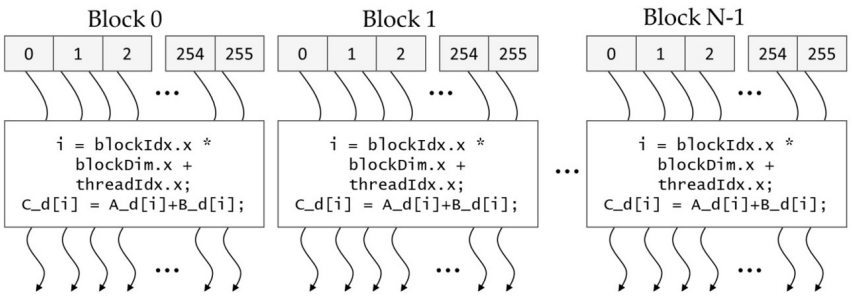


FIGURE 2.9

All threads in a grid execute the same kernel code.

code to be executed by GPU threads during a parallel phase. Since all these threads execute the same code, CUDA C++ programming is an instance of the well-known Single-Program Multiple-Data (SPMD) [2] parallel programming style, a popular programming style for parallel computing systems.

When a program's host code calls a kernel, the CUDA runtime system running on the host launches a grid of GPU threads that are organized into a two-level hierarchy. Each grid is organized as an array of thread blocks, which will be referred to as blocks for brevity. All blocks of a grid have the same size. Each block can contain up to 1024 threads on current systems. Fig. 2.9 shows an example where each block consists of 256 threads. Each thread is represented by a curly arrow stemming from a box that is labeled with the thread's index number in the block.

The total number of blocks in the grid and the total number of threads in each block is specified by the host code when a kernel is called. The same kernel can be called with different numbers of blocks and threads in different parts of the host code. For a given grid of threads, the number of threads in a block is available in a *built-in variable* named `blockDim`. The `blockDim` variable's type is a C++ `struct` with three unsigned integer fields (`x`, `y`, and `z`) which help the programmer organize the threads into a one-, two-, or three-dimensional array. For a one-dimensional organization, only the `x` field is used. For a two-dimensional organization, the `x` and `y` fields are used. For a three-dimensional structure, all three `x`, `y`, and `z` fields are used. The choice of dimensionality for organizing threads usually reflects the dimensionality of the data. This makes sense since the threads are created to process data in parallel, so it is only natural that the organization of the threads reflects the organization of the data.

Built-in Variables

Many programming languages have built-in variables. These variables have special meaning and purpose. The values of these variables are often pre-initialized by the runtime system and are typically read-only in the program. The programmers should refrain from redefining these variables for any other purposes.

In Fig. 2.9, each thread block is organized as a one-dimension array of threads because the data are one-dimensional vectors. The value of the `blockDim.x` variable indicates the total number of threads in each block, which is 256 in Fig. 2.9. In general, it is recommended that the total number of threads in a thread block is a multiple of 32 for hardware efficiency reasons. We will revisit this point later.

CUDA kernels have access to two more built-in variables, `threadIdx` and `blockIdx`, that allow threads to distinguish themselves from each other and to determine the area of data each thread is to work on. The `threadIdx` variable gives each thread a unique coordinate within a block. In Fig. 2.9, since we are using a one-dimensional thread organization, only `threadIdx.x` is used. The `threadIdx.x` value for each thread is shown in the small shaded box of each thread in Fig. 2.9. The first thread in each block has value 0 in its `threadIdx.x` variable, the second thread has value 1, the third thread has value 2, etc.

The `blockIdx` variable gives all threads in a block a common block coordinate. In Fig. 2.9, all threads in the first block have value 0 in their `blockIdx.x` variables, those in the second thread block value 1, and so on. Using an analogy with the telephone system, one can think of `threadIdx.x` as local phone number and `blockIdx.x` as area code (see Sidebar on *Hierarchical Organizations*). The two together gives each telephone line a unique phone number in the whole country. Similarly, each thread can combine its `threadIdx` and `blockIdx` values to create a unique global index for itself within the entire grid.

In Fig. 2.9, a unique global index `i` is calculated with the C++ statement `i = blockIdx.x * blockDim.x + threadIdx.x`. Recall that `blockDim.x` is 256 in our example. The `i` values of threads in block 0 range from 0 to 255. The `i` values of threads in block 1 range from 256 to 511. The `i` values of threads in block 2 range from 512 to 767. That is, the `i` values of the threads in these three blocks form a continuous coverage of the values from 0 to 767. Since each thread uses `i` to access `A`, `B`, and `C`, these threads cover the first 768 iterations of the original loop. By launching a grid with a larger number of blocks, one can process larger vectors. By launching a grid with `n` or more threads, one can process vectors of length `n`.

Hierarchical Organizations

Like CUDA threads, many real-world systems are organized hierarchically. The U.S. telephone system is a good example. At the top level, the telephone system consists of "areas" each of which corresponds to a geographical area. All telephone lines within the same area have the same 3-digit "area code". A telephone area is sometimes larger than a city. For example, many counties and cities of central Illinois are within the same telephone area and share the same area code 217. Within an area, each phone line has a seven-digit local phone number, which allows each area to have a maximum of about ten million numbers. One can think of each phone line as a CUDA thread, with the area code as the value of `blockIdx` and the seven-digital local number as the value of `threadIdx`. This hierarchical organization allows the system to have a very large number of phone

```
01    // Compute vector sum C = A + B
02    // Each thread performs one pair-wise addition
03    __global__
04    void vecAddKernel(float* A, float* B, float* C, int n) {
05        int i = threadIdx.x + blockDim.x * blockIdx.x;
06        if (i < n) {
07            C[i] = A[i] + B[i];
08        }
09    }
```

FIGURE 2.10

A simple vector addition kernel function.

lines while preserving "locality" for calling the same area. That is, when dialing a phone line in the same area, a caller only needs to dial the local number. As long as we make most of our calls within the local area, we seldom need to dial the area code. If we occasionally need to call a phone line in another area, we dial 1 and the area code, followed by the local number (this is the reason why no local number in any area should start with a 1). The hierarchical organization of CUDA threads also offers a form of locality. We will study this locality soon.

Fig. 2.10 shows a kernel function for vector addition. Note that we do not use the "`_h`" and "`_d`" convention in the kernel code because there is no potential for confusion. We do not access any host memory from the kernel code in our examples.[3] The syntax of a kernel follows the C++ standard with some notable extensions. First, there is a CUDA-specific keyword `__global__` in front of the declaration of the `vecAddKernel` function. This keyword indicates that the function is a kernel and that it can be called to generate a grid of threads on a device.

In general, CUDA C++ extends the C++ language with three *qualifier keywords* that can be used in function declarations. The meaning of these keywords is summarized in Fig. 2.11. The `__global__` keyword indicates that the function being declared is a CUDA C++ kernel function. Note that there are two underscore characters on each side of the word "`global`". Such a kernel function is executed on the device and can be called from the host. In CUDA systems that support dynamic parallelism (starting with the Kepler architecture), it can also be called from the device. The important feature is that calling such a kernel function results in a new grid of threads being launched on the device, i.e., the GPU.

[3] In general, it is possible for the kernel code to access host memory through a mechanism called *Unified Virtual Address* (UVA). This mechanism allows the GPU to access host memory but at lower bandwidth than accessing its own device global memory, and is discussed in Appendix C.

Qualifier Keyword	Callable From	Executed On	Executed By
__host__ (default)	Host	Host	Caller host thread
__global__	Host (or Device)	Device	New grid of device threads
__device__	Device	Device	Caller device thread

FIGURE 2.11

CUDA C++ keywords for function declaration.

The `__device__` keyword in Fig. 2.11 indicates that the function being declared is a CUDA device function. A device function executes on a GPU device and can only be called from a kernel function or another device function. The device function is executed by the device thread that calls it and does not result in any new device threads being launched.

The `__host__` keyword indicates that the function being declared is a CUDA host function. A host function is simply a traditional C++ function that executes on host and can only be called from another host function. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration. This makes sense since many CUDA applications are ported from CPU-only execution environments. The programmer would add kernel functions and device functions during the porting process. The original functions remain as host functions. Having all functions to default into host functions spares the programmer the tedious work to change all original function declarations.

Since all functions are host functions by default, the reader may wonder why the `__host__` keyword is needed. In some cases, a programmer may want a function to be available on both the host and the device. Labeling the function with the `__device__` keyword makes the function available only on the device. To make it available on both, one can use both the `__host__` and `__device__` keywords together in a function declaration. This combination tells the compilation system to generate two versions of object code for the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function. This capability supports a common use case when the same function source code can be recompiled to generate a device version. Many user library functions will likely fall into this category.

The second notable CUDA C++ extension used the `vecAdd` kernel function in Fig. 2.10 is the built-in variables `threadIdx`, `blockIdx`, and `blockDim`. Recall that all threads execute the same kernel code and there needs to be a way for them to distinguish themselves from each other and direct each thread towards a particular part of the data. These built-in variables are the means for threads to access hardware registers that provide the identifying coordinates to threads. Different threads will see

different values in their threadIdx.x, blockIdx.x, and blockDim.x variables. For readability, we will sometimes refer to a thread as $thread_{\texttt{blockIdx.x, threadIdx.x}}$ in our discussions.

The global index computed using the built in variables (line 05 in Fig. 2.10) is written to an automatic (local) variable i of type int.[4] In a CUDA kernel function, automatic variables are private to each thread. That is, a version of i will be generated for every thread. If the grid is launched with 10,000 threads, there will be 10,000 versions of i, one for each thread. The value assigned by a thread to its i variable is not visible to other threads. We will discuss these automatic variables in more details in Chapter 5.

A quick comparison between Fig. 2.4 and Fig. 2.10 reveals an important insight for CUDA kernels. The kernel function in Fig. 2.10 does not have a loop that corresponds to the one in Fig. 2.10. The reader should ask where the loop went. The answer is that the loop is now replaced with the grid of threads. The entire grid forms the equivalent of the loop. Each thread in the grid corresponds to one iteration of the original loop. This is sometimes referred to as *loop parallelism*, where iterations of the original sequential code are executed by threads in parallel.

Note that there is an if (i<n) statement in vecAddVecKernel in Fig. 2.10. This is because not all vector lengths can be expressed as multiples of the block size. For example, let's assume that the vector length is 100. The smallest efficient thread block dimension is 32. Assume that we picked 32 as the block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their i values against n, which is 100. With the if (i<n) statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to be called to process vectors of arbitrary lengths.

2.6 Calling kernel functions

Having implemented the kernel function, the remaining step is to call that function from the host code to launch the grid. Fig. 2.12 shows the host code for calling the vector addition kernel. When the host code calls a kernel function, it sets the grid and thread block dimensions via execution configuration parameters. The configuration

[4] Different C and C++ programmers may choose to use the int or the unsigned int type when declaring variables intended to represent array indices, array sizes, loop iteration counters, and loop bounds. The use of unsigned int makes the intent clear that these variables should never be negative and may result in beneficial compiler warnings that prevent bugs. On the other hand, the use of int avoids bugs when performing subtraction or intermixing unsigned int variables with other int variables when performing arithmetic. Moreover, since the overflow of int variables is undefined, the compiler can take advantage of that fact when performing optimizations. Throughout this book, we sometimes used int and sometimes use unsigned int when declaring array indices, array sizes, loop iteration counters, and loop bounds. Readers can use either style in their own implementations.

```
01    int vectAdd(float* A, float* B, float* C, int n) {
02        // A_d, B_d, C_d allocations and copies omitted
03        ...
04        // Launch ceil(n/256) blocks of 256 threads each
05        vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
06    }
```

FIGURE 2.12

A host code example with a vector addition kernel call statement.

```
01    void vecAdd(float* A, float* B, float* C, int n) {
02        float *A_d, *B_d, *C_d;
03        int size = n * sizeof(float);
04
05        cudaMalloc((void **) &A_d, size);
06        cudaMalloc((void **) &B_d, size);
07        cudaMalloc((void **) &C_d, size);
08
09        cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
10        cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
11
12        vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
13
14        cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
15
16        cudaFree(A_d);
17        cudaFree(B_d);
18        cudaFree(C_d);
19    }
```

FIGURE 2.13

A vector addition kernel call statement.

parameters are given between the "<<<" and ">>>" brackets before the traditional C++ function arguments. The first configuration parameter gives the number of blocks in the grid. The second specifies the number of threads in each block. In this example, there are 256 threads in each block.

To ensure that we have enough threads in the grid to cover all the vector elements, we need to set the number of blocks in the grid to the ceiling division (rounding up the quotient to the immediate higher integer value) of the desired number of threads (`n` in this case) by the thread block size (256 in this case). There are many ways to perform a ceiling division. One way is to apply the C++ ceiling function to `n/256.0`. Using floating point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we want 1000 threads, we would launch `ceil(1000/256.0)` = 4 thread blocks. As a result, the statement will launch 4*256 = 1024 threads. With the `if (i < n)` statement in the kernel as shown in Fig. 2.10, the first 1000 threads will perform addition on the 1000 vector elements. The remaining 24 will not.

Fig. 2.13 shows the final host code in the `vecAdd` function. This source code completes the skeleton code in Fig. 2.5. Fig. 2.10 and Fig. 2.13 jointly illustrate a simple

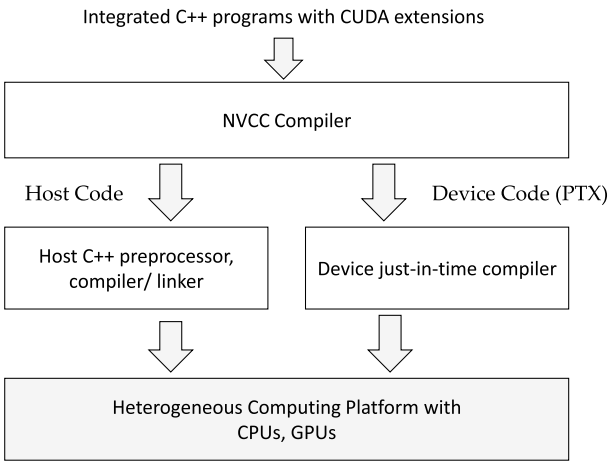


FIGURE 2.14

Overview of the compilation process of a CUDA C++ program.

CUDA program that consists of both host code and a device kernel. The code is hard-wired to use thread blocks of 256 threads each. The number of thread blocks used, however, depends on the length of the vectors (n). If n is 750, then three thread blocks will be used. If n is 4000, then 16 thread blocks will be used. If n is 2,000,000, then 7813 blocks will be used. Note that all the thread blocks operate on different parts of the vectors. They can be executed in any arbitrary order. Programmers must not make any assumptions regarding execution order. A small GPU with a small amount of execution resources may execute only one or two of these thread blocks in parallel. A larger GPU may execute 128 or 256 blocks in parallel. This gives CUDA kernels scalability in execution speed with hardware. That is, the same code runs at lower speed on small GPUs and higher speed on larger GPUs. We revisit this point in Chapter 4.

2.7 Compilation

We have seen that implementing CUDA C++ kernels requires using various extensions that are not part of C++. Once these extensions are used in the code, it is no longer acceptable to a traditional C++ compiler. The code needs to be compiled by a compiler that recognizes and understands these extensions such as NVCC (NVIDIA CUDA Compiler). As shown at the top of Fig. 2.14, the NVCC compiler processes a CUDA C++ program, using the CUDA keywords to separate the host code and device code. The host code is straight C++ code, which is compiled with a standard C/C++ compiler and is run as a traditional CPU process. The device code which is marked with CUDA keywords that designate CUDA kernels and their associated device functions and data structures is compiled by NVCC into virtual binary files called PTX

files. These PTX files are further compiled by a run-time component of NVCC into the real object files and executed on a CUDA-capable GPU device.

2.8 Summary

This chapter provided a quick, simplified overview of the CUDA C++ programming model. CUDA C++ extends the C++ language to support parallel computing. We discussed an essential subset of these extensions in this chapter. For the reader's convenience, we summarize these extensions as follows.

Function Declarations. CUDA C++ extends the C++ function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Fig. 2.11. Using one of "`__global__`", "`__device__`", or "`__host__`", a programmer can instruct the compiler to generate a kernel function, a device function, or a host function. All function declarations without any of these keywords default to host functions. If both the "`__host__`" and "`__device__`" keywords are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host.

Kernel Call and Grid Launch. CUDA C++ extends the C++ function call syntax with kernel execution configuration parameters surrounded by the <<< and >>> brackets. These execution configuration parameters are only used when calling a kernel function to launch a grid. We discussed the execution configuration parameters that define the dimensions of the grid and the dimensions of each block. The reader should refer to the CUDA Programming Guide [3] for more details of the kernel launch extensions as well as other types of execution configuration parameters.

Built-in (Predefined) Variables. CUDA kernels can access a set of built-in read-only variables that allow each thread to distinguish itself from other threads and to determine the area of data to work on. We discussed the `threadIdx`, `blockDim`, and `blockIdx` variables in this chapter. In Chapter 3, we discuss more details of using these variables to manage multi-dimensional blocks and grids while processing multi-dimensional data.

Runtime API. CUDA supports a set of Application Programming Interface (API) functions to provide services to CUDA C++ programs. The services that we discussed in this chapter are the `cudaMalloc`, `cudaFree` and `cudaMemcpy` functions. These functions are called by the host code to allocate device global memory, deallocate device global memory, and transfer data between host and device on behalf of the calling program, respectively. The reader is referred to the CUDA C++ Programming Guide [3] for other CUDA API functions.

Our goal for this chapter is to introduce the core concepts of CUDA C++ and the essential CUDA extensions to C++ for writing a simple CUDA program. The chapter is by no means a comprehensive account of all CUDA features. More features will be covered in the remainder of the book. However, our emphasis will be on the key parallel computing concepts supported by these features. We will only introduce enough CUDA C++ features that are needed in our code examples for parallel programming techniques. In general, we encourage the reader to always consult the CUDA C++ Programming Guide [3] for more details of the latest features of CUDA.

2.9 Exercises

1. If we want to use each thread in a grid to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to the data index (`i`) in our kernel function?
   - **a.** `i=threadIdx.x + threadIdx.y;`
   - **b.** `i=blockIdx.x + threadIdx.x;`
   - **c.** `i=blockIdx.x*blockDim.x + threadIdx.x;`
   - **d.** `i=blockIdx.x * threadIdx.x;`
2. Assume that we want to use each thread to calculate two adjacent elements of a vector addition. What would be the expression for mapping the thread/block indices to the data index (`i`) of the first element to be processed by a thread?
   - **a.** `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
   - **b.** `i=blockIdx.x*threadIdx.x*2;`
   - **c.** `i=(blockIdx.x*blockDim.x + threadIdx.x)*2;`
   - **d.** `i=blockIdx.x*blockDim.x*2 + threadIdx.x;`
3. We want to use each thread to calculate two elements of a vector addition. Each thread block process `2*blockDim.x` consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one element. Assume that variable `i` should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?
   - **a.** `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
   - **b.** `i=blockIdx.x*threadIdx.x*2;`
   - **c.** `i=(blockIdx.x*blockDim.x + threadIdx.x)*2;`
   - **d.** `i=blockIdx.x*blockDim.x*2 + threadIdx.x;`
4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel call to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?
   - **a.** 8000
   - **b.** 8196
   - **c.** 8192
   - **d.** 8200
5. If we want to allocate an array of `v` integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the call to `cudaMalloc`?
   - **a.** `n`
   - **b.** `v`
   - **c.** `n * sizeof(int)`
   - **d.** `v * sizeof(int)`
6. If we want to allocate an array of `n` floating-point elements and have a floating-point pointer variable `A_d` to point to the allocated memory, what would be an appropriate expression for the first argument of the `cudaMalloc` call?

a. `n`
b. `(void *) A_d`
c. `*A_d`
d. `(void **) &A_d`

7. If we want to copy 3000 bytes of data from host array `A_h` (`A_h` is a pointer to element 0 of the source array) to device array `A_d` (`A_d` is a pointer to element 0 of the destination array), what would be an appropriate API call for this data copy in CUDA?

a. `cudaMemcpy(3000, A_h, A_d, cudaMemcpyHostToDevice);`
b. `cudaMemcpy(A_h, A_d, 3000, cudaMemcpyDeviceToHost);`
c. `cudaMemcpy(A_d, A_h, 3000, cudaMemcpyHostToDevice);`
d. `cudaMemcpy(3000, A_d, A_h, cudaMemcpyHostToDevice);`

8. How would one declare a variable `err` that can appropriately receive the returned value of a CUDA API call?

a. `int err;`
b. `cudaError err;`
c. `cudaError_t err;`
d. `cudaSuccess_t err;`

9. Consider the CUDA kernel and the corresponding host function that calls it in Fig. 2.15.

```
01    __global__ void foo_kernel(float* a, float* b, unsigned int N){
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        if(i < N) {
04            b[i] = 2.7f*a[i] - 4.3f;
05        }
06    }
07    void foo(float* a_d, float* b_d) {
08        unsigned int N = 200000;
09        foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d, N);
10    }
```

FIGURE 2.15
CUDA code for Exercise Question 9.

a. What is the number of threads per block?
b. What is the number of threads in the grid?
c. What is the number of blocks in the grid?
d. What is the number of threads that execute the code on line 02?
e. What is the number of threads that execute the code on line 04?

10. A new CUDA programmer is frustrated with CUDA. He has been complaining that CUDA is very tedious: he had to declare many functions that he plans to execute on both the host and the device twice, once as a host function and once as a device function. How would you help this new programmer?

References

[1] Y.N. Patt, S.J. Patel, Introduction to Computing Systems: from Bits and Gates to C and Beyond, McGraw Hill Publisher, 2020, 2000, 2004.
[2] M.J. e. Atallah, Algorithms and Theory of Computation Handbook, CRC Press, 1998.
[3] NVIDIA, CUDA C++ programming guide. Release 13.0, https://docs.nvidia.com/cuda/cuda-c-programming-guide, 2025.

This page intentionally left blank

CHAPTER

Multidimensional grids and data

3

CONTENTS

3.1 Multidimensional grid organization 45
3.2 Mapping threads to MultiDimensional data 48
3.3 Image blur – a more complex kernel 55
3.4 Matrix multiplication 59
3.5 Summary 64

In Chapter 2, we learned to write a simple CUDA C++ program that calls a kernel function to launch a one-dimensional grid of threads that operates on elements of one-dimensional arrays. A kernel specifies the statements that are executed by each individual thread in the grid. In this chapter, we look more generally at how threads are organized and learn how threads and blocks can be used to process multidimensional arrays. Multiple examples are used throughout the chapter, including converting a colored image to a grayscale image, blurring an image, and matrix multiplication. These examples also serve to familiarize the reader with reasoning about data parallelism before we proceed to discuss the GPU architecture, memory organization, and performance optimizations in the upcoming chapters.

3.1 Multidimensional grid organization

In CUDA, all threads in a grid execute the same kernel function, and they rely on their coordinates, i.e., thread indices, to distinguish themselves from each other and to identify the appropriate portion of the data to process. As we saw in Chapter 2, these threads are organized into a two-level hierarchy: a grid consists of one or more blocks and each block in turn consists of one or more threads. All threads in the same block share the same block index, which can be accessed via the `blockIdx` (built-in) variable. Each thread also has a thread index, which can be accessed via the `threadIdx` (built-in) variable. When a thread executes a kernel function, references to the `blockIdx` and `threadIdx` variables return the coordinates of the thread. The execution configuration parameters in a kernel call statement specify the dimensions of the grid and the dimensions of each block. These dimensions are available via the `gridDim` and `blockDim` (built-in) variables.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00012-3

Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

In general, a grid is a three-dimensional array of blocks, and each block is a three-dimensional array of threads. When calling a kernel, the program needs to specify the size of the grid and the blocks in each dimension. These are specified using the execution configuration parameters (within <<< ... >>>) of the kernel call statement. The first execution configuration parameter specifies the dimensions of the grid in number of blocks. The second specifies the dimensions of each block in number of threads. Each such parameter has the type `dim3`, which is an integer vector type of three elements x, y, and z. These three elements specify the sizes of the three dimensions. The programmer can use fewer than three dimensions by setting the size of the unused dimensions to 1.

For example, the following host code can be used to call the `vecAddKernel` kernel function and generate a 1D grid that consists of 32 blocks, each of which consists of 128 threads.

```
dim3 dimGrid(32, 1, 1);
dim3 dimBlock(128, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

In this case, the total number of threads in the grid is $128 * 32 = 4096$. Note that `dimBlock` and `dimGrid` are host code variables defined by the programmer. These variables can have any legal C++ variable name as long as they have the type `dim3`. For example, the following statements accomplish the same as the statements above:

```
dim3 dog(32, 1, 1);
dim3 cat(128, 1, 1);
vecAddKernel<<<dog, cat>>>(...);
```

The grid and block dimensions can also be calculated from other variables. For example, the kernel call in Fig. 2.12 can be written as:

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```

This allows the number of blocks to vary with the size of the vectors so that the grid will have enough threads to cover all vector elements. In this example, the programmer chooses to fix the block size at 256. The value of variable `n` at kernel call time determines the dimension of the grid. If `n` is equal to 1000, the grid will consist of four blocks. If `n` is equal to 4000, the grid will have 16 blocks. In each case, there will be enough threads to cover all the vector elements. Once the grid is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

For convenience, CUDA provides a special shortcut for calling a kernel with one-dimensional grids and blocks. Instead of using `dim3` variables, one can use arithmetic expressions to specify the configuration of 1D grids and blocks. In this case, the CUDA compiler simply takes the arithmetic expression as the `x` dimension and assumes that the `y` and `z` dimensions are 1. This gives us the kernel call statement shown in Fig. 2.12:

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

Readers who are familiar with C++ would realize that this "short-hand" convention for 1D configurations takes advantage of how C++ constructors and default parameters work. The default values of the parameters to the `dim3` constructor are 1. When a single value is passed where a `dim3` is expected, that value will be passed to the first parameter of the constructor while the second and third parameters take the default value of 1. The result is a 1D grid or block where the size of the x-dimension is the value passed and the sizes of the y- and z-dimensions are 1.

Within the kernel function, the `x` field of the variables `gridDim` and `blockDim` are pre-initialized according to the values of the execution configuration parameters. For example, if `n` is equal to 4000, references to `gridDim.x` and `blockDim.x` in the `vecAddKernel` kernel result in 16 and 256 respectively. Note that unlike the `dim3` variables in the host code, the names of these variables within the kernel functions are part of the CUDA C++ specification and cannot be changed. That is, the `gridDim` and `blockDim` variables are built-in variables and always reflect the dimensions of the grid and the blocks, respectively.

In CUDA C++, the allowed values of `gridDim.x` range from 1 to $2^{31} - 1$, and the allowed values of `gridDim.y` and `gridDim.z` range from 1 to $2^{16} - 1$ (65,535). All threads in a block share the same `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` values. Among blocks, the `blockIdx.x` value ranges from 0 to `gridDim.x`-1, the `blockIdx.y` value ranges from 0 to `gridDim.y`-1, and the `blockIdx.z` value ranges from 0 to `gridDim.z`-1.

We now turn our attention to the configuration of blocks. Each block is organized into a three-dimensional array of threads. Two-dimensional blocks can be created by setting `blockDim.z` to 1. One-dimensional blocks can be created by setting both `blockDim.y` and `blockDim.z` to 1, as was in the `vecAddKernel` example. As mentioned before, all blocks in a grid have the same dimensions and sizes. The number of threads in each dimension of a block is specified by the second execution configuration parameter at the kernel call. Within the kernel, this configuration parameter can be accessed as the `x`, `y`, and `z` fields of `blockDim`.

The total size of a block in current CUDA systems is limited to 1024 threads. These threads can be distributed across the three dimensions in any way as long as the total number of threads does not exceed 1024. For example, `blockDim` values of (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowed. On the other hand, a `blockDim` value of (32, 32, 2) is not allowed since the total number of threads exceeds 1024.

A grid and its blocks do not need to have the same dimensionality. A grid can have higher dimensionality than its blocks and vice versa. For example, Fig. 3.1 shows a small toy grid example with a `gridDim` of (2, 2, 1) and a `blockDim` of (4, 2, 2). Such a grid can be created with the following host code:

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

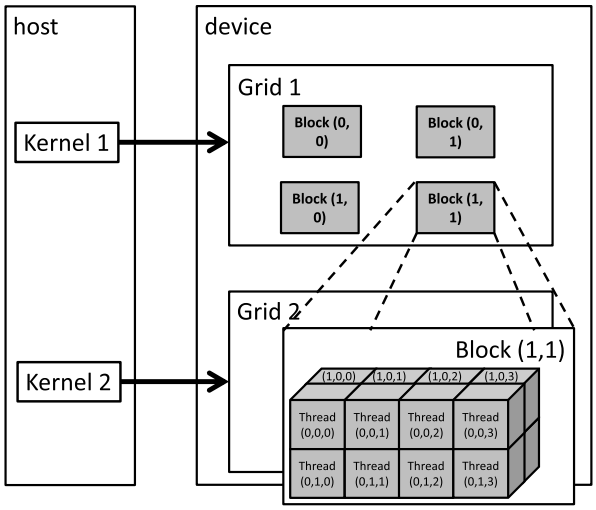


FIGURE 3.1

A multidimensional example of a CUDA grid organization.

The grid in Fig. 3.1 consists of four blocks organized into a 2×2 array. Each block is labeled with (`blockIdx.y`,`blockIdx.x`). For example, Block (1,0) has `blockIdx.y`= 1 and `blockIdx.x`= 0. Note that the ordering of the block and thread labels is such that highest dimension comes first. This notation uses a reverse ordering of that used in the code for setting configuration parameters where the lowest dimension comes first. This reversed ordering for labeling blocks works better when we illustrate the mapping of thread coordinates into data indexes in accessing multidimensional data.

Each `threadIdx` also consists of three fields: the x coordinate `threadId.x`, the y coordinate `threadIdx.y`, and the z coordinate `threadIdx.z`. Fig. 3.1 illustrates the organization of threads within a block. In this example, each block is organized into a 4×2×2 array of threads. Since all blocks within a grid have the same dimensions, we only show one of them. Fig. 3.1 expands Block (1,1) to show its 16 threads. For example, Thread (1,0,2) has `threadIdx.z`= 1, `threadIdx.y`= 0, and `threadIdx.x`= 2. Note that in this example, we have 4 blocks of 16 threads each, with a grand total of 64 threads in the grid. We use these small numbers to keep the illustration simple. Typical CUDA grids contain thousands to millions of threads.

3.2 Mapping threads to MultiDimensional data

The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data. For example, pictures are 2D arrays of pixels. Using a 2D grid that consists of 2D blocks is often convenient for processing the pixels in a picture. Fig. 3.2 shows

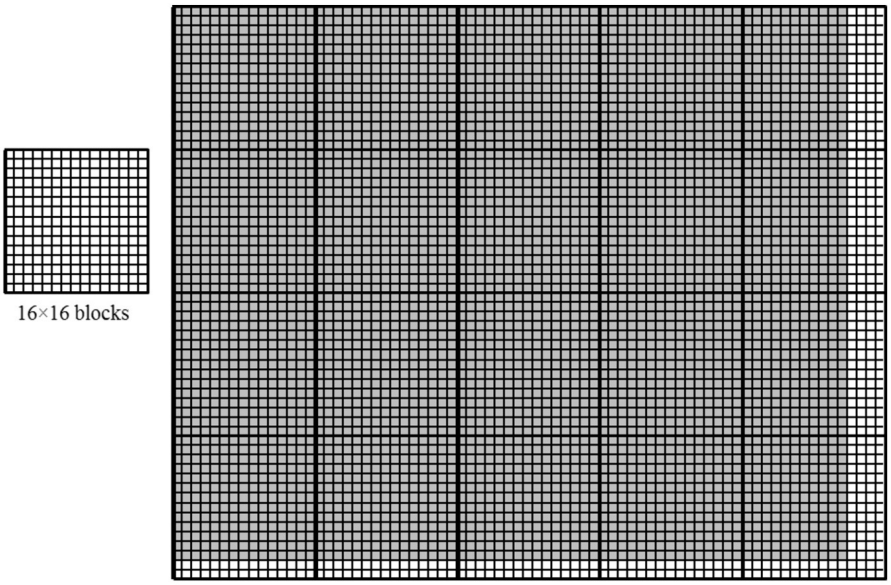


FIGURE 3.2

Using a 2D thread grid to process a 62×76 picture Pin.

such an arrangement for processing a 62×76[1] picture Pin (62 pixels in the vertical or y direction and 76 pixels in the horizontal or x direction). Assume that we decided to use a 16 × 16 block, with 16 threads in the x direction and 16 threads in the y direction. We need 4 blocks in the y direction and 5 blocks in the x direction, which results in $4 \times 5 = 20$ blocks as shown in Fig. 3.2. The heavy lines mark the block boundaries. The shaded area depicts the threads that cover pixels. Each thread is assigned to process a pixel whose y and x coordinates are derived from its `blockIdx`, `blockDim`, and `threadIdx` variable values:

Vertical (row) coordinate = `blockIdx.y*blockDim.y + threadIdx.y`

Horizontal (column) coordinate = `blockIdx.x*blockDim.x + threadIdx.x`

For example, the Pin element to be processed by Thread (0,0) of Block (1,0) can be identified as follows:

[1] Throughout this book, we refer to the dimensions of multidimensional data in descending order: the z dimension followed by the y dimension followed by the x dimension. For example, for a picture with n pixels in the vertical or y dimension and m pixels in the horizontal or x dimension, we refer to it as a $n \times m$ picture. This follows the C++ multidimensional array indexing convention. For example, we can refer to `P[y][x]` as $P_{y,x}$ in text and figures for conciseness. Unfortunately, this ordering is opposite to the order in which data dimensions are ordered in the `gridDim` and `blockDim` dimensions. The discrepancy can be especially confusing when we define the dimensions of a thread grid based on a multidimensional array that is to be processed by its threads.

$$Pin_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} = Pin_{1*16+0,0*16+0} = Pin_{16,0}$$

Note that in Fig. 3.2, we have 2 extra threads in the y direction and 4 extra threads in the x direction. That is, we generate a grid of 64×80 threads in total to process 62×76 pixels. This situation is similar to when a 1000-element vector is processed using four 256-thread blocks by the 1D kernel `vecAddKernel` in Fig. 2.10. Recall that an if-statement in Fig. 2.10 is needed to prevent the extra 24 threads from taking effect. Similarly, we should expect that the picture processing kernel function will have if-statements to test whether the thread's vertical and horizontal indices fall within the valid range of pixels.

We assume that the host code uses an integer variable `n` to track the number of pixels in the y direction and another integer variable `m` to track the number of pixels in the x direction. We further assume that the input picture data has been copied to the device global memory and can be accessed through a pointer variable `Pin_d`. The output picture has been allocated in the device memory and can be accessed through a pointer variable `Pout_d`. The following host code can be used to call a 2D kernel `colorToGrayscaleConversion` to process the picture:

```
dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);
dim3 dimBlock(16, 16, 1);
colorToGrayscaleConversion<<<dimGrid,dimBlock>>>(Pin_d, Pout_d, m, n);
```

In this example, we assume for simplicity that the dimensions of the blocks are fixed at 16 × 16. The dimensions of the grid, on the other hand, depend on the dimensions of the picture. To process a 1500×2000 (3-million-pixel) picture, we would generate 11,750 blocks, 94 in the y direction and 125 in the x direction. Within the kernel function, references to `gridDim.x`, `gridDim.y`, `blockDim.x`, and `blockDim.y` result in 125, 94, 16, and 16, respectively.

Before we show the kernel code, we need to first understand how C and C++ statements access elements of dynamically allocated multidimensional arrays. Ideally, we would like to access `Pin_d` as a two-dimensional array where an element at the row `j` and column `i` can be accessed as `Pin_d[j][i]`. However, the C++ standard that CUDA C++ is based on requires the number of columns in `Pin` to be known at compile time for `Pin` to be accessed as a 2D array. Unfortunately, this information is not known at compile time for dynamically allocated arrays. In fact, part of the reason why one uses dynamically allocated arrays is to allow the sizes and dimensions of these arrays to vary according to the data size at run time. Thus, the information on the number of columns in a dynamically allocated two-dimensional array is not known at compile time by design. As a result, programmers need to explicitly linearize, or "flatten", a dynamically allocated two-dimensional array into an equivalent one-dimensional array in the current CUDA C++.

In reality, all multidimensional arrays in C and C++ are linearized. This is due to the use of a "flat" memory space in modern computers (see "Memory Space" sidebar). In the case of statically allocated arrays, the compilers allow the programmers to use higher dimensional indexing syntax such as `Pin_d[j][i]` to access their elements.

Under the hood, the compiler linearizes them into an equivalent one-dimensional array and translates the multidimensional indexing syntax into a one-dimensional offset. In the case of dynamically allocated arrays, such translation is left to the programmer due to lack of dimensional information at compile time that is needed to perform the translation.

Memory Space

A memory space is a simplified view of how a processor accesses its memory in modern computers. A memory space is usually associated with each running application. The data to be processed by an application and the instructions executed for the application are stored in locations in its memory space. Each location typically can accommodate a byte and has an address. Variables that require multiple bytes – 4 bytes for `float` and 8 bytes for `double` – are stored in consecutive byte locations. When accessing a data value from the memory space, the processor gives the starting address (address of the starting byte location) and the number of bytes needed. Most modern computers have at least 4G byte-sized locations, where each G is 1,073,741,824 (2^{30}). All locations are labeled with an address that ranges from 0 to the largest number used. Since there is only one address for every location, we say that the memory space has a "flat" organization. As a result, all multidimensional arrays are ultimately "flattened", into equivalent one-dimensional arrays. While a C++ programmer can use multidimensional array syntax to access an element of a multidimensional array, the compiler translates these accesses into a base pointer that points to the beginning element of the array, along with a one-dimensional offset calculated from these multidimensional indices.

There are at least two ways one can linearize a two-dimensional array. One way is to place all elements of the same row into consecutive locations. The rows are then placed one after another into the memory space. This arrangement, called the *row-major layout*, is illustrated in Fig. 3.3. To improve readability, we use $M_{j,i}$ to denote an element of M at the jth row and the ith column. $M_{j,i}$ is equivalent to the C++ expression `M[j][i]`, but is slightly more readable. Fig. 3.3 shows an example where a 4×4 matrix M is linearized into a 16-element one-dimensional array, with the four elements of row 0 first, followed by the four elements of row 1, etc. Therefore, the one-dimensional equivalent index for an element of M at row j and column i is $j * 4 + i$. The $j * 4$ term skips over all elements of the rows before row j. The i term then selects the right element within the section for row j. For example, the one-dimensional index for $M_{2,1}$ is $2 * 4 + 1 = 9$. This is illustrated in Fig. 3.3, where M_9 is the one-dimensional equivalent to $M_{2,1}$. This is the way C and C++ compilers linearize two-dimensional arrays.

Another way to linearize a two-dimensional array is to place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space. This arrangement, called the *column-major layout*,

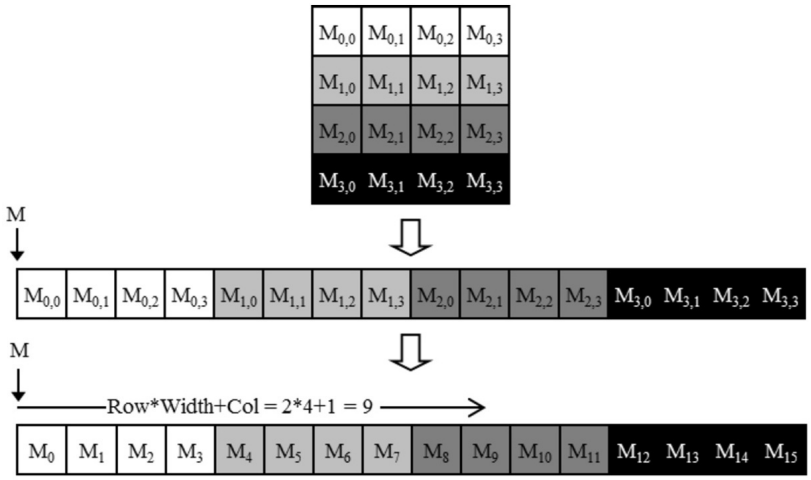


FIGURE 3.3

Row-major layout for a 2D C++ array. The result is an equivalent 1D array accessed by an index expression $j * Width + i$ for an element that is in the jth row and i^{th} column of an array of $Width$ elements in each row.

is used by FORTRAN compilers. Note that the column-major layout of a two-dimensional array is equivalent to the row-major layout of its transposed form. We will not spend more time on this layout except mentioning that readers whose primary previous programming experience was with FORTRAN should be aware that CUDA C++ uses the row-major layout rather than the column major layout. Also, many C libraries that are designed to be used by FORTRAN programs use column-major layout to match the FORTRAN compiler layout. As a result, the manual pages for these libraries usually tell the users to transpose the input arrays if they call these libraries from C or C++ programs.

We are now ready to study the source code of the `colorToGrayscaleConversion` kernel, shown in Fig. 3.4. The kernel code uses the following equation to convert each color pixel to its grayscale counterpart:

$$L = 0.299*r + 0.587*g + 0.114*b$$

There are a total of `blockDim.x*gridDim.x` threads in the horizontal direction. Similar to the `vecAddKernel` example, the following expression generates every integer value from 0 to `blockDim.x*gridDim.x`−1 (line 06):

```
col = blockIdx.x*blockDim.x + threadIdx.x
```

We know that `gridDim.x*blockDim.x` is greater than or equal to the width of the image (`m` value passed in from the host code). In other words, we have at least as many threads as the number of pixels in the horizontal direction. Similarly, we also know that there are at least as many threads as the number of pixels in the vertical direction.

```
01  // The input image is encoded as unsigned chars [0, 255]
02  // Each pixel is 3 consecutive chars for the 3 channels (RGB)
03  __global__
04  void colortoGrayscaleConvertion(unsigned char * Pout,
05                  unsigned char * Pin, int width, int height) {
06      int col = blockIdx.x*blockDim.x + threadIdx.x;
07      int row = blockIdx.y*blockDim.y + threadIdx.y;
08      if (col < width && row < height) {
09          // Get 1D offset for the grayscale image
10          int grayOffset = row*width + col;
11          // One can think of the RGB image having CHANNEL
12          // times more columns than the gray scale image
13          int rgbOffset = grayOffset*CHANNELS;
14          unsigned char r = Pin[rgbOffset    ]; // Red value
15          unsigned char g = Pin[rgbOffset + 1]; // Green value
16          unsigned char b = Pin[rgbOffset + 2]; // Blue value
17          // Perform the rescaling and store it
18          // We multiply by floating point constants
19          Pout[grayOffset] = (unsigned char) 0.21f*r+0.71f*g+0.07f*b;
20      }
21  }
```

FIGURE 3.4

Source code of `colorToGrayscaleConversion` with 2D thread mapping to data.

Therefore, as long as we test and make sure only the threads with both `row` and `col` values within range participate in the computation, i.e., `(col<width) && (row<height)`, we should be able to cover every pixel in the picture (line 07).

Since there are `width` pixels in each row, we can generate the one-dimensional index for the pixel at row `row` and column `col` as `row*width+col` (line 10). This one-dimensional index `grayOffset` is the pixel index for `Pout` since each pixel in the output grayscale image is one byte (unsigned char). Using our 62×76 image example, the linearized one-dimensional index of the `Pout` pixel calculated by Thread (0,0) of Block (1,0) with the formula:

$$Pout_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} =$$
$$Pout_{1*16+0,0*16+0} = Pout_{16,0} = \texttt{Pout[16*76+0] = Pout[1216]}$$

As for `Pin`, we need to multiply the gray pixel index by 3 (line 13)[2] since each colored pixel is stored as three elements (r, g, b), each of which is one byte. The resulting `rgbOffset` gives the starting location of the color pixel in the `Pin` array. We read the `r`, `g`, and `b` value from the three consecutive byte locations of the `Pin` array (lines 14-16), perform the calculation of the grayscale pixel value, and write that value into the `Pout` array using `grayOffset` (line 19). Using our 62×76 image example, the linearized one-dimensional index of the first component of the `Pin` pixel that is processed by Thread (0,0) of Block (1,0) can be calculated with the formula:

[2] We assume that `CHANNELS` is a constant of value 3 and its definition is outside the kernel function.

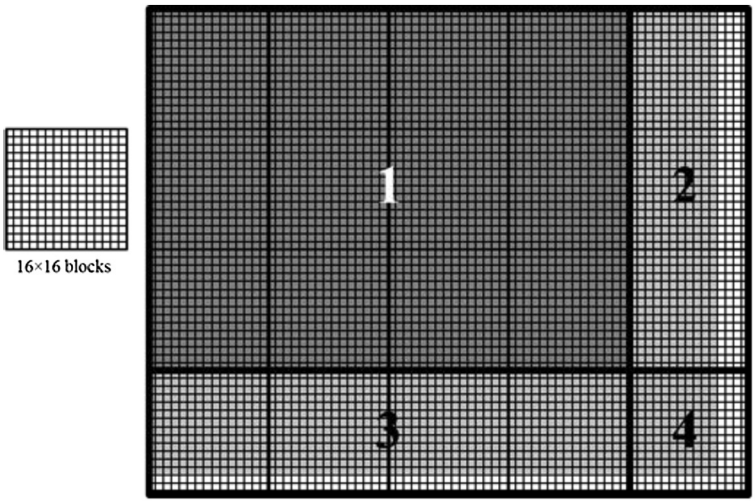


FIGURE 3.5

Covering a 76 × 62 picture with 16 × 16 blocks.

$$Pin_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} = Pin_{1*16+0,0*16+0} = Pin_{16,0} = \texttt{Pin[16*76*3+0] = Pin[3648]}$$

The data being accessed are the three bytes starting at byte offset 3648.

Fig. 3.5 illustrates the execution of `colorToGrayscaleConversion` when processing our 62 × 76 example. Assuming 16 × 16 blocks, calling the `colorToGrayscaleConversion` kernel generates 64 × 80 threads. The grid will have 4 × 5 = 20 blocks, 4 in the vertical direction and 5 in the horizontal direction. The execution behavior of blocks will fall into one of four different cases, shown as four shaded areas in Fig. 3.5.

The first area, marked as 1 in Fig. 3.5, consists of the threads that belong to the 12 blocks covering the majority of pixels in the picture. Both `col` and `row` values of these threads are within range; all these threads pass the if-statement test and process pixels in the dark shaded area of the picture. That is all 16 × 16 = 256 threads in each block will process pixels.

The second area, marked as 2 in Fig. 3.5, contains the threads that belong to the three blocks in the medium shaded area covering the upper right pixels of the picture. Although the `row` values of these threads are always within range, the `col` values of some of them exceed the `m` value of 76. This is because the number of threads in the horizontal direction is always a multiple of the `blockDim.x` value chosen by the programmer (16 in this case). The smallest multiple of 16 needed to cover 76 pixels is 80. As a result, 12 threads in each row will find their `col` values within range and will process pixels. On the other hand, the remaining 4 threads in each row will find their `col` values out of range, and thus fail the if-statement condition. These threads

will not process any pixels. Overall, $12 \times 16 = 192$ of the $16 \times 16 = 256$ threads in each of these blocks will process pixels.

The third area, marked as 3 in Fig. 3.5, accounts for the four lower left blocks covering the medium shaded area of the picture. Although the `col` values of these threads are always within range, the `row` values of some of them exceed the `n` value of 62. This is because the number of threads in the vertical direction is always a multiple of the `blockDim.y` value chosen by the programmer (16 in this case). The smallest multiple of 16 to cover 62 is 64. As a result, 14 threads in each column will find their row values within range and will process pixels. On the other hand, the remaining 2 threads in each column will not pass the if-statement and will not process any pixels. Overall, $16 \times 14 = 224$ of the 256 threads will process pixels.

The fourth area, marked as 4 in Fig. 3.5, contains the threads that cover the lower right, lightly shaded area of the picture. Like Area 2, 4 threads in each of the top 14 rows will find their col values out of range. Like Area 3, the entire bottom two rows of this block will find their row values out of range. Overall, only $14 \times 12 = 168$ of the $16 \times 16 = 256$ threads will process pixels.

We can easily extend our discussion of 2D arrays to 3D arrays by including another dimension when we linearize arrays. This is done by placing each "plane" of the array one after another into the address space. Assume that the programmer uses variables `m` and `n` to track the number of columns and rows in a 3D array, respectively. The programmer also needs to determine the values of `blockDim.z` and `gridDim.z` when calling a kernel. In the kernel, the array index will involve another global index:

```
int plane = blockIdx.z*blockDim.z + threadIdx.z;
```

The linearized access to a three-dimensional array `P` will be in the form of `P[plane*m*n + row*m + col]`. A kernel processing the 3D `P` array needs to check whether all the three global indices, `plane`, `row`, and `col` fall within the valid range of the array. The use of 3D arrays in CUDA kernels will be further studies for the stencil pattern in Chapter 8.

3.3 Image blur – a more complex kernel

So far, we have studied two kernels, `vecAddKernel` and `colorToGrayscaleConversion`, where each thread performs only a small number of arithmetic operations on one array element. These kernels serve their purposes well: to illustrate the basic CUDA program structure and data parallel execution concepts. At this point, the reader should ask the obvious question: do all threads in CUDA programs perform only such simple and trivial operations independently of each other? The answer is no. In real CUDA programs, threads often perform complex operations on their data and need to cooperate with each other. For the next few chapters, we are going to work on increasingly more complex examples that exhibit these characteristics. We will start with an image blurring function.

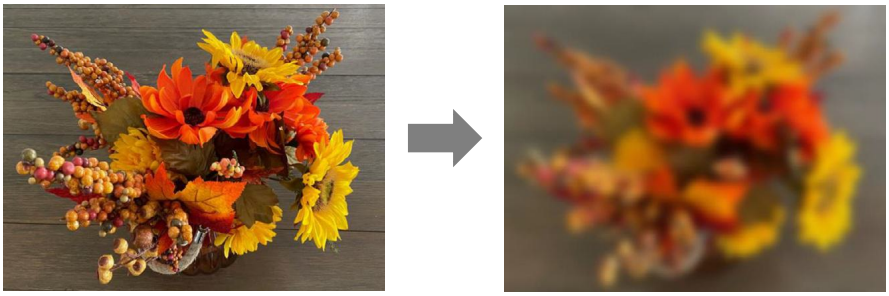


FIGURE 3.6

An original image (left) and a blurred version (right).

Image blurring smoothens out abrupt variation of pixel values in an image. Fig. 3.6 illustrates the effect of image blurring. Simply stated, we make the image blurry. To human eyes, a blurred image tends to obscure the fine details and present the "big picture" impression, or the major thematic objects in the picture. In computer image processing algorithms, a common use case of image blurring is to reduce the impact of noise and granular rendering effects in an image by correcting problematic pixel values with the clean surrounding pixel values. In computer vision, image blurring can be used to allow edge detection and object recognition algorithms to focus on thematic objects rather than being bogged down by a massive quantity of fine-grained objects. In displays, image blurring is sometimes used to highlight a particular part of the image by blurring the rest of the image.

Mathematically, an image blurring function calculates the value of an output image pixel as a weighted sum of a patch of pixels encompassing the original pixel in the input image. As we will learn in Chapter 7, the computation of such weighted sums belongs to the convolution pattern. We will use a simplified approach in this chapter by taking a simple average value of an $N \times N$ patch of pixels surrounding, and including, the target pixel. To keep the algorithm simple, we will not place a weight on the value of any pixel based on its distance from the target pixel. In practice, placing such weights is quite common in convolution blurring approaches such as Gaussian Blur.

Fig. 3.7 shows an example of image blurring using a 3×3 patch. When calculating an output pixel value at the (row, col) position, we see that the patch is centered at the input pixel located at the (row, col) position. The 3×3 patch spans three rows (row-1, row, row+1) and three columns (col-1, col, col+1). For example, the coordinates of the nine pixels for calculating the output pixel at (25, 50) are (24, 49), (24, 50), (24, 51), (25, 49), (25, 50), (25, 51), (26, 49), (26, 50), and (26, 51).

Fig. 3.8 shows an image blur kernel. Similar to the strategy used in the kernel `colorToGrayscaleConversion`, we use each thread to calculate an output pixel. That is, the thread-to-output-data mapping remains the same. At the beginning of the kernel, we see the familiar calculation of the `col` and `row` indices (lines 03-04). We also see the familiar if-statement that verifies that both `col` and `row` are within the valid range

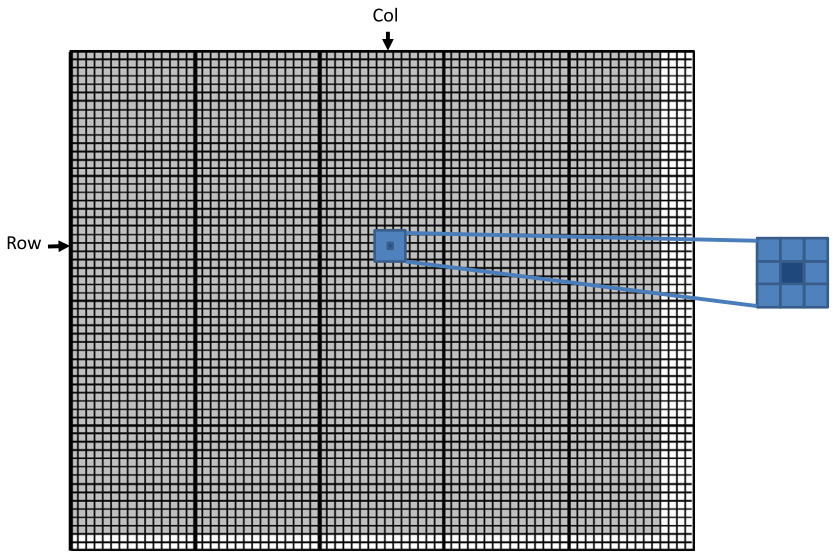


FIGURE 3.7

Each output pixel is the average of a patch of surrounding pixels and itself in the input image.

```
01  __global__
02  void blurKernel(unsigned char *in, unsigned char *out, int w, int h){
03    int col = blockIdx.x*blockDim.x + threadIdx.x;
04    int row = blockIdx.y*blockDim.y + threadIdx.y;
05    if(col < w && row < h) {
06      int pixVal = 0;
07      int pixels = 0;
09          // Get average of the surrounding BLUR_SIZE x BLUR_SIZE box
10      for(int blurRow=-BLUR_SIZE; blurRow<BLUR_SIZE+1; ++blurRow){
11         for(int blurCol=-BLUR_SIZE; blurCol<BLUR_SIZE+1; ++blurCol){
12            int curRow = row + blurRow;
13            int curCol = col + blurCol;
14                  // Verify we have a valid image pixel
15             if(curRow>=0 && curRow<h && curCol>=0 && curCol<w) {
16                 pixVal += in[curRow*w + curCol];
17                 ++pixels; // Keep track of number of pixels in the avg
18               }
19            }
20          }
21          // Write our new pixel value out
22          out[row*w + col] = (unsigned char)((float)pixVal/pixels);
23      }
24  }
```

FIGURE 3.8

An image blur kernel.

according to the height `h` and width `w` of the image (line 05). Only the threads whose `col` and `row` indices are both within valid ranges are allowed to participate in the execution.

As shown in Fig. 3.7, the `col` and `row` values also give the central pixel location of the patch of input pixels used for calculating the output pixel for the thread. The nested for-loops in Fig. 3.8 (lines 10-11) iterate through all the pixels in the patch. We assume that the program has a defined constant `BLUR_SIZE`. The value of `BLUR_SIZE` is set such that `BLUR_SIZE` gives the number of pixels on each side of the patch, also called its radius. In this case, 2∗`BLUR_SIZE`+1 gives the total number of pixels across one dimension of the patch. For example, for a 3×3 patch, `BLUR_SIZE` is set to 1, whereas for a 7×7 patch, `BLUR_SIZE` is set to 3. The outer loop iterates through the rows of the patch. For each row, the inner loop iterates through the columns of the patch.

In our 3×3 patch example, the `BLUR_SIZE` is 1. For the thread that calculates output pixel (25, 50), during the first iteration of the outer loop, the `curRow` variable is `row`−`BLUR_SIZE` $= (25 - 1) = 24$. Thus, during the first iteration of the outer loop, the inner loop iterates through the patch pixels in row 24. The inner loop iterates from column `col`−`BLUR_SIZE` $= 50 - 1 = 49$ to `col`+`BLUR_SIZE` $= 51$ using the `curCol` variable. Therefore, the pixels processed in the first iteration of the outer loop are (24, 49), (24, 50), and (24, 51). The reader should verify that in the second iteration of the outer loop, the inner loop iterates through pixels (25, 49), (25, 50), and (25, 51). Finally, the in the third iteration of the outer loop, the inner loop iterates through pixels (26, 49), (26, 50), and (26, 51).

Line 16 uses the linearized index of `curRow` and `curCol` to access the value of the input pixel visited in the current iteration. It accumulates the pixel value into a running sum variable `pixVal`. Line 17 records the fact that one more pixel value has been added into the running sum by incrementing the `pixels` variable. After all the pixels in the patch are processed, line 22 calculates the average value of the pixels in the patch by dividing the `pixVal` value by the `pixels` value. It uses the linearized index of `row` and `col` to write the result into its output pixel.

Line 15 contains a conditional statement that guards the execution of lines 16 and 17. For example, when computing output pixels near the edge of the image, the patch may extend beyond the valid range of the input image. This situation is illustrated in Fig. 3.9 assuming 3×3 patches. In Case 1, the pixel at the upper left corner is being blurred. Five out of the nine pixels in the intended patch do not exist in the input image. In this case, the `row` and `col` values of the output pixel are 0 and 0, respectively. During the execution of the nested loop, the `curRow` and `curCol` values for the nine iterations are (−1,-1), (−1,0), (−1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1). Note that for the five pixels that are outside the image, at least one of the index values is less than 0. The `curRow>=0` and `curCol>=0` conditions of the if-statement catch these values and skips the execution of lines 16 and 17. As a result, only the values of the four valid pixels are accumulated into the running sum variable. The `pixels` value is also correctly incremented only four times so that the average can be calculated properly at line 22.

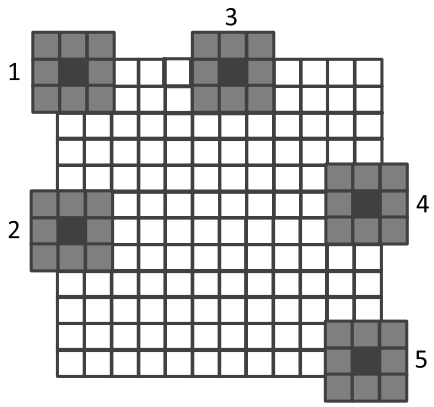


FIGURE 3.9

Handling boundary conditions for pixels near the edges of the image.

The readers should work through the other cases in Fig. 3.9 and analyze the execution behavior of the nested loop in `blurKernel`. Note that most of the threads find all the pixels in their assigned 3×3 patch within the input image. They accumulate all the nine pixels. However, for the pixels on the four corners, the responsible threads accumulate only 4 pixels. For other pixels on the four edges, the responsible threads accumulate 6 pixels. These variations are what necessitates keeping track of the actual number of pixels accumulated with variable pixels.

3.4 Matrix multiplication

Matrix-matrix multiplication, or matrix multiplication in short, is an important component of the Basic Linear Algebra Subprograms (BLAS) standard (see "Linear Algebra Functions" sidebar). It is the basis of many linear algebra solvers such as LU decomposition. It is also an important computation for deep learning, as we will see in Chapter 19 on convolutional neural networks and Chapter 20 on large language models.

Linear Algebra Functions

Linear algebra operations are widely used in science and engineering applications. In the Basic Linear Algebra Subprograms (BLAS), a de facto standard for publishing libraries that perform basic algebra operations, there are three levels of linear algebra functions. As the level increases, the number of operations performed by the function increases. Level-1 functions perform vector operations of the form $y = \alpha x + y$, where x and y are vectors and α is a scalar. Our vector addition example in Chapter 2 is a special case of a level-1 BLAS

function with $\alpha = 1$. Level-2 functions perform matrix-vector operations of the form $y = \alpha Ax + \beta y$, where A is a matrix, x and y are vectors, and α and β are scalars. We will be studying a form of level-2 functions Chapter 17 on sparse linear algebra. Level-3 functions perform matrix-matrix operations of the form $C = \alpha AB + \beta C$, where A, B, and C are matrices, and α and β are scalars. Our matrix-matrix multiplication example is a special case of a level-3 function where $\alpha = 1$ and $\beta = 0$. These BLAS functions are important because they are used as basic building blocks of higher-level algebraic functions such as linear system solvers and eigenvalue analysis. As we will discuss later, the performance of different implementations of BLAS functions can vary by orders of magnitude in both sequential and parallel computers.

Matrix multiplication between an $i \times j$ (i rows by j columns) matrix M and a $j \times k$ matrix N produces an $i \times k$ matrix P. When performing a matrix multiplication, each element of the output matrix P is an inner product of a row of M and a column of N. We will continue to use the convention where $P_{row,col}$ is the element at row^{th} position in the vertical direction and col^{th} position in the horizontal direction. As shown in Fig. 3.10, $P_{row,col}$ (the small square in P) is the inner product of the vector formed by the row^{th} row of M (shown as a horizontal strip in M) and the vector formed by the col^{th} column of N (shown as a vertical strip in N). The inner product, sometimes called the dot product, of two vectors is the sum of products of the individual vector elements. That is:

$$P_{row,col} = \sum M_{row,k} * N_{k,col} \text{ for } k = 0, 1, ..., Width - 1$$

For example, in Fig. 3.10, assuming $row = 1$ and $col = 5$:

$$P_{1,5} = M_{1,0} * N_{0,5} + M_{1,1} * N_{1,5} + M_{1,2} * N_{2,5} + ... + M_{1,Width-1} * N_{Width-1,5}$$

To implement matrix multiplication using CUDA, we can map the threads in the grid to the elements of the output matrix P with the same approach that we used for the `colorToGrayscaleConversion` kernel. That is, each thread is responsible for calculating one P element. The row and column indices for the P element to be calculated by each thread are the same as before:

```
row = blockIdx.y*blockDim.y + threadIdx.y
```

and

```
col = blockIdx.x*blockDim.x + threadIdx.x.
```

With this one-to-one mapping, the `row` and `col` thread indices are also the row and column indices for their output elements. Fig. 3.11 shows the source code of the kernel based on this thread-to-data mapping. The reader should immediately see the familiar pattern of calculating `row` and `col` (lines 03-04), and the if-statement testing if `row` and `col` are both within range (line 05). These statements are almost identical

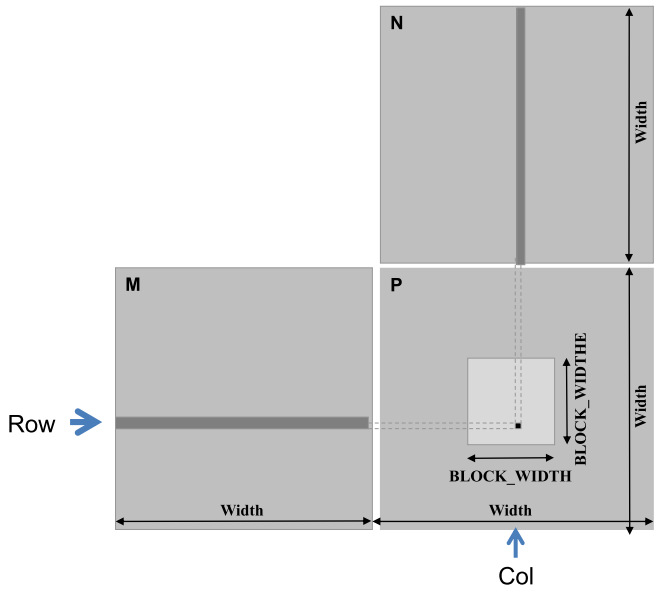


FIGURE 3.10

Matrix multiplication using multiple blocks by tiling P.

```
01    __global__ void MatrixMulKernel(float* M, float* N,
02                                    float* P, int Width) {
03        int row = blockIdx.y*blockDim.y+threadIdx.y;
04        int col = blockIdx.x*blockDim.x+threadIdx.x;
05        if ((row < Width) && (col < Width)) {
06            float Pvalue = 0;
07            for (int k = 0; k < Width; ++k) {
08                Pvalue += M[row*Width+k]*N[k*Width+col];
09            }
10            P[row*Width+col] = Pvalue;
11        }
12    }
```

FIGURE 3.11

A matrix multiplication kernel using one thread to compute one `P` element.

to their counterparts in the `colorToGrayscaleConversion` kernel. The only significant difference is that we are making a simplifying assumption that `matrixMulKernel` needs to handle only square matrices so we replace both `width` and `height` with just `Width`. This thread-to-data mapping effectively divides P into tiles, one of which is shown as a large, light-colored square in Fig. 3.10. Each block is responsible for calculating one of these tiles.

We now turn our attention to the work done by each thread. Recall that $P_{row,col}$ is calculated as the inner product of the row^{th} row of M and the col^{th} column of N. In Fig. 3.11, we use a for-loop to perform this inner product operation. Before we

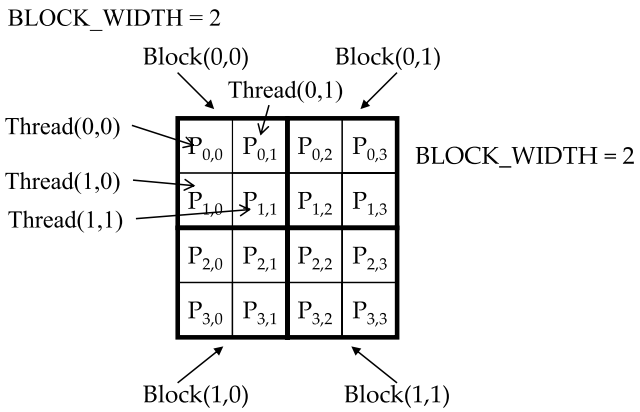


FIGURE 3.12

A small execution example of matrixMulKernel.

enter the loop, we initialize a local variable `Pvalue` to 0 (line 06). Each iteration of the loop accesses an element from the `row`th row of `M` and one from the `col`th column of `N`, multiplies the two elements together, and accumulates the product into `Pvalue` (line 08).

Let us first focus on accessing the `M` element within the for-loop. `M` is linearized into an equivalent 1D array using row major order. That is, the rows of `M` are placed one after another in the memory space, starting with the 0th row. Therefore, the beginning element of row 1 is `M[1*Width]` because we need to account for all elements of row 0. In general, the beginning element of the `row`th row is `M[row*Width]`. Since all the elements of a row are placed in consecutive locations, the `k`th element of the `row`th row is at `M[row*Width+k]`. This linearized array offset is what we use in Fig. 3.11 (line 08).

We now turn our attention to accessing `N`. As shown in Fig. 3.11, the beginning element of the `col`th column is the `col`th element of row 0, which is `N[col]`. Accessing the next element in the `col`th column requires skipping over an entire row. This is because the next element of the same column is the same element in the next row. Therefore, the `k`th element of the `col`th column is `N[k*Width+col]` (line 08).

After the execution exits the for-loop, all threads have their `P` element values in the `Pvalue` variables. Each thread then uses the one-dimensional equivalent index expression `row*Width+col` to write its `P` element (line 10). Again, this index pattern is like that used in the `colorToGrayscaleConversion` kernel.

We now use a small example to illustrate the execution of the matrix multiplication kernel. Fig. 3.12 shows a 4×4 matrix P with BLOCK_WIDTH $= 2$. Although such small matrix and block sizes are not realistic, they allow us to fit the entire example in one picture. The P matrix is divided into four tiles and each block calculates one tile. We do so by creating blocks that are 2×2 arrays of threads, with each thread calculating one P element. In the example, Thread (0,0) of Block (0,0) calculates $P_{0,0}$ whereas Thread (0,0) of Block (1,0) calculates $P_{2,0}$.

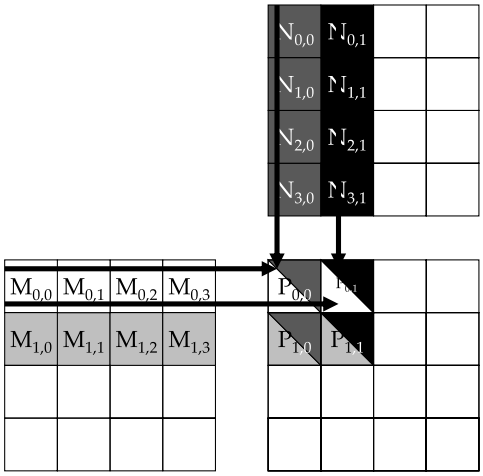


FIGURE 3.13

Matrix multiplication actions of one thread block.

The `row` and `col` indices in `matrixMulKernel` identify the P element to be calculated by a thread. The `row` index also identifies the row of M and the `col` index identifies the column of N as input values for the thread. Fig. 3.13 illustrates the multiplication actions in each thread block. For the small matrix multiplication example, threads in Block (0,0) produce four dot products. The `row` and `col` indices of Thread (1,0) in Block (0,0) are $0 * 0 + 1 = 1$ and $0 * 0 + 0 = 0$, respectively. The thread thus maps to $P_{1,0}$ and calculates the dot product of row 1 of M and column 0 of N.

Let us walk through the execution of the for-loop of Fig. 3.11 for Thread (0,0) in Block (0,0). During iteration 0 ($k = 0$), $row * Width + k = 0 * 4 + 0 = 0$ and $k * Width + col = 0 * 4 + 0 = 0$. Therefore, the input elements accessed are $M[0]$ and $N[0]$, which are the 1D equivalent of $M_{0,0}$ and $N_{0,0}$. Note that these are indeed the 0th elements of row 0 of M and column 0 of N. During iteration 1 ($k = 1$), $row * Width + k = 0 * 4 + 1 = 1$ and $k * Width + col = 1 * 4 + 0 = 4$. Therefore, we are accessing $M[1]$ and $N[4]$, which are the 1D equivalent of $M_{0,1}$ and $N_{1,0}$. These are the 1st elements of row 0 of M and column 0 of N. During iteration 2 ($k = 2$), $row * Width + k = 0 * 4 + 2 = 2$ and $k * Width + col2 * 4 + 0 = 8$, which results in $M[2]$ and $N[8]$. Therefore, the elements accessed are the 1D equivalent of $M_{0,2}$ and $N_{2,0}$. Finally, during iteration 3 ($k = 3$), $row * Width + k = 0 * 4 + 3 = 3$ and $k * Width + col = 3 * 4 + 0 = 12$, which results in $M[3]$ and $N[12]$, the 1D equivalent of $M_{0,3}$ and $N_{3,0}$. We now have verified that the for-loop performs the inner product between the 0th row of M and the 0th column of N for Thread (0,0) in Block (0,0). After the loop, the thread writes `P[row*Width+col]`, which is $P[0]$. This is the 1D equivalent of $P_{0,0}$ so Thread (0,0) in Block (0,0) successfully calculated the inner product between the 0th row of M and the 0th column of N and deposited the result in $P_{0,0}$. We will leave it as an exercise for the reader to hand-execute and verify the for-loop for other threads in Block (0,0) or in other blocks.

Since the size of a grid is limited by the maximum number of blocks per grid and threads per block, the size of the largest output matrix `P` that can be handled by `matrixMulKernel` is also be limited by these constraints. In the situation where output matrices larger than this limit are to be computed, one can divide the output matrix into sub-matrices whose sizes can be covered by a grid and use the host code to launch a different grid for each sub-matrix. Alternatively, we can change the kernel code so that each thread calculates multiple elements of P. We will further optimize our implementation of matrix multiplication in Chapter 5 and dive deeply into advanced optimizations for this important computation in Chapter 15.

3.5 Summary

CUDA grids and blocks are multidimensional with up to three dimensions. The multidimensionality of grids and blocks is useful for organizing threads to be mapped to multidimensional data. The kernel execution configuration parameters define the dimensions of a grid and its blocks. Unique coordinates in `blockIdx` and `threadIdx` allow threads of a grid to identify themselves and their domains of data. It is the programmer's responsibility to use these variables in kernel functions so that the threads can properly identify the portion of the data to process. When accessing multidimensional data, programmers will often have to linearize multidimensional indices into a one-dimensional offset. The reason is that dynamically allocated multidimensional arrays in C++ are typically stored as one-dimensional arrays in row-major order. We use examples of increasing complexity to familiarize the reader with the mechanics of processing multidimensional arrays with multidimensional grids. These skills are foundational for understanding the parallel patterns to be presented in the book and their associated optimization techniques.

1. In this chapter, we implemented a matrix multiplication kernel that has each thread produce one output matrix element. In this question, you will implement different matrix-matrix multiplication kernels and compare them.
   a. Write a kernel that has each thread produce one output matrix row. Fill in the execution configuration parameters for the design.
   b. Write a kernel that has each thread to produce one output matrix column. Fill in the execution configuration parameters for the design.
   c. Analyze the pros and cons of each kernel design above.
2. A matrix-vector multiplication takes an input matrix B and a vector c and produces one output vector a. Each element of the output vector a is the dot product of one row of the input matrix B and c, i.e., $a_i = \sum_j B_{i,j} \times c_j$. For simplicity, we will only handle square matrices whose elements are single precision floating-point number. Write a matrix-vector multiplication kernel and the host stub function that can be called with four parameters: pointer to the output vector a, pointer to the input matrix B, pointer to the input vector c, and the number of elements in each dimension N. Use one thread to calculate an output vector element.

3. Consider the following CUDA kernel and the corresponding host function that calls it:

```
01    __global__ void foo_kernel(float* a, float* b, unsigned int M,
          unsigned int N) {
02        unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
03        unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
04        if(row < M && col < N) {
05            b[row*N + col] = a[row*N + col]/2.1f + 4.8f;
06        }
07    }
08    void foo(float* a_d, float* b_d) {
09        unsigned int M = 150;
10        unsigned int N = 300;
11        dim3 bd(16, 32);
12        dim3 gd((N - 1)/16 + 1, (M - 1)/32 + 1);
13        foo_kernel <<< gd, bd >>>(a_d, b_d, M, N);
14    }
```

   a. What is the number of threads per block?
   b. What is the number of threads in the grid?
   c. What is the number of blocks in the grid?
   d. What is the number of threads that execute the code on line 05?

4. Consider a two-dimensional matrix with a width of 400 and a height of 500. The matrix is stored as a one-dimensional array. Specify the array index of the matrix element at row 20 and column 10:
   a. If the matrix is stored in row major order.
   b. If the matrix is stored in column major order.

5. Consider a three-dimensional tensor with a width of 400, a height of 500, and a depth of 300. The tensor is stored as a one-dimensional array in row major order. Specify the array index of the tensor element at $x = 10$, $y = 20$, and $z = 5$?

This page intentionally left blank

CHAPTER

Compute architecture and scheduling

4

CONTENTS

4.1 Architecture of a modern GPU 68
4.2 Thread block scheduling 69
4.3 Synchronization and transparent scalability 70
4.4 Warps and SIMD hardware 74
4.5 Control divergence 79
4.6 Warp scheduling and latency tolerance 82
4.7 Resource partitioning and occupancy 85
4.8 Querying device properties 87
4.9 Summary 89
4.10 Exercises 90
References 92

In Chapter 1, we saw that CPUs are designed to minimize the latency of instruction execution and that GPUs are designed to maximize the throughput of executing instructions. In Chapter 2 and Chapter 3, we learned the core features of CUDA programming for creating and calling kernels to launch and execute threads. In the next three chapters, we discuss the architecture of modern GPUs, both the compute architecture and the memory architecture, and the performance optimization techniques stemming from the understanding of this architecture.

This chapter presents several aspects of the GPU compute architecture that are essential for CUDA C++ programmers to understand and reason about the performance behavior of their kernel code. We start by showing a high-level simplified view of the compute architecture and explore the concepts of flexible resource assignment, scheduling of thread blocks, and occupancy. We then advance into thread scheduling, latency tolerance, control divergence, and synchronization. We will finish the chapter with a description of the API functions that can be used to query the resources available in the GPU and the tools to help estimate the occupancy of the GPU when executing a kernel.

In the following two chapters, we present the core concepts and programming considerations of the GPU memory architecture. In particular, Chapter 5 focuses on the on-chip memory architecture, and Chapter 6 briefly covers the off-chip memory architecture then elaborates on various performance considerations of the GPU architecture as a whole. A CUDA C++ programmer who masters these concepts is well equipped to write and to understand high-performance parallel kernels.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00013-5
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

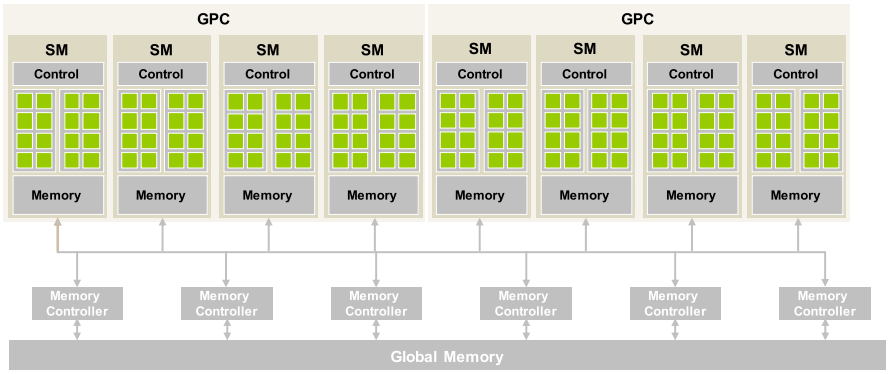


FIGURE 4.1

Architecture of a CUDA-capable GPU.

4.1 Architecture of a modern GPU

Fig. 4.1 shows a high-level CUDA C++ programmer's view of the architecture of a typical CUDA-capable GPU. The GPU is organized into an array of highly threaded Streaming Multiprocessors (SMs). Each SM has several processing units called *streaming processors*[1] that share control logic and memory resources. The streaming processors are shown as small tiles inside SMs in Fig. 4.1. For example, the Hopper H100 GPU has 132 SMs with 128 streaming processors each, which totals 16,896 streaming processors in the entire GPU. Starting with the Hopper architecture, the SMs of a GPU are grouped into GPU Processing Clusters (GPCs). For example, the 132 SMs of the Hopper H100 GPU are arranged into 8 GPCs.

The memory resources of SMs consist of different on-chip memory structures collectively labeled as *Memory* in Fig. 4.1. These on-chip memory structures will be the topic of Chapter 5. GPUs also come with gigabytes of off-chip device memory, referred to as *Global Memory* in Fig. 4.1. The global memory is accessed through multiple memory controllers or memory channels, labeled as *Memory Controller* in Fig. 4.1, that collectively support high data access bandwidth. While older GPUs used GDDR (Graphics Double Data Rate) SDRAM (Synchronous Dynamic Random Access Memory), GPUs starting with NVIDIA's Pascal architecture may use HBM (High-Bandwidth Memory), HBM2, HBM2E, or HBM3, which consist of DRAM modules tightly integrated with the GPU in the same package. For brevity, we will broadly refer to all these types of memory as DRAM for the rest of the book. We will discuss the most important concepts involved in accessing GPU DRAMs in Chapter 6.

[1] Streaming processors are sometimes referred to as *CUDA cores*. However, to avoid potential confusion with CPU cores, we will only use the term streaming processors in our discussions.

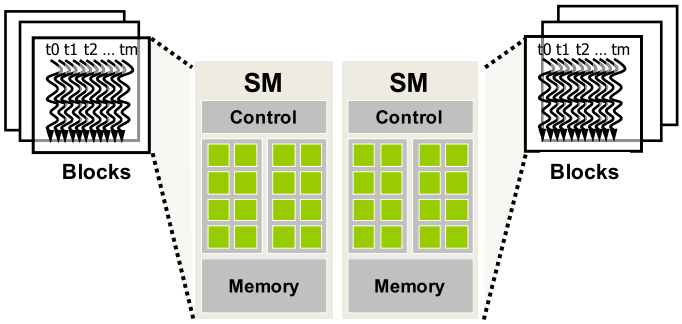


FIGURE 4.2

Thread block assignment to Streaming Multiprocessors (SMs).

4.2 Thread block scheduling

When a kernel is called, the CUDA run-time system launches a grid of threads that execute the kernel code. Although the GPUs have massive amount of execution resources, the grid of threads launched to execute a kernel can easily require resources far exceeding the available GPU execution resources. A thread block scheduling mechanism is needed to assign threads to execution resources as the resources become available. In CUDA, threads are assigned to SMs on a block-by-block basis. That is, all threads in a block are simultaneously assigned to the same SM.

Fig. 4.2 illustrates the assignment of blocks to SMs. Multiple blocks can be simultaneously assigned to the same SM as permitted by the available SM resources. For example, in Fig. 4.2, three blocks are assigned to each SM. However, blocks need to reserve hardware resources to execute, so only a limited number of blocks can be simultaneously assigned to a given SM. The limit on the number of blocks depends on a variety of factors that are discussed in Section 4.6.

With a limited number of SMs and a limited number of blocks that can be simultaneously assigned to each SM, there is a limit on the total number of blocks that can be simultaneously executing in a CUDA device. However, grids usually contain many more blocks than this number. To ensure that all blocks in a grid get executed, the run-time system maintains a list of blocks that need to execute and assigns new blocks to SMs when previously assigned blocks complete execution.

The assignment of threads to SMs on a block-by-block basis guarantees that threads in the same block execute on the same SM and at the same time. This guarantee makes it possible for threads in the same block to interact with each other in ways that threads across different blocks cannot. Such interactions include a block-wide *barrier synchronization*, which is discussed in Section 4.3. They also include data exchange through a low-latency SRAM (Static Random Access Memory) called *shared memory* that resides in the SM, which is discussed in Chapter 5.

With the Hopper architecture, CUDA introduces an optional hierarchy level called *thread block cluster*, which is a group of thread blocks. Similarly to threads in the

same block being assigned onto the same SM, blocks in the same thread block cluster are co-scheduled onto the same GPC. Threads in the same cluster can synchronize via the corresponding API, which is introduced in Section 4.3. They also have access to *distributed shared memory* which is composed of the shared memory of all blocks participating in a thread block cluster. We will learn more about distributed shared memory in Chapter 5.

4.3 Synchronization and transparent scalability

CUDA allows threads in the same block to coordinate their activities using the block-wide barrier synchronization function `__syncthreads()`. Note that "__" consists of two "_" characters, which is the convention for indicating intrinsic functions in CUDA (see Sidebar: Intrinsic Functions). When a thread calls `__syncthreads()`, it will be held at the program location of the call until every thread in the same block reaches that location. A thread that calls a synchronization function is said to *arrive at* the barrier that corresponds to the program location. A barrier ensures that all threads in a block have completed a phase of their execution before any of them can move on to the next phase. Only after all the threads in the block have arrived at the barrier are they *released from the barrier*, which allows them to execute beyond the barrier's program location.

Intrinsic Functions

Modern processors often offer special instructions that either perform critical functionality or substantial performance enhancement. These instructions are typically exposed to the programmers as *intrinsic functions*, or simply *intrinsics*. From the programmer's perspective, these are library functions. However, they are treated in a special way by compilers; each such call is translated into the corresponding special instruction. There is typically no function call in the final code, just the special instructions in line with the user code. All major modern compilers, such as the GNU Compiler Collection (gcc), Intel C Compiler, and Clang/LLVM C Compiler support intrinsics.

At the thread block cluster level, CUDA provides a mechanism to perform a cluster-wide synchronization using the Cooperative Groups API [1]. All threads in a thread block cluster can synchronize at a program location to ensure that all threads in the same thread block cluster have arrived at the barrier and that all their distributed shared memory operations are completed before they can be released from the barrier. The collection of threads involved in a barrier is called the *scope* of the barrier.

At the grid-level, threads across the entire grid can perform a grid-wide barrier synchronization also through the Cooperative Groups API [1]. However, the grid-wide barrier synchronization is more heavyweight than the block-wide and cluster-wide ones, and there are several important restrictions that must be obeyed to ensure

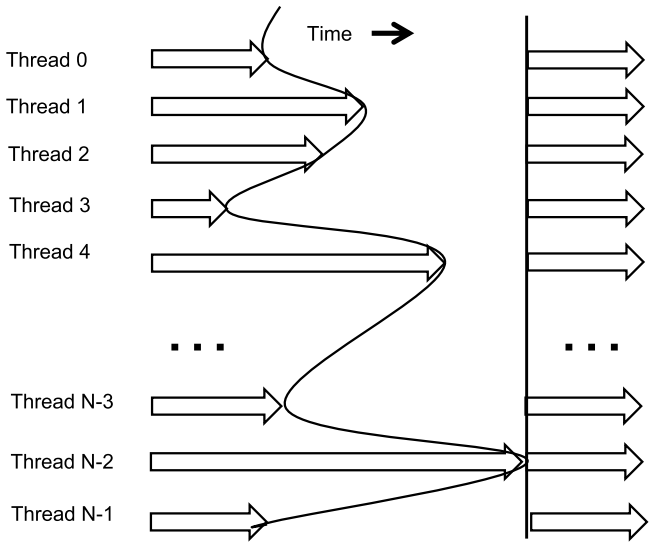


FIGURE 4.3

An example execution timing of barrier synchronization. The arrows represent execution activities over time. The vertical curve marks the time when each thread executes the `__syncthreads` statement. The empty space to the right of the vertical curve depicts the time that each thread waits for all threads to complete. The vertical line marks the time that the last thread executes the `__syncthreads` statement, after which all threads are allowed to proceed to execute the statements after the `__syncthreads` statement.

that all threads in the grid are indeed simultaneously executing and can reach the barrier. We will see an example of using the Cooperative Groups API to perform grid-wide barrier synchronization in Chapter 18. For the rest of this section, we will focus our discussion on block-wide barrier synchronizations.

Barrier synchronization is a simple and popular method for coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities involving multiple persons. For example, assume that four friends go to a shopping mall in a car. They can all go to different stores to shop for their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit all the stores of interest since not all of them are interested in the same stores. However, barrier synchronization is needed before they leave the mall. They must wait until all four friends have returned to the car before they can leave. The ones who finish earlier than the others must wait for those who finish later. Without the barrier synchronization, one or more persons can be left in the mall when the car leaves, which can seriously damage their friendship!

Fig. 4.3 illustrates the execution of barrier synchronization. There are N threads involved. Time goes from left to right. Some of the threads arrive at the barrier synchronization program location early and some of them much later. The ones that

```
1    void incorrect_barrier_example(int n) {
2         ...
3         if (threadIdx.x % 2 == 0) {
4              ...
5              __syncthreads();
6         }
7         else {
8              ...
9              __syncthreads();
10        }
11   }
```

FIGURE 4.4

An incorrect use of `__syncthreads()`.

arrive at the barrier early wait for those who arrive late. When the latest one arrives at the barrier, everyone can be released from the barrier and continue their execution. With barrier synchronization, "no one is left behind."

In CUDA, if a `__syncthreads()` statement is present, it must be executed by all threads in a block. When a `__syncthreads()` statement is placed in an if-statement, either all threads in a block execute the path that includes the `__syncthreads()` or none of them does. For an if-the-else statement, if each path has a `__syncthreads()` statement, either all threads in a block execute the then-path or all of them execute the else-path. The two `__syncthreads()` are different barrier synchronization points. Fig. 4.4 shows an incorrect usage of `__syncthreads()` where two `__syncthreads()` are used in the if-statement starting in line 04. All threads with even `threadIdx.x` values execute the then-path while the remaining threads execute the else-path. The `__syncthreads()` calls at line 06 and line 10 define two different barriers. Since not all threads in a block are guaranteed to execute either of the barriers, the code violates the rules for using `__syncthreads()` and will result in undefined execution behavior. In general, incorrect usage of barrier synchronization can result in incorrect results, or in threads waiting for each other forever, which is referred to as a *deadlock*. It is the responsibility of the programmer to avoid such inappropriate use of barrier synchronization.

Barrier synchronization imposes execution constraints on threads within its scope. These threads should execute in close time proximity with each other to avoid excessively long waiting times. More importantly, the system needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier. Otherwise, a thread that never arrives at the barrier can cause a deadlock. The CUDA run-time system satisfies this constraint by assigning execution resources to all threads in a block as a unit, as we saw in Section 4.2. Not only do all threads in a block have to be assigned to the same SM, but also, they need to be assigned to that SM simultaneously. That is, a block can only begin execution when the run-time system has secured all the resources needed by all threads

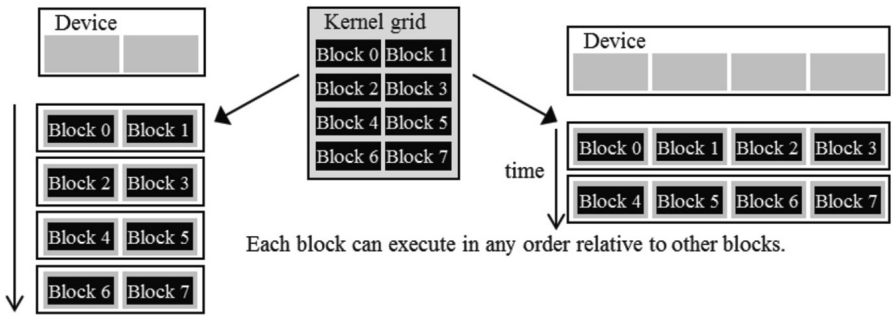


FIGURE 4.5

Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs.

in the block to complete execution. This ensures the time proximity of all threads in a block and prevents excessive or even indefinite waiting time during barrier synchronization.

This leads us to the importance of having limited scope in the use of CUDA barrier synchronization. If the use of barrier synchronization is constrained to the thread block scope, the CUDA run-time system can execute blocks in any order relative to each other. This flexibility enables scalable implementations as shown in Fig. 4.5. Time in the figure progresses from top to bottom. In a low-cost system with only a few execution resources, i.e. SMs, one can execute a small number of blocks at the same time, portrayed as executing two blocks a time on the left-hand side of Fig. 4.5. In a higher-end implementation with more execution resources, one can execute many blocks at the same time, portrayed as executing four blocks at a time on the right-hand side of Fig. 4.5. A high-end GPU today can execute hundreds or even thousands of blocks simultaneously.

A group of thread blocks executing simultaneously is usually referred to as a *wave*. The ratio of the total number of thread blocks in the grid to the number that can execute simultaneously is referred to as the number of waves. For example, in Fig. 4.5, the example on the left is said to have four waves, whereas the example on the right is said to have two waves. It is desirable to have a large number of waves when the load across blocks is imbalanced to provide enough opportunities for the hardware to balance the load across SMs. On the other hand, in cases where the load across blocks is balanced, a small number of waves is acceptable and sometimes desirable as we will see in Chapter 6 when we discuss thread coarsening.

If using a small number of waves, one has to be careful to avoid situations where the last wave is not complete, i.e., consists of only a small number of blocks, and thus does not fully utilize the hardware. For example, if a grid with 660 blocks runs on a GPU that can execute 264 blocks simultaneously, it will have $660/264 = 2.5$ waves. If the blocks are fairly load balanced, then the first wave of 264 blocks will execute, followed by the second wave of 264 blocks, followed by a partial wave of

132 blocks. In this case, the partial wave of 132 blocks will underutilize the GPU resources, a phenomenon sometimes referred to as a *tail effect*. To avoid tail effects, it is desirable to configure grids with a number of blocks that is a multiple of the number that can run simultaneously, which results in an integer number of waves. This number depends on the amount of hardware resources available in the GPU.

The ability to execute the same application code on a wide range of GPUs with different amounts of resources allows the production of a wide range of GPUs according to the cost, power, and performance requirements of different market segments. For example, a mobile processor may execute an application slowly but at extremely low power consumption and a desktop processor may execute the same application at a higher speed while consuming more power. Both execute the same application program with no change to the code. The ability to execute the same application code on different hardware with different amounts of execution resources is referred to as *transparent scalability*, which reduces the burden on application developers and improves the usability of applications.

4.4 Warps and SIMD hardware

We have seen that when blocks are in different barrier synchronization scopes, they can execute in any order relative to each other, which allows for transparent scalability across different devices. However, we did not say much about the execution timing of threads within each block. Conceptually, one should assume that threads in a block can execute in any order with respect to each other. In algorithms with phases, barrier synchronizations should be used whenever we want to ensure that all threads in the barrier's scope have completed a previous phase of their execution before any of them start the next phase. The correctness of executing a kernel should not depend on any assumption that certain threads will execute in synchrony with each other without use of barrier synchronizations.

Thread scheduling in CUDA GPUs is a hardware implementation concept and thus must be discussed in the context of specific hardware implementations. In most implementations to date, once a block is assigned to a streaming multiprocessor, it is further divided into 32-thread units called *warps*. The size of warps is implementation specific and can vary in future generations of GPUs. Knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices.

A warp is the unit of thread scheduling in SMs. Fig. 4.6 shows the division of blocks into warps in an implementation. In this example, there are three blocks – Block 1, Block 2, and Block 3 – all assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes. Each warp consists of 32 threads of consecutive `threadIdx` values: threads 0 through 31 form the first warp, threads 32 through 63 form the second warp, and so on. We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM. In this example, if each block has 256 threads, we can determine that each

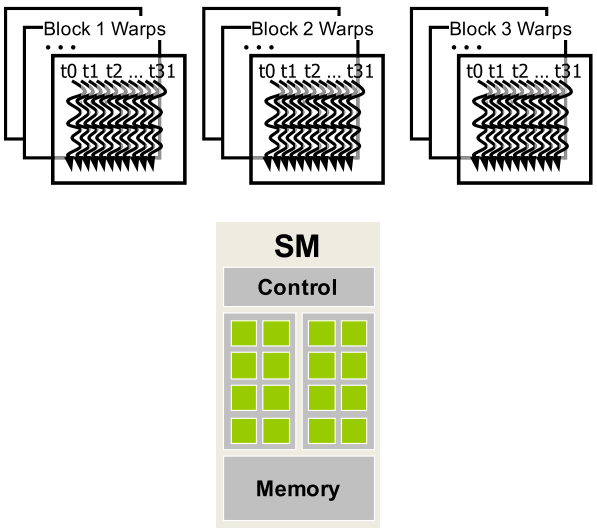


FIGURE 4.6

Blocks are partitioned into warps for thread scheduling.

block has 256/32 or 8 warps. With three blocks in the SM, we have $8 \times 3 = 24$ warps in the SM.

Blocks are partitioned into warps based on thread indices. If a block is organized into a one-dimensional array, i.e., only `threadIdx.x` is used, the partition is straightforward. The `threadIdx.x` values within a warp are consecutive and increasing. For a warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63, and so on. In general, warp n starts with thread $32 \times n$ and ends with thread $32 \times (n + 1) - 1$. For a block whose size is not a multiple of 32, the last warp will be padded with inactive threads to fill up the 32 thread positions. For example, if a block has 48 threads, it will be partitioned into two warps, and the second warp will be padded with 16 inactive threads.

For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linearized row-major layout before partitioning into warps. The linear layout is determined by placing the rows with larger y and z coordinates after those with lower ones. That is, if a block consists of two dimensions of threads, one will form the linear layout by placing all threads whose `threadIdx.y` is 1 after those whose `threadIdx.y` is 0. Threads whose `threadIdx.y` is 2 will be placed after those whose `threadIdx.y` is 1, and so on. Threads with the same `threadIdx.y` value are placed in consecutive positions in increasing `threadIdx.x` order.

Fig. 4.7 shows an example of placing threads of a two-dimensional block into linear layout. The upper part shows the two-dimensional view of the block. The reader should recognize that the similarity with the row-major layout of two-dimensional arrays. Each thread is shown as $T_{y,x}$, x being `threadIdx.x` and y being `threadIdx.y`. The

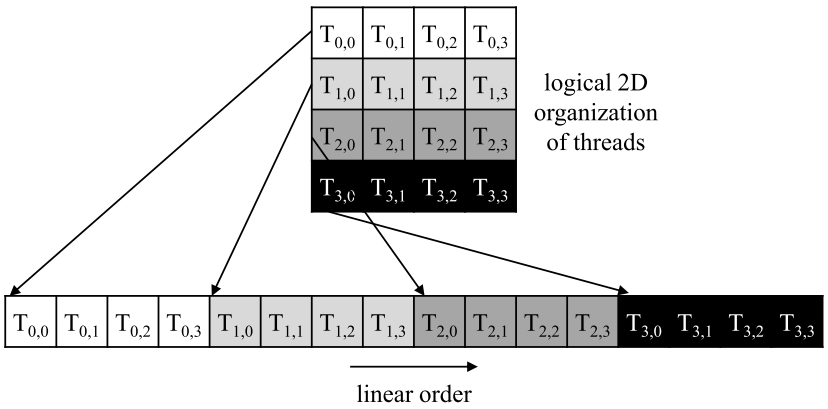


FIGURE 4.7

Placing 2D threads into linear layout.

lower part of Fig. 4.7 shows the linearized view of the block. The first four threads are those threads whose `threadIdx.y` value is 0; they are ordered with increasing `threadIdx.x` values. The next four threads are those threads whose `threadIdx.y` value is 1. They are also placed with increasing `threadIdx.x` values. For this example, all 16 threads form half a warp. The warp will be padded with another 16 threads to complete a 32-thread warp. Imagine a two-dimensional block with 8×8 threads. The 64 threads will form two warps. The first warp starts from $T_{0,0}$ and ends with $T_{3,7}$. The second warp starts with $T_{4,0}$ and ends with $T_{7,7}$. It would be useful for the reader to draw out the picture as an exercise.

For a three-dimensional block, we first place all threads whose `threadIdx.z` value is 0 into the linear order. Among these threads, they are treated as a two-dimensional block as shown in Fig. 4.7. All threads whose `threadIdx.z` value is 1 will then be placed into the linear order, and so on. For example, for a three-dimensional $2 \times 8 \times 4$ block (four in the x dimension, eight in the y dimension, and two in the z dimension), the 64 threads will be partitioned into two warps, with $T_{0,0,0}$ through $T_{0,7,3}$ in the first warp and $T_{1,0,0}$ through $T_{1,7,3}$ in the second warp.

An SM is designed to execute all threads in a warp following the Single Instruction, Multiple Data (SIMD) model. That is, at any instant in time, one instruction is fetched and executed by all threads in the warp (see "Warps and SIMD Hardware" sidebar). For this purpose, an SM is organized into multiple *processing blocks* and the warps on an SM are distributed across the processing blocks. Fig. 4.8 shows a toy example where an SM consists of two processing blocks, and every processing block comprises eight streaming processors with a shared instruction fetch/dispatch unit. As a real example, the Hopper H100 SM, which has 128 streaming processors, is organized into four processing blocks with 32 streaming processors each.

Threads in the same warp are assigned to the same processing block which fetches the instruction for the warp and executes it for all threads in the warp. These threads

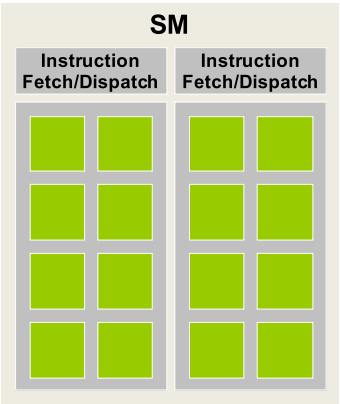


FIGURE 4.8

SMs are organized into processing blocks for SIMD execution. In this toy example, every eight streaming processors form a processing block. The SM consists of two processing blocks.

apply the same instruction to their portions of the data. Because the CUDA thread scheduling mechanism maps warps to the processing blocks and effectively restricts all threads in a warp to execute the same instruction at any point in time, the execution behavior of a warp is often referred to as Single Instruction, Multiple Thread (SIMT). An important advantage of SIMT is that a programmer can write kernel code for scalar threads, which are later grouped into warps by the SM and transparently use the SIMD hardware. This is in contrast to traditional CPU SIMD programming where the programmer/compiler needs to use API functions to use the SIMD hardware within each thread, which adds another level of programming complexity.[2]

The advantage of SIMD is that the cost of the control hardware, such as the instruction fetch/dispatch unit, is shared across many execution units. This design choice allows for a smaller percentage of the hardware to be dedicated to control and a larger percentage to be dedicated for increasing arithmetic throughput. In the foreseeable future, we expect that warp partitioning will remain as a popular implementation technique. However, the size of warps can vary from implementation to implementation. Up to this point in time, all CUDA devices have used similar warp configurations where each warp consists of 32 threads.

Because threads in the same warp have a special relationship in the way that they are scheduled, CUDA provides a collection of *warp-level primitives*, i.e. API functions, that expose efficient data exchange and synchronization mechanisms across threads in a warp. Programmers can use these warp-level primitives to improve the speed and efficiency of computations that involve collaboration between threads.

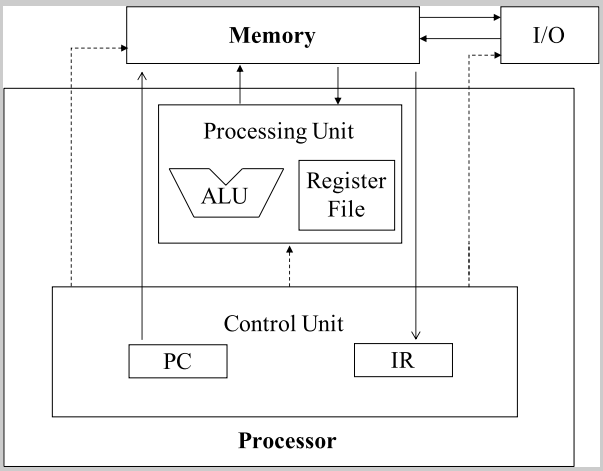
[2] Even though SIMT makes programming easier than SIMD, programmers should avoid *control divergence*, which has a negative impact on performance, as Section 4.5 explains.

These warp-level primitives allow for a warp-centric programming style, which treats warps as an extra level of the software hierarchy (between blocks and threads), and can result in significant performance benefits. Chapter 10, Chapter 11, and Chapter 12 introduce some relevant warp-level primitives.

Warps and SIMD Hardware

In his seminal 1945 report, John von Neumann described a model for building electronic computers, which is based on the design of the pioneering EDVAC computer. This model, now commonly referred to as the "von Neumann Model," has been the foundational blueprint for virtually all modern computers.

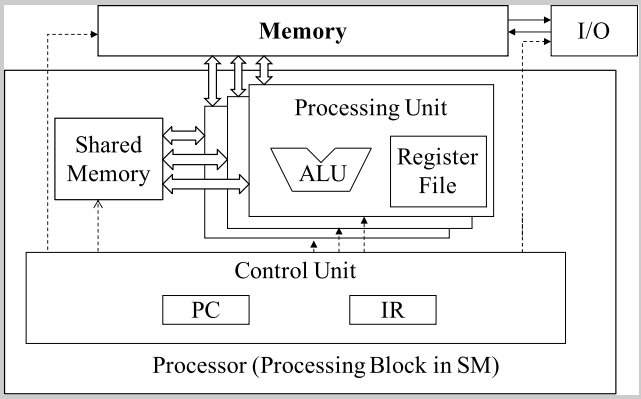
The von Neumann Model is illustrated in the following figure. The computer has an I/O (input/output) that allows both programs and data to be provided to and generated from the system. To execute a program, the computer first inputs the program and its data into the Memory.



The program consists of a collection of instructions. The Control Unit maintains a Program Counter (PC), which contains the memory address of the next instruction to be executed. In each "instruction cycle," the Control Unit uses the PC to fetch an instruction into the Instruction Register (IR). The instruction bits are then examined to determine the action to be taken by all components of the computer. This is the reason why the model is also called the "stored program" model, which means that a user can change the behavior of a computer by storing a different program into its memory.

The motivation for executing threads as warps is illustrated in the following modified von Neumann model that is adapted to reflect a GPU design. The processor, which corresponds to a processing block in Fig. 4.8, has only one control unit that fetches and dispatches instructions. The same control signals (arrows

that go from the Control Unit to the Processing Units in Fig. 4.8) go to multiple processing units that each correspond to a streaming processor in the SM, each of which executes one of the threads in a warp.



Since all processing units are controlled by the same instruction in the Instruction Register (IR) of the Control Unit, their execution differences are due to the different data operand values in the register files. This is called Single-Instruction-Multiple-Data (SIMD) in processor design. For example, although all processing units (cores) are controlled by an instruction, such as add r1, r2, r3, the contents of r2 and r3 are different in different processing units. SIMD is one of the four classes of computer architectures that Flynn's taxonomy [2] establishes. The other three are Single-Instruction-Single-Data (SISD), Multiple-Instruction-Single-Data (MISD), and Multiple-Instruction-Multiple-Data (MIMD).

Control units in modern processors are quite complex, including sophisticated logic for fetching instructions and access ports to the instruction cache. Having multiple processing units to share a control unit can result in significant reduction in hardware manufacturing cost and power consumption.

4.5 Control divergence

SIMD execution works well when all threads within a warp follow the same path of execution, more formally referred to as *control flow*, when working on their data. For example, for an if-else construct, the execution works well when either all threads in a warp execute the then-path or all execute the else-path. However, when threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these paths, one pass for each path. For example, for an if-else construct, if

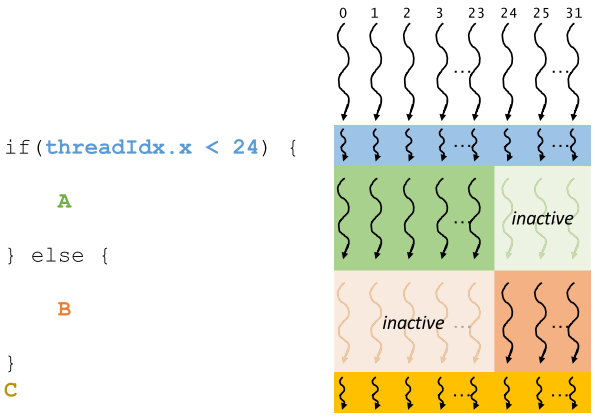


FIGURE 4.9

Example of a warp diverging at an if-else statement.

some threads in a warp follow the then-path while others follow the else path, the hardware will take two passes. One pass executes the threads that follow the then-path and the other executes the threads that follow the else-path. During each pass through a path, the threads that follow the other path are not allowed to take effect.

When threads in the same warp follow different control flow paths, we say that these threads exhibit *control divergence*, i.e., they diverge in their execution control flow paths. The multi-pass approach to divergent warp execution extends the SIMD hardware's ability to implement the full semantics of CUDA threads. While the hardware executes the same instruction for all threads in a warp, it selectively lets these threads take effect in only the pass that corresponds to the path that they took, allowing every thread to appear to take its own control flow path. This preserves the independence of threads while taking advantage of the reduced cost of SIMD hardware. The cost of divergence, however, is the extra passes the hardware needs to take in order to allow different threads in a warp to make their own decisions, as well as the execution resources that are consumed by the inactive threads in each pass.

Fig. 4.9 shows an example of how a warp would execute a divergent if-else statement. In this example, when the warp consisting of threads 0-31 arrives at the if-else statement, threads 0-23 take the then-path while threads 24-31 take the else-path. In this case, the warp will do a pass through the code where threads 0-23 execute `A` while threads 24-31 are inactive. The warp will also do another pass through the code where threads 24-31 execute `B` while threads 0-23 are inactive. The threads in the warp may then reconverge and execute `C`.

In the Pascal architecture and prior architectures, these passes are executed sequentially, meaning that one pass is executed to completion followed by the other pass. From the Volta architecture onwards, the passes may be executed concurrently, meaning that the execution of one pass may be interleaved with the execution of another pass. This feature is referred to *independent thread scheduling*. Interested

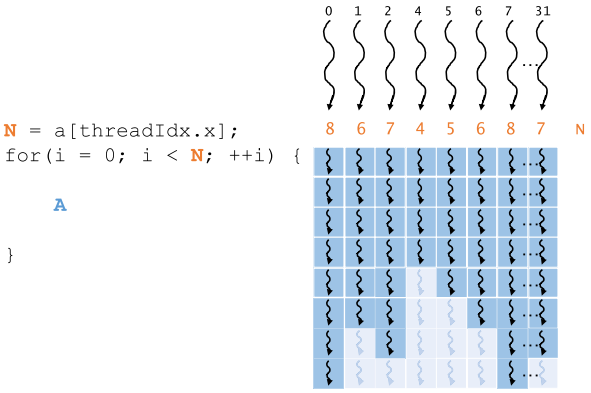


FIGURE 4.10

Example of a warp diverging at a for-loop.

readers are referred to the whitepaper on the Volta V100 architecture [3] for details. An important implication of independent thread scheduling is that one cannot assume that the threads of the warp reconverge after the divergent execution paths are executed. Therefore, if all threads in a warp must complete a phase of their execution before any of them can move on, one must use a warp-level barrier synchronization primitive such as `__syncwarp()` to ensure correctness.

Divergence also can arise in other control flow constructs. Fig. 4.10 shows an example of how a warp would execute a divergent for-loop. In this example, each thread executes a different number of loop iterations, which vary between four and eight. For the first four iterations, all threads are active and execute `A`. For the fifth iteration, some threads execute `A`, while others, e.g. thread 4, are inactive because they have completed their iterations. In the eight iteration, only threads 1 and 7 (among the threads shown) execute `A` while the others are inactive.

One can determine if a control construct can result in thread divergence by inspecting its decision condition. If the decision condition is based on `threadIdx` values, the control statement can potentially cause thread divergence. For example, the statement `if(threadIdx.x > 2) { ... }` causes the threads in the first warp of a block to follow two divergent control flow paths. Threads 0, 1, and 2 follow a different path than threads 3, 4, 5, etc. Similarly, a loop can cause thread divergence if its loop condition is based on thread index values.

A prevalent reason for using a control construct with thread control divergence is handling boundary conditions when mapping threads to data. This is usually because the total number of threads needs to be a multiple of the thread block size whereas the size of the data can be an arbitrary number. Starting with our vector addition kernel in Chapter 2, we had an `if(i < n)` statement in `addVecKernel`. This is because not all vector lengths can be expressed as multiples of the block size. For example, let's assume that the vector length is 1003 and we picked 64 as block size. One would need to launch 16 thread blocks to process all the 1003 vector elements. However,

the 16 thread blocks would have 1024 threads. We need to disable the last 21 threads in thread block 15 from doing work not expected/allowed by the original program. Keep in mind that these 16 blocks are partitioned into 32 warps. Only the last warp (i.e., second warp in the last block) will have control divergence.

Note that the performance impact of boundary condition control divergence decreases as the size of the vectors being processed increases. For a vector length of 100, one of the four warps will have control divergence, which can have significant impact on performance. For a vector size of 1000, only one out of the 32 warps will have control divergence. That is, control divergence will affect only about 3% of the execution time. Even if it doubles the execution time of the warp, the net impact to the total execution time will be about 3%. Obviously, if the vector length is 10,000 or more, only one of the 313 warps will have control divergence. The impact of control divergence will be much less than 1%!

For two-dimensional data, such as the color-to-grayscale conversion example in Chapter 3, if-statements are also used to handle the boundary conditions for threads that operate at the edge of the data. In Fig. 3.2, to process the 62×76 image, we used $20 = 4 \times 5$ two-dimensional blocks that consist of 16×16 threads each. Each block will be partitioned into 8 warps, each one consists of two rows of a block. There are a total 160 warps (8 warps per block) involved. To analyze the impact of control divergence, refer to Fig. 3.5. None of the warps in the 12 blocks in region 1 will have control divergence. There are $12 \times 8 = 96$ warps in region 1. For region 2, all the 24 warps will have control divergence. For region 3, all the bottom warps are mapped to data that are completely outside the image. As result, none of them will pass the if condition. The reader should verify that these warps would have had control divergence if the picture had an odd number of pixels in the vertical dimension. In region 4, the first seven warps will have control divergence, but the last warp will not. All in all, 31 out of the 160 warps will have control divergence.

Once again, the performance impact of control divergence decreases as the number of pixels in the horizontal dimension increases. For example, if we process a 200×150 picture with 16×16 blocks, there will be a total $130 = 13 \times 10$ thread blocks or 1040 warps. The number of warps in regions 1 through 4 will be 864 ($12 \times 9 \times 8$), 72 (9×8), 96 (12×8), and 8 (1×8). Only 80 of these warps will have control divergence. Thus, the performance impact of control divergence will be less than 8%. Obviously, if we process a realistic picture with more than 1000 pixels in the horizontal dimension, the performance impact of control divergence will be less than 2%.

4.6 Warp scheduling and latency tolerance

When threads are assigned to SMs, there are usually more threads assigned to an SM than streaming processors in the SM. That is, each SM has only enough execution units to execute a subset of all the threads assigned to it at any point in time. In earlier GPU designs, each SM can execute only one instruction for a single warp at any given

instant. In more recent designs, each SM can execute instructions for a small number of warps at any given point in time. That is, each SM contains multiple processing blocks as illustrated in Fig. 4.8. In either case, at any point in time, the hardware can execute instructions only for a subset of all warps assigned to the SM. A legitimate question is why we need to have so many warps assigned to an SM if it can only execute a subset of them at any instant? The answer is that this is how GPUs tolerate long-latency operations such as global memory accesses.

When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Instead, another resident warp that is no longer waiting for results of previous instructions will be selected for execution. Warps that are not waiting for the results of previous instructions are ready for execution. If more than one warp is ready for execution, a priority logic is used to select one for execution. The ability for each SM to select different ready warps for execution at each instant is called *fine-grained multithreading* and allows SMs to use the latency time of instructions from some threads to execute instructions from other threads. As a result, fine-grained multithreaded execution of warps can achieve high utilization of execution units in spite of long latency of instructions, which is commonly referred to as "latency tolerance" or "latency hiding" (see "Latency Tolerance" sidebar).

Latency Tolerance

Latency tolerance is needed in many everyday situations. For example, in post offices, each person trying to ship a package should ideally have filled out all the forms and labels before going to the service counter. However, as we all have experienced, some people wait for the service desk clerk to tell them which form to fill out and how to fill out the form.

When there is a long line in front of the service desk, it is important to maximize the productivity of the service clerks. Letting a person fill out the form in front of the clerk while everyone waits is not a good approach. The clerk should be helping the next customers who are waiting in line while the person fills out the form. These other customers are "ready to go" and should not be blocked by the customer who needs more time to fill out a form.

This is why a good clerk would politely ask the first customer to step aside to fill out the form while he/she can serve other customers. In most cases, the first customer will be served as soon as he finishes the form and the clerk finishes serving the current customer, instead of going to the end of the line.

We can think of these post office customers as warps and the clerk as a hardware execution unit. The customer that needs to fill out the form corresponds to a warp whose continued execution is dependent on a long latency operation.

Note that warp scheduling is also used for tolerating other types of operation latencies such as pipelined floating-point arithmetic operations and conditional

branches. With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware while instructions of some warps wait for the results of these long latency operations. The selection of warps ready for execution does not introduce any idle or wasted time into the execution timeline, which is referred to as *zero-overhead thread scheduling* (see "Threads, Context-switching, and Zero-overhead Scheduling" sidebar). With warp scheduling, the long waiting time of warp instructions is "hidden" by executing instructions from other warps. This ability to tolerate long operation latencies is the main reason why GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as CPUs. As a result, GPUs can dedicate more chip area to floating-point execution units and memory access channels.

Threads, Context-switching, and Zero-overhead Scheduling

Based on the von Neumann model, we are ready to more deeply understand how threads are implemented. A thread in modern computers is a program and the state of executing the program on a von Neumann Processor. Recall that a thread consists of the code of a program, the instruction in the code that is being executed, and values of its variables and data structures.

In a computer based on the von Neumann model, the code of the program is stored in the memory. The PC keeps track of the address of the instruction of the program that is being executed. The IR holds the instruction that is being executed. The registers and memory hold the values of the variables and data structures.

Modern processors are designed to allow context-switching, where multiple threads can time-share a processor by taking turns to make progress. By carefully saving and restoring the PC value and the contents of registers and memory, we can suspend the execution of a thread and correctly resume the execution of the thread later. However, saving and restoring register contents during context-switching in these processors can incur significant overhead in terms of added execution time.

Zero-overhead scheduling refers to the GPU's ability to put a warp that needs to wait for a long-latency instruction result to sleep and activate a warp that is ready to go without introducing any extra idle cycles in the processing units. Context switching on traditional CPUs incurs such idle cycles because switching the execution from one thread to another requires saving the execution state (such as register contents of the out-going thread) to memory and loading the execution state of the incoming thread from memory. GPU SMs achieves zero-overhead scheduling by holding all the execution states for the assigned warps in the hardware registers so there is no need to save and restore states when switching from one warp to another.

For latency tolerance to be effective, it is desirable for an SM to have many more threads assigned to it than what can be simultaneously supported with its execution

resources to maximize the chance of finding a warp that is ready to execute at any point in time. For example, in a Hopper H100 GPU, an SM has 128 streaming processors but can have up to 2048 threads assigned to it at the same time. Thus, the SM can have up to 16 times more threads assigned to it than its streaming processors can support at any given clock cycle. This oversubscription of threads to SMs is essential for latency tolerance. It increases the chance of finding another warp to execute when a currently executing warp encounters a long-latency operation.

4.7 Resource partitioning and occupancy

We have seen that it is desirable to assign many warps to an SM in order to tolerate high operation latencies. However, it may not always be possible to assign to the SM the maximum number of warps that the SM supports. The ratio of the number warps assigned to an SM to the maximum number it supports is referred to as *occupancy*. To understand what may prevent an SM from reaching maximum occupancy, it is important first to understand how SM resources are partitioned.

Besides the streaming processors, the execution resources in an SM include registers, shared memory (discussed in Chapter 5), thread block slots, and thread slots. These resources are dynamically partitioned across threads to support their execution. For example, a Hopper H100 GPU can support a maximum of 32 blocks per SM, 64 warps (2048 threads) per SM, and 1024 threads per block. If a grid is launched with a block size of 1024 threads (the maximum allowed), the 2048 thread slots in each SM are partitioned and assigned to 2 blocks. In this case, each SM can accommodate up to 2 blocks. Similarly, if a grid is launched with a block size of 512, 256, 128, or 64 threads, the 2048 thread slots are partitioned and assigned to 4, 8, 16, or 32 blocks, respectively.

This ability to dynamically partition thread slots among blocks makes SMs versatile. They can either execute many blocks each having few threads or execute few blocks each having many threads. This dynamic partitioning can be contrasted with a fixed partitioning method where each block would receive a fixed amount of resources regardless of its real needs. Fixed partitioning results in wasted thread slots when each block demands fewer threads than the fixed partition supports and fails to support blocks that require more thread slots than that.

Dynamic partitioning of resources can lead to subtle interactions between resource limitations, which can cause underutilization of resources. Such interactions can occur between block slots and thread slots. In the example of the Hopper H100, varying the block size from 1024 to 64 would result in 2 to 32 blocks per SM. In all these cases, the total number of threads assigned to the SM is 2048 which maximizes occupancy. Consider, however, the case where each block has only 32 threads. In this case, filling 2048 thread slots would require 64 blocks. However, each Hopper SM has only 32 block slots, and can only accommodate 32 blocks at any point in time. This means that only 1024 of the thread slots will be utilized, i.e., 32 blocks with 32 threads each. The occupancy in this case is (1024 assigned threads)/(2048 maximum

threads) = 50%. Therefore, to fully utilize the thread slots and achieve maximum occupancy, one needs at least 64 threads in each block.

Another situation that could negatively impact occupancy is when the number of thread slots is not divisible by the block size. In the example of the Hopper H100, we saw that up to 2048 threads per SM can be supported. However, if a block size of 768 is selected, then the SM will only be able to accommodate 2 thread blocks (1536 threads), leaving 512 thread slots unutilized. In this case, neither the maximum threads per SM nor the maximum blocks per SM is reached. The occupancy in this case is (1536 assigned threads)/(2048 maximum threads) = 75%.

The preceding discussion does not consider the impact of other resource constraints such as registers and shared memory. We will see in Chapter 5 that automatic variables declared in a CUDA kernel are placed into registers. Some kernels may use many automatic variables and others may use few of them. Thus, one should expect that some kernels require many registers per thread, and some require few. By dynamically partitioning registers in an SM across threads, the SM can accommodate many blocks if they require few registers per thread and fewer blocks if they require more registers per thread.

One does, however, need to be aware of the potential impact of register resource limitations on occupancy. For example, the H100 GPU allows a maximum of 65,536 registers per SM. To run at full occupancy, each SM needs enough registers for 2048 threads which means that each thread should not use more than (65,536 registers)/(2048 threads) = 32 registers per thread. If, for example, a kernel uses 64 registers per thread, the maximum number of threads that can be supported with 65,536 registers is 1024 threads. In this case, the kernel cannot run with full occupancy regardless of what the block size is set to be. Instead, the occupancy will be at most 50%. In some cases, the compiler may perform register spilling to reduce the register requirement per thread and thus elevate the level of occupancy. This, however, is typically at the cost of increased execution time for the threads to access the spilled register values from memory and may cause the total execution time of the grid to increase. A similar occupancy analysis is done for the shared memory resource in 5.

Assume that a programmer implements a kernel that uses 31 registers per thread and configures it with 512 threads per block. In this case, the SM will have (2048 threads)/(512 threads/block) = 4 blocks running simultaneously. These threads will use a total of (2048 threads)×(31 registers/thread) = 63,488 registers which is less than the 65,536 register limit. Now assume that the programmer declares another two automatic variables in the kernel that bumps the number of registers used by each thread to 33. The number of registers required by 2048 threads is now 67,584 registers, which exceeds the register limit. The CUDA runtime system may deal with this situation by assigning only 3 blocks to each SM instead of 4, thus reducing the number of registers required to 50,688 registers. This, however, reduces the number of threads running on an SM from 2048 to 1536. That is, by using two extra automatic variables, the program saw a reduction in occupancy from 100% to 75%. This is sometimes referred to as a "performance cliff" where a slight increase in resource usage can result in significant reduction in parallelism and performance achieved [4].

It should be clear to the reader that the constraints of all the dynamically partitioned resources interact with each other in a complex manner. Accurate determination of the number of threads running in each SM can be difficult. The reader is referred to the Occupancy Calculator [5] that calculates the actual number of threads running on each SM for a particular device implementation given the usage of resources by a kernel. The Occupancy Calculator is part of the NVIDIA Nsight Compute profiler [6]. The application host code can also call the CUDA Occupancy API functions such as `cudaOccupancyMaxActiveBlocksPerMultiprocessor()` to determine the occupancy of a kernel when called with a grid configuration on the GPU being used.

4.8 Querying device properties

Our discussion on partitioning of SM resources raises an important question. How do we find out the amount of resources available for a particular device? When a CUDA application executes on a system, how can it find out the number of SMs in a device, and the number of blocks and threads that can be assigned to each SM? The same questions apply to other kinds of resources, some of which we have not discussed so far. In general, many modern applications are designed to execute on a wide variety of hardware systems. There is often a need for the application to query the available resources and capabilities of the underlying hardware in order to take advantage of the more capable systems while compensating for the less capable systems (see "Resource and Capability Queries" sidebar).

Resource and Capability Queries

In everyday life, we often query the resources and capabilities in an environment. For example, when we make a hotel reservation, we can check the amenities that come with a hotel room. If the room comes with a hair dryer, we do not need to bring one. Most American hotel rooms come with hair dryers while many hotels in other regions do not.

Some Asian and European hotels provide toothpastes and even toothbrushes while most American hotels do not. Most American hotels provide both shampoo and conditioner while hotels in other continents often only provide shampoo.

If the room comes with a microwave oven and a refrigerator, we can take the leftover from dinner and expect to eat it the second day. If the hotel has a pool, we can bring swimsuits and take a dip after business meetings. If the hotel does not have a pool but has an exercise room, we can bring running shoes and exercise clothes. Some high-end Asian hotels even provide exercise clothing!

These hotel amenities are part of the properties, or resources and capabilities, of the hotels. Veteran travelers check these properties at hotel web sites, choose the hotels that better match their needs, and pack more efficiently and effectively.

The amount of resources in each CUDA device SM is specified as part of the *compute capability* of the device. In general, the higher the compute capability level, the more the resources available in each SM. The compute capability of GPUs also tends to increase from generation to generation. The A100 GPU is of compute capability 8.0 and the H100 GPU compute capability 9.0.

In CUDA C++, there is a built-in mechanism for host code to query the properties of the devices available in the system. The CUDA run-time system (device driver) has an API function `cudaGetDeviceCount` that returns the number of CUDA-capable devices in the system. The host code can find out the number of available CUDA devices using the following statements:

```
1    int devCount;
2    cudaGetDeviceCount(&devCount);
```

While it may not be obvious, a modern PC system often has two or more CUDA-capable devices. This is because many PC systems come with one or more "integrated" GPUs. These GPUs are the default graphics units and provide rudimentary capabilities and hardware resources to perform minimal graphics functionalities for modern window-based user interfaces. Most CUDA applications will not perform very well on these integrated devices. This would be a reason for the host code to iterate through all the available devices, query their resources and capabilities, and choose the ones that have enough resources to execute the application with satisfactory performance.

The CUDA runtime numbers all the available devices in the system from 0 to `devCount-1`. It provides an API function `cudaGetDeviceProperties` that returns the properties of the device whose number is given as an argument. For example, we can use the following statements in the host code to iterate through the available devices and query their properties:

```
1    cudaDeviceProp devProp;
2    for(unsigned int i = 0; i < devCount; i++) {
3        cudaGetDeviceProperties(&devProp, i);
4        // Decide if device has sufficient resources/capabilities
5    }
```

The built-in type `cudaDeviceProp` is a C struct type with fields that represent the properties of a CUDA device. The reader is referred to the CUDA C++ Programming Guide for all the fields of the type. We will discuss a few of these fields that are particularly relevant to the assignment of execution resources to threads. We assume that the properties are returned in the `devProp` variable whose fields are set by the `cudaGetDeviceProperties` function. If the reader chooses to name the variable differently, the appropriate variable name will obviously need to be substituted in the following discussion.

As the name suggests, the field `devProp.maxThreadsPerBlock` gives the maximal number of threads allowed in a block in the queried device. Some devices allow up to 1024 threads in each block and other devices may allow fewer. It is possible that future devices may even allow more than 1024 threads per block. Therefore, it is a good idea to query the available devices and determine which ones will allow sufficient number of threads in each block as far as the application is concerned. For CUDA programs that use thread block clusters, it is recommended to query the maximum cluster size, which can be obtained by calling the `cudaOccupancyMaxPotentialClusterSize()` API function.

The total number of SMs in the device is given in the field `devProp.multiProcessorCount`. If the application requires many SMs to achieve satisfactory performance, it should definitely check this property of the prospective device. Furthermore, the clock frequency of the device is in `devProp.clockRate`. The combination the clock rate and the number of SMs gives a good indication of the maximal hardware execution throughput of the device.

The host code can find the maximal number of threads allowed along each dimension of a block in fields `devProp.maxThreadsDim[0]` (for the x dimension), `devProp.maxThreadsDim[1]` (for the y dimension), and `devProp.maxThreadsDim[2]` (for the z dimension). An example use of this information is for an automated tuning system to set the range of block dimensions when evaluating the best performing block dimensions for the underlying hardware. Similarly, it can find the maximal number of blocks allowed along each dimension of a grid in `devProp.maxGridSize[0]` (for the x dimension), `devProp.maxGridSize[1]` (y dimension), and `devProp.maxGridSize[2]` (z dimension). A typical use of this information is to determine whether a grid can have enough threads to handle the entire data set or if some kind of iterative approach is needed.

The field `devProp.regsPerBlock` gives the number of registers available in each SM. This field can be useful in determining whether the kernel can achieve maximum occupancy on a particular device or if it will be limited by its register usage. Note that the name of the field is a little misleading. For most compute capability levels, the maximum number of registers a block can use is indeed the same as the total number of registers available in the SM. However, for some compute capability levels, the maximum number of registers a block can use is less than the total available on the SM.

We have also discussed that the size of warps depends on the hardware. The size of warps can be obtained from the `devProp.warpSize` field.

There are many more fields in the `cudaDeviceProp` type. We will discuss them throughout the book as we introduce the concepts and features that they are designed to reflect.

4.9 Summary

A CUDA-capable GPU is organized into Streaming Multiprocessors (SMs) which consist of multiple processing blocks of streaming processors that share control logic

and memory resources. When a grid is launched, its blocks are assigned to SMs in arbitrary order, resulting in transparent scalability of CUDA applications. The transparent scalability comes with a constraint: threads in different thread blocks should not synchronize with each other. Starting with the Hopper architecture, SMs are grouped into GPCs, and blocks are optionally grouped into thread block clusters. Blocks in the same cluster execute on the same GPC, can synchronize with each other, and can access each other's distributed shared memory.

Threads are assigned to SMs for execution on a block-by-block basis. Once a block is assigned to an SM, it is further partitioned into warps. Threads in a warp are executed following the Single Instruction Multiple Data (SIMD) model. If threads in the same warp diverge by taking different execution paths, the processing block executes these paths in passes where each thread is only active in the pass corresponding to the path that it takes.

An SM may have many more threads assigned to it than it can execute simultaneously. At any time, the SM executes instructions of only a subset of its resident warps. This allows the other warps to wait for long latency operations without affecting the overall execution throughput of the massive number of processing units. The ratio of the number of threads assigned to the SM to the maximum number of threads it can support is referred to as *occupancy*. The higher the occupancy of an SM, the better it can hide long operation latencies and achieve high execution throughput.

Each CUDA-capable device imposes a potentially different limitation on the amount of resource available in each SM. For example, each CUDA device has a limit on the number of blocks, the number of threads, the number of registers, and the amount of other resources that each of its SMs can accommodate. For each kernel, one or more of these resource limitations can become the limiting factor for occupancy. CUDA C++ provides occupancy calculators and API functions for developers to choose the best kernels and launch configurations for different GPUs. It also provides programmers with the ability to query the resources available in a GPU at run-time.

4.10 Exercises

1. Consider the following CUDA kernel and the corresponding host function that calls it:

```
__global__ void foo_kernel(int* a, int* b) {
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(threadIdx.x < 40 || threadIdx.x >= 104) {
        b[i] = a[i] + 1;
    }
    if(i%2 == 0) {
        a[i] = b[i]*2;
    }
    for(unsigned int j = 0; j < 5 - (i%3); ++j) {
```

```
10              b[i] += j;
11          }
12      }
13      void foo(int* a_d, int* b_d) {
14          unsigned int N = 1024;
15          foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
16      }
```

a. What is the number of warps per block?
b. What is the number of warps in the grid?
c. For the statement on line 04:
  i. How many warps in the grid are active?
  ii. How many warps in the grid are divergent?
  iii. What is the SIMD efficiency (in %) of warp 0 of block 0?
  iv. What is the SIMD efficiency (in %) of warp 1 of block 0?
  v. What is the SIMD efficiency (in %) of warp 3 of block 0?
d. For the statement on line 07:
  i. How many warps in the grid are active?
  ii. How many warps in the grid are divergent?
  iii. What is the SIMD efficiency (in %) of warp 0 of block 0?
e. For the loop on line 09:
  i. How many iterations have no divergence?
  ii. How many iterations have divergence?

2. For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?

3. For the previous question, how many warps do you expect to have divergence due to the boundary check on vector length?

4. Consider a hypothetical block with 8 threads executing a section of code before reaching a barrier. The threads require the following amount of time (in micro seconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, 2.9 and spend the rest of their time waiting for the barrier. What percentage of the threads' total execution time is spent waiting for the barrier?

5. A CUDA programmer says that if they launch a kernel with only 32 threads in each block, they can leave out the __syncthreads() instruction wherever barrier synchronization is needed. Do you think this is a good idea? Explain.

6. If a CUDA device's SM (streaming multiprocessor) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the most number of threads in the SM?
a. 128 threads per block
b. 256 threads per block
c. 512 threads per block
d. 1024 threads per block

7. Assume a device that allows up to 64 blocks per SM and 2048 threads per SM. Indicate which of the following assignments per SM is possible. In the cases where it is possible, indicate the occupancy level.
   a. 8 blocks with 128 threads each
   b. 16 blocks with 64 threads each
   c. 32 blocks with 32 threads each
   d. 64 blocks with 32 threads each
   e. 32 blocks with 64 threads each
8. Consider a GPU with the following hardware limits: 2048 threads/SM, 32 blocks/SM, and 64 K (65,536) registers/SM. For each of the following kernel characteristics, specify if the kernel can achieve full occupancy. If not, specify the limiting factor.
   a. The kernel uses 128 threads/block and 30 registers/thread.
   b. The kernel uses 32 threads/block and 29 registers/thread.
   c. The kernel uses 256 threads/block and 34 registers/thread.
9. A student mentions that they were able to multiply two 1024×1024 matrices using a matrix multiplication kernel with 32×32 thread blocks. The student is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. The student further mentions that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

References

[1] NVIDIA, CUDA C++ programming guide. Release 12.3, https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2023.
[2] M. Flynn, Very high-speed computing systems, Proceedings of the IEEE 54 (12) (1966) 1901–1909.
[3] NVIDIA, NVIDIA Tesla V100 GPU Architecture. Version WP-08608-001_v1.1, 2017.
[4] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.-Z. Ueng, J.A. Stratton, W.-m.W. Hwu, Program optimization space pruning for a multithreaded GPU, in: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2008, pp. 195–204.
[5] NVIDIA, Nsight compute: occupancy calculator, https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator, 2023.
[6] NVIDIA, Nsight compute, https://docs.nvidia.com/nsight-compute/index.html, 2023.

CHAPTER

Memory architecture and data locality

5

CONTENTS

5.1 Memory bandwidth as a performance limiter 93
5.2 CUDA memory types 98
5.3 Tiling for reduced memory traffic 104
5.4 A tiled matrix multiplication kernel 108
5.5 Boundary checks 112
5.6 Impact of memory usage on occupancy 115
5.7 Summary 118
5.8 Exercises 119
References 121

So far, we have learned how to write a CUDA kernel function and how to configure and coordinate its execution by a massive number of threads. We have also looked at the compute architecture of GPU hardware and how threads are scheduled to execute. In this chapter, we will focus on the on-chip memory architecture of the GPU and begin to study how one can organize and position data for efficient access by a massive number of threads. The CUDA kernels that we have learned so far will likely achieve only a tiny fraction of the potential speed of the underlying hardware. This poor performance is because global memory, which is typically implemented with off-chip DRAM, tends to have long access latency (hundreds of clock cycles) and finite access bandwidth. While having many threads available for execution can theoretically tolerate long memory-access latencies, one can easily run into a situation where traffic congestion in the global memory access paths prevents all but very few threads from making progress, thus rendering some of the cores in the Streaming Multiprocessors (SMs) idle. To circumvent such congestion, GPUs provide a number of additional on-chip memory resources for accessing data that can remove the majority of redundant traffic to and from the global memory. In this chapter, we study the use of different memory types to boost the execution performance of CUDA kernels.

5.1 Memory bandwidth as a performance limiter

Since GPUs have a finite amount of hardware resources, they have limits on the amount of work that they can perform in a given time frame. The two most commonly

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00014-7
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

referenced limits are the *peak computational throughput* and the *peak memory bandwidth*. The peak computational throughput specifies the limit on how many arithmetic operations the hardware can perform per unit of time. For example, the Hopper H100 GPU has a peak computational throughput of 66.9 terra floating-point operations per second (TFLOPS). The hardware typically has different limits for different datatypes such as 32-bit integers, 64-bit integers, double-precision floating point values, etc. We will focus on single-precision floating point operations in this chapter.

The peak memory bandwidth specifies the limit on how many bytes of data the hardware can access from a particular memory structure per unit of time. For example, the Hopper H100 GPU has a peak global memory bandwidth of 3.35 terra bytes per second (TB/s). The hardware typically has different limits for different memory structures. We will focus on global memory bandwidth in this chapter.

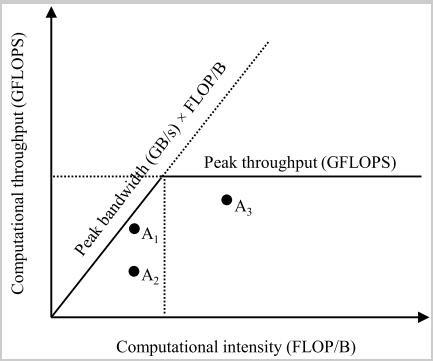
Different kernels find their performance limited by different hardware resources. A kernel is said to be *compute-bound* if its performance is limited by the computational throughput of the hardware. For such kernels, the compute cores are busy the majority of the time with a steady supply of instructions to execute. Their performance is limited by how fast the cores can execute these instructions. A kernel is said to be *memory-bound* if its performance is limited by the memory bandwidth of the hardware. For such kernels, the memory channels are busy the majority of the time, and the compute cores are frequently idle waiting for data to arrive from memory. Their performance is limited by how fast the memory can supply the cores with data.

To determine if a kernel is compute-bound or memory-bound, we can look at the ratio of floating-point operations it performs to bytes it accesses from global memory (FLOP/B). We will refer to this ratio as the *compute-to-global-memory-access ratio*. This ratio is sometimes also referred to as *arithmetic intensity* or *computational intensity* in the literature. Compute-bound kernels tend to have a high *compute-to-global-memory-access ratio*, performing many arithmetic operations relative to the amount of memory that they access. On the other hand, memory-bound kernels tend to have a small ratio, performing few arithmetic operations relative to the amount of memory that they access. The threshold that separates memory-bound kernels from compute-bound kernels depends on the hardware and can be found by taking the ratio of the peak computational throughput to the peak memory bandwidth. For example, the H100 GPU has a threshold of (66.9 TFLOPS)/(3.35 TB/s) = 20.0 FLOP/B. In other words, kernels whose *compute-to-global-memory-access ratio* is higher than 20.0 FLOP/B are likely to be compute-bound on the H100 GPU, whereas kernels whose ratio is lower are likely to be memory-bound.

The peak computational throughput and the peak memory bandwidth are hardware *limits*, not *guarantees*. A compute-bound kernel is not guaranteed to perform 66.9 TFLOPS, and a memory-bound is not guaranteed to access 3.35 TB/s. Only the most well-optimized kernels that use hardware resources efficiently are able to perform close to these limits. Accordingly, by comparing the performance of one's kernel to these hardware limits, one can assess how efficiently one's kernel is optimized. The *roofline model* is a convenient way to analyze whether a kernel is compute-bound or memory-bound, and how well it performs relative to the hardware's peak (see The Roofline Model sidebar).

The Roofline Model

The Roofline Model is a visual model for assessing the performance achieved by an application relative to the limits of the hardware it is running on. A basic example of the Roofline model is shown below.



On the x-axis, we have arithmetic or computational intensity measured in FLOP/B. It reflects the amount of work done by an application for every byte of data loaded. On the y-axis, we have computational throughput measured in GFLOPS. The two lines inside of the plot reflect the hardware limits. The horizontal line is determined by the peak computational throughput (GFLOPS) that the hardware can sustain. The line with a positive slope starting from the origin is determined by the peak memory bandwidth that the hardware can sustain. The point of intersection between these two lines represents the computational intensity threshold at which applications transition from being memory-bound to being compute bound. Applications with lower computational intensity are memory-bound and cannot achieve peak computational throughput because they are limited by memory bandwidth. Within this regime, increasing an application's computational intensity elevates its potential computational throughput, as reflected by the position slope of the hardware limit line. Once an application's computational intensity increases beyond the threshold, it becomes compute-bound and is no longer limited by the peak memory bandwidth but by the peak computational throughput.

A point in the plot represents an application with its computational intensity on the x-axis and the computational throughput it achieves on the y-axis. Of course, the points will be under the two lines because they cannot achieve higher throughput than the hardware peak. The position of a point relative to the two lines tells us about an applications efficiency. Points close to the two lines indicate that an application is using memory bandwidth or compute units efficiently, whereas applications far below the lines indicate inefficient use of resources.

As an example, points A1 and A2 both represent memory-bound applications, while A3 represents a compute-bound application. A1 uses resources efficiently

and operates close to the peak memory bandwidth, whereas A2 does not. For A2, there may be room for additional optimizations to improve throughput by improving memory bandwidth utilization. However, for A1 the only way to improve throughput is to increase the computational intensity of the application.

As an example of reasoning about a kernel's resource bound and efficiency, let us consider the vector addition kernel implemented in Fig. 2.10 from Chapter 2. The main computation performed by each thread in this kernel is replicated as follows:

```
C[i] = A[i] + B[i];
```

We observe that each thread performs one floating point addition operation and accesses 12 B from global memory (4 B to load `A[i]`, 4 B to load `B[i]`, and 4 B to store `C[i]`). Accordingly, the kernel's *compute-to-global-memory-access ratio* is (1 FLOP)/(12 B) = 0.083 FLOP/B which is much less than the H100 GPU's threshold of 20.0 FLOP/B. With such a low ratio, the vector addition kernel executing on the H100 GPU is deep in the memory-bound regime.

Having established that the vector addition kernel is memory-bound, we can now reason about its performance relative to the hardware peak. If the kernel is executed to add two vectors with one billion elements each, the kernel will access 12 GB of data (4 GB to load the first vector, 4 GB to load the second vector, and 4 GB to store the result vector). If the kernel completes execution in 4 ms, then the average rate at which the data was accessed from memory is (12 GB)/(4 ms) = 3 TB/s. If we divide this rate by the hardware's peak memory bandwidth, we get (3 TB/s)/(3.35 TB/s) = 90%, which means that the kernel utilizes 90% of the hardware's memory bandwidth resources. This kind of analysis is referred to as the *speed-of-light analysis*, and we say that the kernel executes at 90% of the speed of light, i.e., the speed as limited by the hardware. The analysis is useful because it indicates that the kernel utilizes hardware resources efficiently, and that there is only 10% room for further performance optimization. In fact, we can estimate that the fastest possible vector addition kernel when adding two vectors with one billion elements each would take (12 GB)/(3.35 TB/s) = 3.6 ms on the H100 GPU.

Let us now apply the same analysis to matrix multiplication. Before looking at a specific implementation, let us reason about matrix multiplication at an abstract level. Assume we would like to multiply two $N \times N$ matrices and get a result that is $N \times N$. For each of the $N \times N$ output elements, we need to perform a dot-product of two vectors of size N which consists of N multiply operations and N add operations. In total, the matrix multiplication operation performs $N^2 \cdot (N + N) = 2N^3$ FLOP. As for the memory accessed, the operation needs to load two matrices with N^2 elements each and store one matrix with N^2 elements. Since each matrix element is 4 B, the total amount of data accessed by the computation is $12N^2$ B. Therefore, if we assume a perfect implementation that accesses each data element only once, the *compute-to-global-memory-access ratio* for matrix multiplication would be $(2N^3)/(12N^2) = 0.167N$ FLOP/B. For $N = 1024$, the ratio would be 167 FLOP/B

which is significantly higher than the H100 GPU's threshold of 25.1 FLOP/B or 20.0 FLOP/B. Clearly, matrix multiplication has the potential to be a highly compute-bound computation!

We now turn our attention to our specific implementation of matrix multiplication in Fig. 3.11 from Chapter 3. The most dominant part of the kernel in terms of execution time is the for-loop that performs the dot-product of a row of `M` with a column of `N`, which is replicated as follows:

```
for (int k = 0; k < Width; ++k) {
    Pvalue += M[row*Width+k] * N[k*Width+col];
}
```

In every iteration of the loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition. The global memory accesses fetch elements from the `M` and `N` arrays. The floating-point multiplication operation multiplies these two elements together, and the floating-point add operation accumulates the product into `Pvalue`. Thus, the compute-to-global-memory-access ratio is 2 FLOP to 8 B, or 0.25 FLOP/B. The ratio is actually slightly lower because the `Pvalue` needs to be stored into the global memory after each thread exits the `for`-loop. This ratio is lower than the ideal ration of $0.167N$ of a perfect implementation because each input element is accessed multiple times. The ratio is even lower than the H100 GPU's threshold of 20.0 FLOP/B, which makes our specific implementation of matrix multiplication memory-bound instead of compute bound.

Since our matrix multiplication kernel is memory-bound, the global memory bandwidth limits the throughput of single-precision floating-point operations that can be performed by the kernel to (3.35 TB/s)(0.25 FLOP/B) = 0.84 TFLOPS. However, 0.84 TFLOPS is only around 1% of the peak single-precision operation throughput of the H100 GPU which is 66.9 TFLOPS. The H100 also comes with special purpose units called tensor cores that are useful for accelerating matrix multiplication operations. If one considers the H100's tensor core peak throughput of 989 TFLOPS, 0.84 TFLOPS is only 0.08% of the peak. In practice, the actual compute-to-global-memory-access ratio is higher than 0.25 FLOP/B because some of the global memory accesses are served from the on-chip caches instead of the DRAM. Hence, the kernel would achieve a higher percentage of the peak compute throughput. Still, we can further improve the performance of our matrix multiplication implementation by deliberately increasing the ratio at the kernel implementation level.

To increase the compute-to-global-memory-access ratio of the kernel, we must reduce the number of global memory accesses it performs. As we will see, matrix multiplication presents opportunities for reduction of global memory accesses that can be captured with relatively simple techniques. The execution speed of matrix multiplication functions can vary by orders of magnitude depending on the level of reduction of global memory accesses. Therefore, matrix multiplication provides an excellent initial example for such techniques. This chapter introduces a commonly used technique for reducing the number of global memory accesses and demonstrates the technique on matrix multiplication.

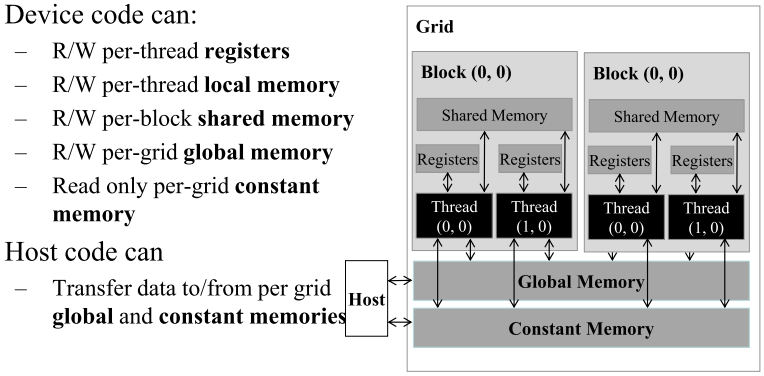


FIGURE 5.1

An (incomplete) overview of the CUDA device memory model. An important type of CUDA memory not covered in this figure is the texture memory since its use is not covered in this textbook.

5.2 CUDA memory types

A CUDA device contains several types of memory that can help programmers to improve the compute-to-global-memory-access ratio of their kernels. By declaring a CUDA variable into one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable. Fig. 5.1 shows these CUDA device memories. At the bottom of the picture, we see *global memory* and *constant memory*. Both these types of memory can be written (W) and read (R) by the host. The global memory can also be written and read by the device, whereas the constant memory supports short-latency, high bandwidth read-only access by the device. We have already introduced global memory in Chapter 2 and will look at constant memory in detail in Chapter 7.

Another type of memory is the *local memory*, which can also be read and written. CUDA variables declared into the local memory are actually placed in global memory and have similar access latency, but are not shared across threads. Each thread has its own section of global memory that it uses as its own private local memory where it places its local variables and data that cannot be allocated in registers. Such variables and data include statically allocated arrays, spilled registers, and other elements of the thread's call stack. However, statically allocated arrays that have small and constant size and are accessed only with constant indices can be allocated into registers (see Chapter 6 for details).

Registers in Fig. 5.1 are one type of on-chip memory. Variables that reside in registers can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual threads; each thread can only access its own registers (see the "CPU vs. GPU Register Architecture" sidebar). A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

CPU vs. GPU Register Architecture

The different design objectives across the CPUs and GPUs result in different register architectures. As we saw in Chapter 4, when CPUs context switch between different threads, they save the registers of the outgoing thread to memory and restore the registers of the incoming thread from memory. In contrast, GPUs achieve zero-overhead scheduling by keeping the registers of all the threads that are scheduled on the processing block in the processing block's register file. This way, switching between warps of threads is instantaneous because the registers of the incoming threads are already in the register file. Consequently, GPU register files need to be substantially larger than CPU register files.

We also saw in Chapter 4 that GPUs support dynamic resource partitioning where an SM may provision few registers per thread and execute a large number of threads, or it my provision more registers per thread and execute fewer threads. For this reason, GPU register files need to be designed to support such dynamic partitioning of registers. In contrast, the CPU register architecture dedicates a fixed set of registers per thread regardless of the thread's actual demand for registers.

Shared memory in Fig. 5.1 is another type of on-chip memory. Data that resides in shared memory can also be accessed at a high speed relative to global memory, though not as fast as registers. Shared memory is allocated to thread blocks; all threads in a block can access shared-memory variables declared for the block. Shared memory is an efficient means for threads in the same block to cooperate by sharing their input data and intermediate results.

To fully appreciate the difference between registers, shared memory, and global memory, we need to go into a little more detail of how these different memory types are realized and used in modern processors. As we discussed in the "Warps and SIMD Hardware" sidebar of Chapter 4, virtually all modern processors find their root in the model proposed by John von Neumann in 1945, which is shown in Fig. 5.2. CUDA devices are no exception. The global memory in a CUDA device maps to the Memory box in Fig. 5.2. The processor box corresponds to the processor chip boundary that we typically see today. The global memory is off the processor chip and is implemented with DRAM technology, which implies long access latencies and relatively low access bandwidth. The registers correspond to the "Register File" of the von Neumann model. The Register File is on the processor chip, which implies very short access latency and drastically higher access bandwidth when compared to the global memory. In a typical device, the aggregated access bandwidth of all the register files across all the SMs is at least two orders of magnitude higher than that of the global memory. Furthermore, whenever a variable is stored in a register, its accesses no longer consume off-chip global memory bandwidth. This will be reflected as an increased compute-to-global-memory-access ratio.

A subtler point is that an access to a register involves fewer instructions than an access to the global memory. Arithmetic instructions in most modern processors have

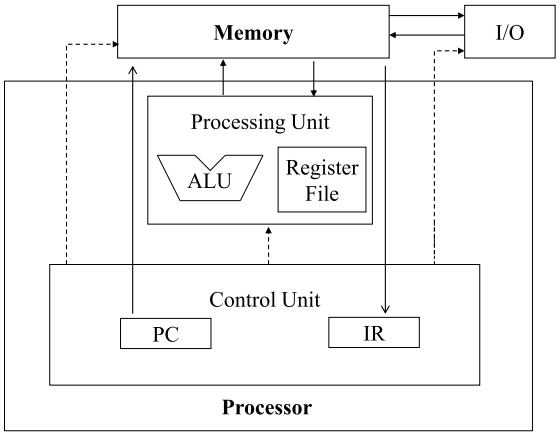


FIGURE 5.2

Memory vs. registers in a modern computer based on the von Neumann model.

"built-in" register operands. For example, a floating-point addition instruction might be of the form:

```
fadd r1, r2, r3
```

where `r2` and `r3` are the register numbers that specifies the location in the register file where the input operand values can be found. The location for storing the floating-point addition result value is specified by `r1`. Therefore, when an operand of an arithmetic instruction is in a register, there is no additional instruction required to make the operand value available to the arithmetic and logic unit (ALU), where the arithmetic calculation is done.

Meanwhile, if an operand value is in the global memory, the processor needs to perform a memory load operation to make the operand value available to the ALU. For example, if the first operand of a floating-point addition instruction is in the global memory, the instructions involved will likely look like the following example:

```
load r2, r4, offset
fadd r1, r2, r3
```

where the load instruction adds an offset value to the contents of `r4` to form an address for the operand value. It then accesses the global memory and places the value into register `r2`. Once the operand value is in `r2`, the `fadd` instruction performs the floating-point addition using the values in `r2` and `r3` and places the result into `r1`. Since the processor can only fetch and execute a limited number of instructions per clock cycle, the version with an additional load will likely take more time to process than the one without. This is another reason why placing the operands in registers can improve execution speed.

Finally, there is yet another subtle reason why placing an operand value in registers is preferable. In modern computers, the energy consumed for accessing a value

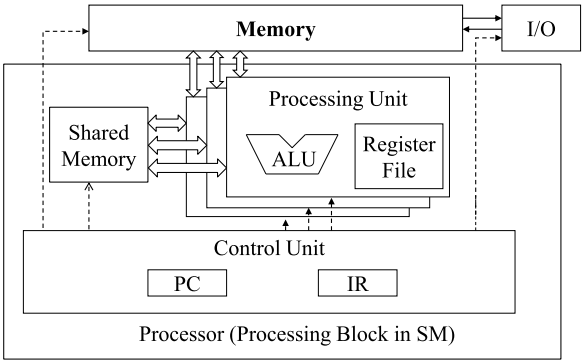


FIGURE 5.3

Shared memory vs. registers in a CUDA device SM.

from the register file is at least an order of magnitude lower than for accessing a value from the global memory. Accessing a value from registers has a tremendous advantage in energy efficiency over accessing the value from the global memory. We will look at more details of the speed and energy difference in accessing these two hardware structures in modern computers soon. On the other hand, as we will soon learn, the number of registers available to each thread is quite limited in today's GPUs. As we have seen in Chapter 4, the occupancy achieved for an application can be reduced if the register usage exceeds the limit for full-occupancy scenarios. Therefore, we also need to avoid oversubscribing to this limited resource whenever possible.

Fig. 5.3 shows the shared memory and registers in a CUDA device. Although both are on-chip memories, they differ significantly in functionality and cost of access. Shared memory is designed as part of the memory space that resides on the processor chip. When the processor accesses data that reside in the shared memory, it needs to perform a memory load operation, just like accessing data in the global memory. However, because shared memory resides on chip, it can be accessed with much lower latency and much higher throughput than the global memory. Because of the need to perform a load operation, shared memory has longer latency and lower bandwidth than registers. In computer architecture terminology, the shared memory is a form of scratchpad memory.

One important difference between the shared memory and registers in CUDA is that variables that reside in the shared memory are accessible by all threads in a block. This contrasts with register data, which is private to a thread. That is, shared memory is designed to support efficient, high bandwidth sharing of data among threads in the same block. As shown in Fig. 5.3, a CUDA device SM typically employs multiple processing units, to allow multiple threads to make simultaneous progress (see "Threads" sidebar in Chapter 2). Threads in a block can be spread across these processing units. Therefore, the hardware implementations of the shared memory in these CUDA devices are typically designed to allow multiple processing units to simultaneously access its contents to support efficient data sharing among threads in

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	register	thread	grid
Automatic array variables	local	thread	grid
__device__ __shared__ int SharedVar;	shared	block	grid
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstVar;	constant	grid	application

FIGURE 5.4

CUDA variable declaration type qualifiers and the properties of each type.

a block. We will be learning several important types of parallel algorithms that can greatly benefit from such efficient data sharing among threads.

Starting with the Hopper architecture, threads in the same thread block cluster can access the shared memory of any block in the cluster. This feature is referred to as *distributed shared memory*. Distributed shared memory expands the scope of threads that can collaborate without accessing global memory, and gives them access to a larger collective pool of fast memory.

It should be clear by now that registers, local memory, shared memory, and global memory all have different functionalities, latencies, and bandwidth. It is therefore important to understand how to declare a variable so that it will reside in the intended type of memory. Fig. 5.4 presents the CUDA syntax for declaring program variables into the various memory types. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the set of threads that can access the variable: a single thread only, all threads of a block, or all threads of all grids. If a variable's scope is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable. For example, if a kernel declares a variable whose scope is a thread and it is launched with one million threads, one million versions of the variable will be created so that each thread initializes and uses its own version of the variable.

Lifetime tells the portion of the program's execution duration when the variable is available for use: either within a grid's execution or throughout the entire application. If a variable's lifetime is within a grid's execution, it must be declared within the kernel function body and will be available for use only by the kernel's code. If the kernel is invoked several times, the value of the variable is not maintained across these invocations. Each invocation must initialize the variable in order to use it. On the other hand, if a variable's lifetime is throughout the entire application, it must be declared outside of any function body. The contents of these variables are maintained throughout the execution of the application and are available to all kernels.

We refer to variables that are not arrays as scalar variables. As shown in Fig. 5.4, all automatic scalar variables declared in kernel and device functions are placed into registers. The scopes of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all its automatic variables also cease to exist. For example, in Fig. 3.8 in Chapter 3,

the variables `blurRow`, `blurCol`, `curRow`, `curCol`, `pixels`, and `pixVal` are all automatic variables and fall into this category. Note that accessing these variables is extremely fast and parallel, but one must be careful not to exceed the limited capacity of the register storage in the hardware implementations. Using a large number of registers can negatively impact the occupancy of each SM, as we have seen in Chapter 4.

Automatic array variables are not stored in registers by default. They are stored into the thread's local memory and may incur long access delays and potential access congestion. The compiler may decide to store an automatic array variable in registers instead of local memory if all accesses are done with constant index values. We revisit this point in Chapter 6. The scope of automatic array variables, like automatic scalar variables, is limited to individual threads. That is, a private version of each automatic array is created for and used by every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist.

If a variable declaration is preceded by the `__shared__` keyword (each "__" consists of two "_" characters), it declares a shared memory variable in CUDA. One can also add an optional `__device__` in front of `__shared__` in the declaration to achieve the same effect. Such a declaration is typically made within a kernel function or a device function. The scope of a shared memory variable is within a thread block, that is, all threads in a block see the same version of a shared memory variable. A private version of the variable is created for and used by each block during kernel execution. The lifetime of a shared memory variable is within the duration of the kernel execution. When a kernel terminates its grid's execution, the contents of its shared memory variables cease to exist. As we discussed earlier, shared memory variables are an efficient means for threads within a block to collaborate with each other. Accessing shared memory variables is extremely fast and highly parallel. CUDA programmers often use shared memory variables to hold the portion of global memory data that are frequently used and reused in an execution phase of the kernel. One may need to adjust the algorithms used to create execution phases that heavily focus on small portions of the global memory data, as we will demonstrate with matrix multiplication in Section 5.4.

If a variable declaration is preceded by the `__constant__` keyword (each "__" consists of two "_" characters), it declares a constant variable in CUDA. One can also add an optional `__device__` in front of `__constant__` to achieve the same effect. Declaration of constant variables must be outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution. Constant variables are often used to provide input values to kernel functions. The values of the constant variables cannot be changed by the kernel function code. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at 65,536 bytes. One may need to break up the input data volume to fit within this limit. We will demonstrate the usage of constant memory in Chapter 7.

A variable whose declaration is preceded only by the keyword `__device__` (each "__" consists of two "_" characters) is a global variable and will be placed in the

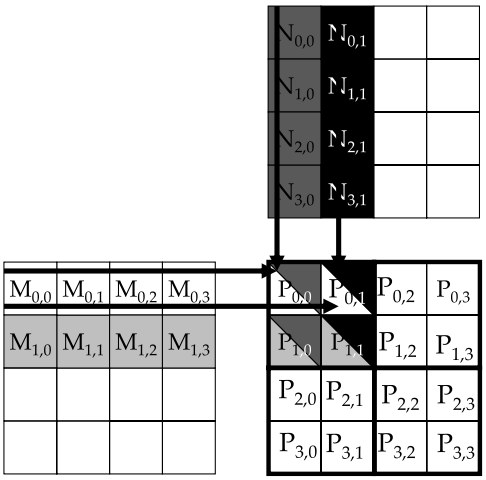


FIGURE 5.5

A small example of matrix multiplication. For brevity, We show `M[y*Width+x]`, `N[y*Width+x]`, and `P[y*Width+x]` as $M_{y,x}$, $N_{y,x}$, and $P_{y,x}$, respectively.

global memory. Access to a global variable is slow. Global variables are visible to all threads of all kernels and their contents persist throughout the entire execution. Global variables are often used to pass information from one kernel invocation to another kernel invocation without providing the information as a kernel parameter. In general, global variables are considered bad style and should be used sparingly because they can cause bugs and negatively impact the code's modularity.

5.3 Tiling for reduced memory traffic

We have an intrinsic tradeoff in the use of device memories in CUDA: the global memory is large but slow whereas the shared memory is small but fast. A common strategy is to partition the data into subsets called tiles so that each tile fits into the shared memory. The term tile draws on the analogy that a large wall (i.e., the global memory data) can be covered by small tiles (i.e., subsets that each can fit into the shared memory). An important criterion is that the kernel computation on these tiles can be done independently of each other. Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example from Chapter 3. Fig. 3.13 showed a small example of matrix multiplication. It corresponds to the kernel function in Fig. 3.11. We replicate the example in Fig. 5.5 for convenient reference by the reader. For brevity, we abbreviate `P[y*Width+x]`, `M[y*Width+x]`, and `N[y*Width+x]` as $P_{y,x}$, $M_{y,x}$, and $N_{y,x}$, respectively. This example assumes that we use four 2×2 blocks to compute the P matrix. The heavy

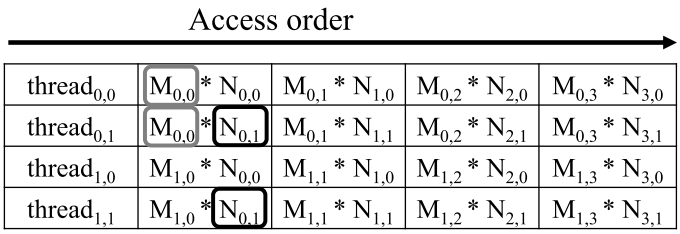


FIGURE 5.6

Global memory accesses performed by threads in $block_{0,0}$.

boxes in the P matrix define the output tile of P elements processed by each block. Fig. 5.5 highlights the computation done by the four threads of $block_{0,0}$. These four threads compute $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$. The accesses to the M and N elements by $thread_{0,0}$ and $thread_{0,1}$ of $block_{0,0}$ are highlighted with black arrows. For example, $thread_{0,0}$ reads $M_{0,0}$ and $N_{0,0}$, followed by $M_{0,1}$ and $N_{1,0}$ followed by $M_{0,2}$ and $N_{2,0}$, followed by $M_{0,3}$ and $N_{3,0}$.

Fig. 5.6 shows the global memory accesses done by all threads in $block_{0,0}$. The threads are listed in the vertical direction, with time of access increasing from left to right in the horizontal direction. Note that each thread accesses four elements of M and four elements of N during its execution. Among the four threads highlighted, there is a significant overlap in the M and N elements they access. For example, $thread_{0,0}$ and $thread_{0,1}$ both access $M_{0,0}$ as well as the rest of row 0 of M. Similarly, $thread_{0,1}$ and $thread_{1,1}$ both access $N_{0,1}$ as well as the rest of column 1 of N.

The kernel in Fig. 3.13 is written so that both $thread_{0,0}$ and $thread_{0,1}$ access row 0 elements of M from the global memory. If we can somehow manage to have $thread_{0,0}$ and $thread_{0,1}$ to collaborate so that these `M` elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In fact, we can see that every M and N element is accessed exactly twice during the execution of $block_{0,0}$. Therefore, if we can have all four threads to collaborate in their accesses to global memory, we can reduce the overall traffic to the global memory by half.

The reader should verify that the potential reduction in global memory traffic in the matrix multiplication example is proportional to the size of the output tile that the block is assigned to. With `TILE_WIDTH`×`TILE_WIDTH` tiles, the potential reduction of global memory traffic would be `TILE_WIDTH`. That is, if we use 32×32 output tiles, one can potentially reduce the global memory traffic to 1/32 of the original level through collaboration between threads.

We now present a tiled matrix-multiplication algorithm. The basic idea is to have the threads collaboratively load subsets of the M and N elements into the shared memory before they individually use these elements in their dot product calculation. Keep in mind that the size of the shared memory is quite small, and one must be careful not to exceed the capacity of the shared memory when loading these M and N elements into the shared memory. This can be accomplished by dividing the M

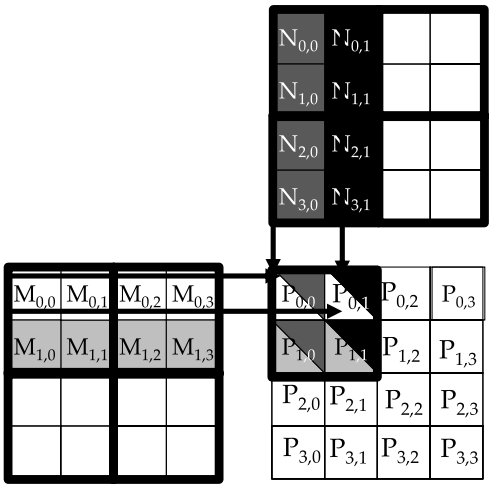


FIGURE 5.7

Tiling `M` and `N` to utilize shared memory.

and N matrices into smaller input tiles. The size of these tiles is chosen so that they can fit into the shared memory. In the simplest form, the input tile dimensions and output tile dimensions are equal to those of the block dimensions, as illustrated in Fig. 5.7.

In Fig. 5.7, we divide M and N into 2×2 input tiles, as delineated by the thick lines. The dot product calculations performed by each thread are now divided into phases. In each phase, all threads in a block collaborate to load a tile of M and a tile of N into the shared memory. This can be done by having every thread in a block to load one M element and one N element into the shared memory, as illustrated in Fig. 5.8. Each row of Fig. 5.8 shows the execution activities of a thread. Note that time progresses from left to right. We only need to show the activities of threads in $block_{0,0}$; the other blocks all have the same behavior. The shared memory array for the M elements is called Mds. The shared memory array for the N elements is called Nds. At the beginning of Phase 1, the four threads of $block_{0,0}$ collaboratively load a tile of M into shared memory: $thread_{0,0}$ loads $M_{0,0}$ into $Mds_{0,0}$, $thread_{0,1}$ loads $M_{0,1}$ into $Mds_{0,1}$, $thread_{1,0}$ loads $M_{1,0}$ into $Mds_{1,0}$, and $thread_{1,1}$ loads $M_{1,1}$ into $Mds_{1,1}$. These loads are shown in the second column of Fig. 5.8. A tile of N is also loaded in a similar manner, shown in the third column of Fig. 5.8.

After the two tiles of M and N are loaded into the shared memory, these elements are used in the calculation of the dot product. Note that each value in the shared memory is used twice. For example, the $M_{1,1}$ value, loaded by $thread_{1,1}$ into $Mds_{1,1}$, is used twice, once by $thread_{1,0}$ and once by $thread_{1,1}$. By loading each global memory value into shared memory so that it can be used multiple times, we reduce the number of accesses to the global memory. In this case, we reduce the number of

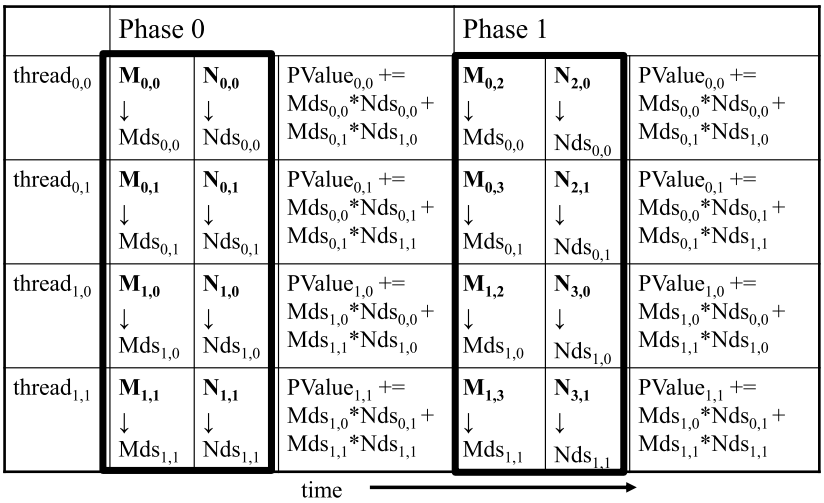


FIGURE 5.8

Execution phases of a tiled matrix multiplication.

accesses to the global memory by a factor of two. The reader should verify that the reduction is by a factor of N if the tiles are $N \times N$ elements.

Note that the calculation of each dot product is now performed in two phases, shown as Phase 1 and Phase 2 in Fig. 5.8. In each phase, each thread accumulates products of two pairs of the input matrix elements into the `Pvalue` variable. Note that `Pvalue` is an automatic variable, so a private version is generated for each thread. The subscripts clarify that these are different instances of the `Pvalue` variable created for each thread. The first phase calculation is shown in the fourth column of Fig. 5.8, and the second phase in the seventh column. In general, if an input matrix is of dimension `Width` and the tile size is `TILE_WIDTH`, the dot product would be performed in `Width/TILE_WIDTH` phases. The creation of these phases is key to the reduction of accesses to the global memory. With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy their overlapping input needs in the phase.

Note also that Mds and Nds are re-used across phases. In each phase, the same Mds and Nds are reused to hold the subset of M and N elements used in the phase. This allows a much smaller shared memory to serve most of the accesses to global memory. This is because each phase focuses on a small subset of the input matrix elements. Such focused access behavior is called locality. When an algorithm exhibits locality, there is an opportunity to use small, high-speed memories to serve most of the accesses and remove these accesses from the global memory. Locality is as important for achieving high-performance in multi-core CPUs as in many-thread GPUs. We will return to the concept of locality in Chapter 6.

```
01    #define TILE_WIDTH 16
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07        int bx = blockIdx.x;  int by = blockIdx.y;
08        int tx = threadIdx.x; int ty = threadIdx.y;
09
10        // Identify the row and column of the P element to work on
11        int Row = by * TILE_WIDTH + ty;
12        int Col = bx * TILE_WIDTH + tx;
13
14        // Loop over the M and N tiles required to compute P element
15        float Pvalue = 0;
16        for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18            // Collaborative loading of M and N tiles into shared memory
19            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20            Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21            __syncthreads();
22
23            for (int k = 0; k < TILE_WIDTH; ++k) {
24                Pvalue += Mds[ty][k] * Nds[k][tx];
25            }
26            __syncthreads();
27
28        }
29        P[Row*Width + Col] = Pvalue;
30
31    }
```

FIGURE 5.9

A tiled Matrix Multiplication Kernel using shared memory.

5.4 A tiled matrix multiplication kernel

We are now ready to present a tiled matrix multiplication kernel that uses shared memory to reduce traffic to the global memory. The kernel shown in Fig. 5.9 implements the phases illustrated in Fig. 5.8. In Fig. 5.9, line 04 and line 05 declare `Mds` and `Nds` as shared memory arrays. Recall that the scope of shared memory variables is a block. Thus, one version of the `Mds` and `Nds` arrays will be created for each block and all threads of a block have access to the same `Mds` and `Nds` version. This is important since all threads in a block must have access to the `M` and `N` elements loaded into `Mds` and `Nds` by their peers so that they can use these values to satisfy their input needs.

Lines 07 and 08 save the `threadIdx` and `blockIdx` values into automatic variables with shorter names to make the code more concise. Recall that automatic scalar variables are placed into registers. Their scope is in each individual thread. That is, one private version of `tx`, `ty`, `bx`, and `by` is created for each thread and will reside in registers that are accessible by the thread. They are initialized with the `threadIdx` and `blockIdx` values and used many times during the lifetime of thread. Once the thread ends, the values of these variables cease to exist.

Lines 11 and 12 determine the row index and column index of the `P` element that the thread is to produce. The code assumes that each thread is responsible for calculating one `P` element. As shown in line 12, the horizontal (x) position, or the

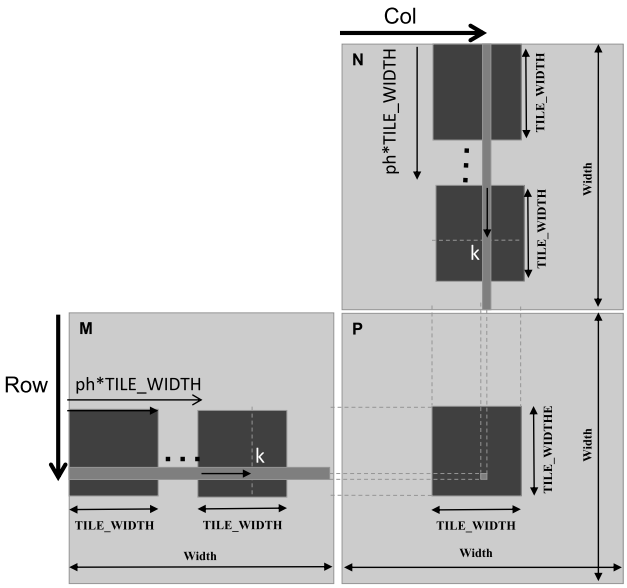


FIGURE 5.10

Calculation of the matrix indices in tiled multiplication.

column index of the `P` element to be produced by a thread, can be calculated as `bx*TILE_WIDTH+tx`. This is because each block covers `TILE_WIDTH` elements of `P` in the horizontal dimension. A thread in block `bx` would have before it `bx` blocks of threads, or `bx*TILE_WIDTH` threads; they cover `bx*TILE_WIDTH` elements of `P`. Another `tx` threads within the same block would cover another `tx` elements. Thus, the thread with indices `bx` and `tx` should be responsible for calculating the `P` element whose x index is `bx*TILE_WIDTH+tx`. For the example in Fig. 5.7, the horizontal (x) index of the `P` element to be calculated by $thread_{0,1}$ of $block_{1,0}$ is 0*2+1 = 1. This horizontal index is saved in the variable `Col` for the thread and is also illustrated in Fig. 5.10.

Similarly, the vertical (y) position, or the row index, of the `P` element to be processed by a thread is calculated as `by*TILE_WIDTH+ty`. Going back to the example in Fig. 5.7, the y index of the `P` element to be calculated by $thread_{0,1}$ of $block_{1,0}$ is 1*2+0 = 2. This vertical index is saved in the variable `Row` for the thread. As shown in Fig. 5.10, each thread calculates the `P` element at the $\texttt{Col}^{th}$ column and the $\texttt{Row}^{th}$ row. Thus, the `P` element to be calculated by $thread_{0,1}$ of $block_{1,0}$ is $P_{2,1}$.

Line 16 of Fig. 5.9 marks the beginning of the loop that iterates through all the phases of calculating the `P` element. Each iteration of the loop corresponds to one phase of the calculation shown in Fig. 5.8. The `ph` variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of `M` and one tile of `N` elements. Therefore, at the beginning of each phase, `ph*TILE_WIDTH` pairs of `M` and `N` elements have been processed by previous phases.

In each phase, lines 19 and 20 in Fig. 5.9 load the appropriate `M` and `N` elements into the shared memory. Since we already know the row of `M` and column of `N` to be processed by the thread, we now turn our focus to the column index of `M` and row index of `N`. As shown in Fig. 5.10, each block has `TILE_WIDTH`^{2} threads that will collaborate to load `TILE_WIDTH`^{2} elements of `M` and `TILE_WIDTH`^{2} elements of `N` into the shared memory. Thus, all we need to do is to assign each thread to load one `M` element and one `N` element. This is conveniently done using the `blockIdx` and `threadIdx`. Note that the beginning column index of the section of `M` elements to be loaded is `ph*TILE_WIDTH`. Therefore, an easy approach is to have every thread load an element that is `tx` positions away from that beginning point. Similarly, the beginning row index of the section of `N` elements to be loaded is also `ph*TILE_WIDTH`. Therefore, every thread loads an element that is `ty` positions away from that beginning point.

This is precisely what we have in lines 19 and 20. In line 19, each thread loads `M[Row*Width + ph*TILE_WIDTH + tx]`, where the linearized index is formed with the row index `Row` and column index `ph*TILE_WIDTH + tx`. Since the value of `Row` is a linear function of `ty`, each of the `TILE_WIDTH`^{2} threads will load a unique `M` element into the shared memory because each thread has a unique combination of `tx` and `ty`. Together, these threads will load a dark square subset of `M` in Fig. 5.10. In a similar way, on Line 20, each thread loads the appropriate `N` element to shared memory using the linearized index `(ph*TILE_WIDTH + ty)*Width + Col`. The reader should use the small example in Fig. 5.7 and Fig. 5.8 to verify that the address calculation works correctly for individual threads.

The barrier `__syncthreads()` in line 21 ensures that all threads have finished loading the tiles of `M` and `N` into `Mds` and `Nds` before any of them can move forward. Recall from Chapter 4 that the call to `__syncthreads()` can be used to make all threads in a block wait for each other to reach the barrier before any of them can proceed. This is important because the `M` and `N` elements to be used by a thread can be loaded by other threads. One needs to ensure that all elements are properly loaded into the shared memory before any of the threads start to use the elements. The loop in line 23 then performs one phase of the dot product based on the tile elements. The progression of the loop for $thread_{ty,tx}$ is shown in Fig. 5.10, with the access direction of the `M` and `N` elements along the arrow marked with `k`, the loop variable in line 23. Note that these elements will be accessed from `Mds` and `Nds`, the shared memory arrays holding these `M` and `N` elements. The barrier `__syncthreads()` in line 26 ensures that all threads have finished using the `M` and `N` elements in the shared memory before any of them move on to the next iteration and load the elements from the next tiles. This way, none of the threads would load the elements too early and corrupt the input values of other threads.

The two `__syncthreads()` calls on lines 21 and 26 demonstrate two different types of data dependences that parallel programmers often have to reason about when coordinating between threads. The first is called a *read-after-write* dependence because threads must wait for data to be written to the proper place by other threads before they try to read it. The second is called a *write-after-read dependence* because a thread must wait for the data to be read by all threads that need it before overwriting it. Other names for read-after-write and write-after-read dependences are true and

false dependences, respectively. A read-after-write dependence is a *true dependence* because the reading thread truly needs the data supplied by the writing thread, so it has no choice but to wait for it. A write-after-read dependence is a *false dependence* because the writing thread does not need any data from the reading thread. The dependence is caused by the fact that they are reusing the same memory location and would not exist if they used different locations. We will revisit these types of dependences in Chapter 6.

The loop nest from line 16 to line 28 illustrates a technique called strip-mining, which takes a long-running loop and breaks it into phases. Each phase consists of an inner loop that executes a few consecutive iterations of the original loop. The original loop becomes an outer loop whose role is to iteratively invoke the inner loop so that all the iterations of the original loop are executed in their original order. By adding barrier synchronizations before and after the inner loop, we force all threads the same block to all focus their work on a section of their input data during each phase. Strip-mining is an important means to creating the phases needed by tiling in data parallel programs.[1]

After all phases of the dot product are complete, the execution exits the outer loop. In Line 29, all threads write to their `P` element using the linearized index calculated from `Row` and `Col`.

The benefit of the tiled algorithm is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of `TILE_WIDTH`. With 32×32 tiles, one can reduce the global memory accesses by a factor of 32. This increases the compute-to-global-memory-access ratio from 0.25 FLOP/B to 8 FLOP/B. This improvement brings the ratio closer to the H100 GPU's threshold of 20 FLOP/B, but keeps the kernel in the memory-bound range. With this improvement, the memory bandwidth of a CUDA device can support a higher computation rate. For example, in the H100 GPU which has a global memory bandwidth of 3.35 TB/s, this improvement allows the device to achieve (3.35 TB/s)*(8 FLOP/B) = 26.8 TFLOPS, which is substantially higher than the 0.84 TFLOPS achieved by the kernel that did not use tiling. In practice, the ratio and compute throughput may be even higher because of the help of the on-chip caches.

Although tiling improves throughput substantially, 26.8 TFLOPS is still only 40% of the device's peak throughput of 66.9 TFLOPS. To push the ratio above 20 FLOP/B into the compute-bound range and improve compute throughput, one can further optimize the code to reduce the number of global memory accesses. We will see some of these optimizations in Chapter 15. Due to the importance of matrix multiplication in many domains, there are highly optimized libraries such as cuBLAS [1] and CUTLASS [2] that already incorporate many of these advanced optimizations. Programmers can use these libraries to immediately achieve close to peak performance on their linear algebra applications.

[1] The reader should note that strip-mining has long been used in programming CPUs. Strip-mining followed by loop interchange is often used to enable tiling for improved locality in sequential programs. Strip-mining is also the main vehicle for vectorizing compilers to generate vector or SIMD instructions for CPU programs.

Note that *tiling* as an optimization does not only refer to placing frequently accessed subset of data in shared memory. It is a general optimization that refers to placing frequently accessed subset of data in a memory that is faster than where the data previously resides. In our tiled matrix multiplication example, we tiled the `M` and `N` matrices in shared memory because the same data elements are repeatedly accessed by different threads in the same block. However, we also applied tiling to the `P` matrix. Since the same element of `P` is repeatedly accessed by the same thread, we placed it in a register (`Pvalue`) and repeatedly accessed it from the register. We refer to this kind of tiling as *register tiling*. In this particular example, the application of register tiling seems trivial. We will see less trivial examples of register tiling in Chapter 8, Chapter 11, and Chapter 15.

The effectiveness of tiling at improving the throughput of matrix multiplication in particular, and applications in general, is not unique to GPUs. There is a long history of applying tiling (or blocking) techniques to improve performance on CPUs by ensuring that the data reused by a CPU thread within a particular time window will be found in the cache. One key difference is that tiling techniques on CPUs rely on the CPU cache to keep reused data on-chip implicitly, whereas tiling techniques on GPUs use shared memory and registers explicitly to keep the data on-chip. The reason is that a CPU core typically runs one or two threads at a time, so a thread can rely on the cache keeping recently used data around. In contrast, a GPU SM runs many threads simultaneously to be able to hide latency. These threads may compete for cache slots which makes the GPU cache less reliable, necessitating the use of shared memory for important data that is to be reused.

While the performance improvement of the tiled matrix multiplication kernel is impressive, it does make a few simplifying assumptions. First, the width of the matrices is assumed to be a multiple of the width of thread blocks. This prevents the kernel from correctly processing matrices with arbitrary width. The second assumption is that the matrices are square matrices. This is not always true in practice. In the next section, we will present a kernel with boundary checks that remove these assumptions.

5.5 Boundary checks

We now extend the tiled matrix multiplication kernel to handle matrices with arbitrary width. The extensions will have to allow the kernel to correctly handle matrices whose width is not a multiple of the tile width. Let's change the small example in Fig. 5.7 to use M, N, and P matrices that have dimensions of 3×3. The revised example is shown in Fig. 5.11. Note that the width of the matrices is 3, which is not a multiple of the tile width (2). Fig. 5.11 shows the memory access pattern during the second phase of $block_{0,0}$. We see that $thread_{0,1}$ and $thread_{1,1}$ attempt to load `M` elements that do not exist. Similarly, we see that $thread_{1,0}$ and $thread_{1,1}$ attempt to access `N` elements that do not exist.

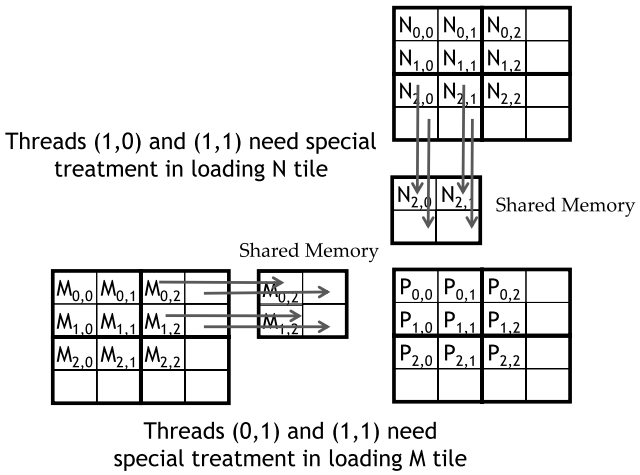


FIGURE 5.11

Loading input matrix elements that are close to the edge – phase 1 of $block_{0,0}$.

Accessing non-existing elements is problematic in two ways. In the case of accessing a non-existing element that are past the end of a row (M accesses by $thread_{0,1}$ and $thread_{1,1}$ in Fig. 5.11), these accesses will be done to incorrect elements. In our example, the threads will attempt to access $M_{0,3}$ and $M_{1,3}$, which do not exist. So, what will happen to these memory loads? To answer this question, we need to go back to the linearized layout of 2D matrices. The element after $M_{0,2}$ in the linearized layout is $M_{1,0}$. Although $thread_{0,1}$ is attempting to access $M_{0,3}$, it will end up getting $M_{1,0}$. The use of this value in the subsequent inner product calculation will obviously corrupt the output value.

A similar problem arises when accessing an element past the end of a column (N accesses by $thread_{1,0}$ and $thread_{1,1}$ in Fig. 5.11). These accesses are to memory locations outside the allocated area for the array. In some systems, they will return random values from other data structures. Other systems will reject these accesses and cause the program to abort. Either way, the outcome of such accesses is undesirable.

From our discussion so far, it may seem that the problematic accesses only arise in the last phase of execution of the threads. This would suggest that we can deal with it by taking special actions during the last phase of the tiled kernel execution. Unfortunately, this is not true. Problematic accesses can arise in all phases. Fig. 5.12 shows the memory access pattern of $block_{1,1}$ during phase 0. We see that $thread_{1,0}$ and $thread_{1,1}$ attempt to access non-existing M elements $M_{3,0}$ and $M_{3,1}$ whereas $thread_{0,1}$ and $thread_{1,1}$ attempt to access $N_{0,3}$ and $N_{1,3}$, which do not exist.

Note that these problematic accesses cannot be prevented by simply excluding the threads that do not calculate valid P elements. For example, $thread_{1,0}$ in $block_{1,1}$ does not calculate any valid P element. However, it needs to load $M_{2,1}$ during phase 0 for other threads in $block_{1,1}$ to use. Furthermore, note that some threads that calculate valid P elements will attempt to access M or N elements that do not exist.

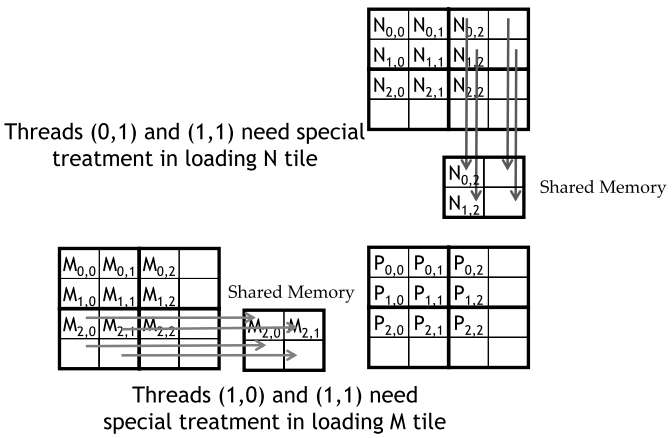


FIGURE 5.12

Loading input elements during phase 0 of $block_{1,1}$.

For example, as we have seen in Fig. 5.11, $thread_{0,1}$ of $block_{0,0}$ calculates a valid P element $P_{0,1}$. However, it attempts to access a non-existing $M_{0,3}$ during phase 1. These two facts indicate that we will need to use different boundary condition tests for loading M tiles, loading N tiles, and calculating/storing P elements. A rule of thumb to follow is that every memory access needs to have a corresponding check that ensures that the indices used in the access are within the bounds of the array being accessed.

Let's start with the boundary test condition for loading input tiles. When a thread is to load an input tile element, it should test whether the input element it is attempting to load is a valid element. This is easily done by examining the y and x indices. For example, at Line 19 in Fig. 5.9, the linearized index is a derived from a y index of `Row` and an x index of `ph*TILE_WIDTH + tx`. The boundary condition test would be that both of indices are smaller than `Width`: `Row < Width && (ph*TILE_WIDTH+tx) < Width`. If the condition is true, the thread should go ahead and load the `M` element. The reader should verify that the condition test for loading the `N` element is `(ph*TILE_WIDTH+ty) < Width && Col < Width`.

If the condition is false, the thread should not load the element. The question is what should be placed into the shared memory location. The answer is zero (`0.0f`), a value that will not cause any harm if it is used in the inner product calculation. If any thread uses this zero value in the calculation of its inner product, there will not be any change in the inner product value.

Finally, a thread should only store its final inner product value if it is responsible for calculating a valid `P` element. The test for this condition is `(Row < Width) && (Col < Width)`. The kernel code with the additional boundary condition checks is shown in Fig. 5.13.

With the boundary condition checks, the tiled matrix multiplication kernel is just one more step away from being a general matrix multiplication kernel. In general,

```
14    // Loop over the M and N tiles required to compute P element
15    float Pvalue = 0;
16    for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ++ph) {
17
18        // Collaborative loading of M and N tiles into shared memory
..        if ((Row < Width) && (ph*TILE_WIDTH+tx) < Width)
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
..        else Mds[ty][tx] = 0.0f;
..        if ((ph*TILE_WIDTH+ty) < Width && Col < Width)
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
..        else Nds[ty][tx] = 0.0f;
21        __syncthreads();
22
23        for (int k = 0; k < TILE_WIDTH; ++k) {
24            Pvalue += Mds[ty][k] * Nds[k][tx];
25        }
26        __syncthreads();
27
28    }
..    if (Row < Width) && (Col < Width)
29        P[Row*Width + Col] = Pvalue;
```

FIGURE 5.13

Tiled matrix multiplication kernel with boundary condition checks.

matrix multiplication is defined for rectangular matrices: a $j \times k$ matrix `M` multiplied with a $k \times l$ matrix `N` results in a $j \times l$ matrix `P`. Our kernel can only handle square matrices so far.

Fortunately, it is quite easy to further extend our kernel into a general matrix multiplication kernel. We need to make a few simple changes. First, the `Width` argument is replaced by three unsigned integer arguments: `j`, `k`, and `l`. Where `Width` is used to refer to the height of `M` or height of `P`, replace it with `j`. Where `Width` is used to refer to the width of `M` or height of `N`, replace it with `k`. Where `Width` is used to refer to the width of `N` or width of `P`, replace it with `l`. The revision of the kernel with these changes is left as an exercise.

5.6 Impact of memory usage on occupancy

Recall that in Chapter 4, we discussed the importance of maximizing the occupancy of threads on SMs to be able to tolerate long latency operations. The memory usage of a kernel plays an important role in occupancy tuning. While CUDA registers and shared memory can be extremely effective at reducing the number of accesses to global memory, one must be mindful of the SM's capacity of these memories. Each CUDA device offers limited memory resources, which limit the number threads that can simultaneously reside in the SM for a given application. In general, the more resources each thread requires, the fewer the number of threads that can reside in each SM. Hence, programmers must manage the delicate balance between the benefit of using on-chip memory resources to increase arithmetic intensity, and the benefit of achieving high occupancy to improve latency tolerance.

We have already seen in Chapter 4 how register usage can be a limiting factor for occupancy. Shared memory usage can also limit the number of threads assigned to each SM. For example, the H100 GPU can be configured to have up to 228 KB of shared memory per SM, and supports a maximum of 2048 threads per SM. Thus, for all 2048 thread slots to be used, a thread block should not use more than an average of (228 KB)/(2048 threads) = 114 B/thread. In the tiled matrix multiplication example, every block has `TILE_WIDTH`^{2} threads, and uses `TILE_WIDTH`^{2}*4 B of shared memory for `Mds` and `TILE_WIDTH`^{2}*4 B of shared memory for `Nds`. Thus, the thread block uses an average of (`TILE_WIDTH`^{2}*4 B + `TILE_WIDTH`^{2}*4 B)/(`TILE_WIDTH`^{2} threads) = 8 B/thread of shared memory. Therefore, the tiled matrix multiplication kernel's occupancy is not limited by the shared memory.

However, consider if a kernel has thread blocks that use 38 KB of shared memory, and each have 256 threads. In this case, the kernel uses an average of (38 KB)/(256 threads) = 152 B/thread of shared memory. With such shared memory usage, the kernel cannot achieve full occupancy. Each SM can only host a maximum of (228 KB)/(152 B/thread) = 1536 threads. Therefore, the maximum achievable occupancy of this kernel will be (1536 assigned threads)/(2048 maximum threads) = 75%.

For kernels that need to use a very large amount of shared memory, CUDA provides the possibility of reconfiguring the SM to use more resources for the shared memory at the expense of other on-chip memory resources, particularly the cache. This configurability enables the coverage of both predictable and less predictable access patterns to benefit from on-chip memory, allowing programmers to apportion the on-chip memory resources according to the best fit for their application.

Note that the size of shared memory in each SM can also vary from device to device. Each generation or model of devices can have a different amount of shared memory in each SM. It is often desirable for a kernel to be able to use different amounts of shared memory according to the amount available in the hardware or the amount needed by the program. That is, we may want a host code to dynamically determine the size of the shared memory and adjust the amount of shared memory used by a kernel. This can be done by calling the `cudaGetDeviceProperties` function. Assume that variable `&devProp` is passed to the function. In this case, the field `devProp.sharedMemPerBlock` gives the amount of shared memory available in each SM. The programmer can then determine the amount of shared memory that should be used by each block.

The kernels in Fig. 5.9 and Fig. 5.13 do not support any dynamic adjustment of shared memory usage by the host code. The declarations used in Fig. 5.9 hardwire the size of its shared memory usage to a compile-time constant:

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

That is, the size of `Mds` and `Nds` is set to be `TILE_WIDTH`^{2} elements, whatever the value of `TILE_WIDTH` is set to be at compile time. Since the code contains:

```
#define TILE_WIDTH 32
```

```
01    #define TILE_WIDTH 32
02    __global__ void matrixMulKernel(float* M, float* N, float* P, int Width,
                                      unsigned Mdz_sz, unsigned Nds_sz) {
03
04       extern __shared__ char float Mds_Nds[];
05
06     float *Mds = (float *) Mds_Nds;
07     float *Nds = (float *) Mds_Nds + Mds_sz;
```

FIGURE 5.14

Tiled matrix multiplication kernel with dynamically sized shared memory usage.

Both `Mds` and `Nds` will have $32^2 = 1024$ elements. If we want to change the size of `Mds` and `Nds`, we need to change the value of `TILE_WIDTH` and recompile the code. The kernel cannot easily adjust its shared memory usage at runtime without recompilation.

We can enable dynamic adjustment of the shared memory size with a different style of declaration in CUDA by adding the `extern` keyword in front of the shared memory declaration and omitting the size of the array in the declaration. Based on this style, the declarations for `Mds` and `Nds` need to be merged into one dynamically allocated array as follows:

```
extern __shared__ Mds_Nds[];
```

Since there is only one merged array, we will also need to manually define where the `Mds` section of the array starts and where the `Nds` section starts. Note that the merged array is one-dimensional. We will need to access it using a linearized index based on the vertical and horizontal indices.

At run time, when we call the kernel, we can dynamically configure the amount of shared memory to be used for each block as a third configuration parameter to the kernel call. For example, the revised kernel could be launched with the following statements:

```
size_t size = ...;
matrixMulKernel <<< dimGrid, dimBlock, size >>>
    (Md, Nd, Pd, Width, size/2, size/2);
```

where `size_t` is a built-in type for declaring a variable to hold the size information for dynamically allocated data structures. The size is expressed in number of bytes. In our matrix multiplication example, for a 32×32 tile, we have size to be $2 \times 32 \times 32 \times 4 = 8,192$ bytes to accommodate both `Mds` and `Nds`. We have omitted the details of the calculation for setting the value of `size` at run time and leave it as an exercise for the reader.

In Fig. 5.14, we show how one can modify the kernel code in Figs. 5.9 and 5.11 to use dynamically sized shared memory for the `Mds` and `Nds` arrays. It may also be useful to pass the sizes of each section of the array as arguments into the kernel function. In this example, we added two arguments: the first one is the size of the `Mds` section and the second is the size of the `Nds` section, both in terms of bytes. Note that the in the host code above, we pass `size/2` as the values of these arguments,

which is 1024 bytes. With the assignments in lines 06 and 07, the rest of the kernel code can use `Mds` and `Nds` as the base of the array and use a linearized index to access the `Mds` and `Nds` elements. For example, instead of using `Mds[ty][tx]`, one would use `Mds[ty*TILE_WIDTH+tx]`.

5.7 Summary

In summary, the execution speed of a program in modern processors depends on the compute-to-global-memory-access ratio in the kernel code. If the ratio is high, the kernel is compute-bound and its execution speed can be close the peak computational throughput of the hardware. If the ratio is low, the kernel is memory-bound and its execution speed is limited by the rate at which its operands are accessed from memory.

CUDA provides access to registers, shared memory, and constant memory. These memories are much smaller than the global memory but can be accessed at much higher speed. Placing data in these memories allows them to be accessed without consuming global memory bandwidth, and thus elevating the compute-to-global-memory-access ratio for the code. However, using these memories effectively requires re-design of the algorithm. We use matrix multiplication as an example to illustrate tiling, a popular strategy to enhance locality of data access and enable effective use of shared memory. In parallel programming, tiling uses barrier synchronization to force multiple threads to jointly focus on a subset of the input data at each phase of the execution so that the subset data can be placed into these special memory types to enable much higher access speed.

It is, however, important for CUDA programmers to be aware of the limited sizes of these special types of memory. Their capacities are implementation dependent. Once their capacities are exceeded, they limit the number of threads that can be simultaneously executing in each SM and can negatively impact the GPU's computation throughput as well as its ability to tolerate latency. The ability to reason about hardware limitations when developing an application is a key aspect of parallel programming.

Although we introduced tiled algorithms in the context of CUDA programming for GPUs, it is an effective strategy for achieving high-performance in virtually all types of parallel computing systems. The reason is that an application must exhibit locality in data access to make effective use of high-speed memories in these systems. For example, in a multi-core CPU system, data locality allows an application to effectively use on-chip data caches to reduce memory access latency and achieve high performance. These on-chip data caches are also of limited size and require the computation to exhibit locality. Therefore, the reader will also find the tiled algorithm useful when developing a parallel application for other types of parallel computing systems using other programming models.

Our goal for this chapter is to introduce the concept of locality, tiling, and different CUDA memory types. We introduced a tiled matrix multiplication kernel using

shared memory. We further studied the need for boundary test conditions to allow for arbitrary data dimensions when applying tiling techniques. We also briefly discussed the use of dynamically sized shared memory allocation so that the kernel can adjust the size of shared memory used by each block according to the hardware capability.

5.8 Exercises

1. Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: analyze the elements accessed by each thread and see if there is any commonality between threads.
2. Draw the equivalent of Fig. 5.7 for a 8×8 matrix multiplication with 2×2 tiling and 4×4 tiling. Verify that the reduction in global memory bandwidth is indeed proportional to the dimension size of the tiles.
3. What type of incorrect execution behavior can happen if one forgot to use one or both `__syncthreads()` in the kernel of Fig. 5.9?
4. Assuming capacity were not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.
5. For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices `M` and N?
6. Assume that a CUDA kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?
7. In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
8. Consider performing a matrix multiplication of two input matrices with dimensions N×N. How many times is each element in the input matrices requested from global memory when:
   a. There is no tiling?
   b. Tiles of size TxT are used?
9. A kernel performs 36 floating point operations and 7 32-bit global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.
   a. Peak FLOPS = 200 GFLOPS, Peak Memory Bandwidth = 100 GB/s
   b. Peak FLOPS = 300 GFLOPS, Peak Memory Bandwidth = 250 GB/s
10. To manipulate tiles, a new CUDA programmer has written the following device kernel which will transpose each tile in a matrix. The tiles are of size BLOCK_WIDTH by BLOCK_WIDTH, and each of the dimensions of matrix A is known to be a multiple of BLOCK_WIDTH. The kernel invocation and code are shown below. BLOCK_WIDTH is known at compile-time and could be set anywhere from 1 to 20.

```
01  dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
02  dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
03  BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

04  __global__ void
05  BlockTranspose(float* A_elements, int A_width, int A_height)
06  {
07     __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];

08     int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
09     baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;

10     blockA[threadIdx.y][threadIdx.x] = A_elements[baseIdx];

11     A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
12  }
```

a. Out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will this kernel function execute correctly on the device?

b. If the code does not execute correctly for all BLOCK_SIZE values, what is the root cause of this incorrect execution behavior? Suggest a fix to the code to make it work for all BLOCK_SIZE values.

11. Consider the following CUDA kernel and the corresponding host function that calls it:

```
01  __global__ void foo_kernel(float* a, float* b) {
02     unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03     float x[4];
04     __shared__ float y_s;
05     __shared__ float b_s[128];
06     for(unsigned int j = 0; j < 4; ++j) {
07        x[j] = a[j*blockDim.x*gridDim.x + i];
08     }
09     if(threadIdx.x == 0) {
10        y_s = 7.4f;
11     }
12     b_s[threadIdx.x] = b[i];
13     __syncthreads();
14     b[i] = 2.5f*x[0] + 3.7f*x[1] + 6.3f*x[2] + 8.5f*x[3]
15            + y_s*b_s[threadIdx.x] + b_s[(threadIdx.x + 3)%128];
16  }
17  void foo(int* a_d, int* b_d) {
18     unsigned int \texttts{N} = 1024;
19      foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
20  }
```

a. How many versions of the variable i are there?

b. How many versions of the array x[] are there?

c. How many versions of the variable y_s are there?
d. How many versions of the array b_s[] are there?
e. What is the amount of shared memory used per block (in B)?
f. What is the floating point to global memory access ratio of the kernel (in OP/B)?

12. Consider a GPU with the following hardware limits: 2048 threads/SM, 32 blocks/SM, 64 K (65536) registers/SM, and 96 KB of shared memory/SM. For each of the following kernel characteristics, specify if the kernel can achieve full occupancy. If not, specify the limiting factor.
a. The kernel uses 64 threads/block, 27 registers/thread, and 4 KB of shared memory/SM.
b. The kernel uses 256 threads/block, 31 registers/thread, and 8 KB of shared memory/SM.

References

[1] NVIDIA, cuBLAS, http://docs.nvidia.com/cuda/cublas, 2025.
[2] NVIDIA, CUTLASS, https://docs.nvidia.com/cutlass, 2025.

This page intentionally left blank

CHAPTER

Performance considerations

6

CONTENTS

6.1 Global memory access coalescing 124
6.2 Hiding memory latency 132
6.3 Vector loads and stores 138
6.4 Shared memory bank conflicts 139
6.5 Thread coarsening 141
6.6 Loop unrolling 144
6.7 Double buffering 146
6.8 A checklist of optimizations 147
6.9 Optimization strategy 153
6.10 Summary 154
6.11 Exercises 154
References 155

The execution speed of a parallel program can vary greatly depending on the interactions between the resource demands of the program and the resource constraints of the hardware. Managing the interaction between parallel code and hardware resource constraints is important for achieving high performance in virtually all parallel programming models. It is a practical skill that requires deep understanding of the hardware architecture and is best learned with hands-on exercises in a parallel programming model designed for high-performance.

So far, we have learned about the basic aspects of the GPU architecture and their implications on performance. In Chapter 4, we learned about the compute architecture of the GPU and related performance considerations such as control divergence and occupancy. In Chapter 5, we learned about the on-chip memory architecture of the GPU and the use of shared memory tiling and register tiling to alleviate global memory bandwidth limitations. In this chapter, we will learn more about the off-chip memory (DRAM) architecture and discuss related performance considerations such as memory coalescing, memory latency hiding, and vector loads and stores. We also introduce additional important optimizations such as avoiding shared memory bank conflicts, thread granularity coarsening, loop unrolling, and double-buffering. Finally, we wrap up this part of the textbook with a checklist of common performance optimizations that will serve as a guide for optimizing the performance of the parallel patterns discussed in the second and third parts of the book.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00015-9
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

Since different applications impose different demands on architectural resources, different architecture constraints may dominate and become the limiting factors of performance, commonly referred to as bottlenecks. One can often dramatically improve the performance of an application on a particular CUDA device, by trading one resource usage for another. This strategy works well if the resource constraint thus alleviated was the dominating constraint before the strategy was applied and the one thus exacerbated does not have negative effects on parallel execution. Without such understanding, performance tuning would be a guess work; plausible strategies may or may not lead to performance enhancements.

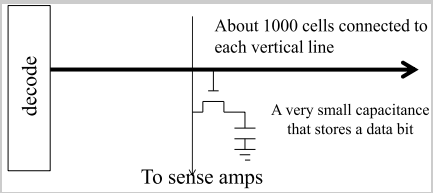
6.1 Global memory access coalescing

One of the most important factors of CUDA kernel performance is accessing data in the global memory whose limited bandwidth can become the bottleneck. CUDA applications exploit massive data parallelism. Naturally, CUDA applications tend to process a massive amount of data from the global memory within a short period of time. In Chapter 5, we studied tiling techniques that leverage the shared memory to reduce the total amount of data that must be accessed from the global memory by a collection of threads in each thread block. In this chapter, we will further discuss memory coalescing techniques for moving data between global memory and shared memories or registers in an efficient manner. Memory coalescing techniques are often used in conjunction with tiling techniques to allow CUDA devices to reach their performance potential by efficiently utilizing the global memory bandwidth.

The global memory of a CUDA device is implemented with DRAM as we saw in Chapter 5. In DRAM, data bits are stored in DRAM cells that are small capacitors, where the presence or absence of a tiny amount of electrical charge distinguishes between a 1 and a 0 value. Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to a sensor and to set off the sensor's detection mechanism that determines whether a sufficient amount of charge is present in the capacitor to qualify as a "1". This process takes tens of nanoseconds in modern DRAM chips (see "Why is DRAM so slow?" sidebar). This is in sharp contrast with the sub-nanosecond clock cycle time of modern computing devices. Because this process is very slow relative to the desired data access speed (sub-nanosecond access per byte), modern DRAM designs use parallelism to increase their rate of data access, commonly referred to as memory access throughput.

Why are DRAMs so slow?

The following figure shows a DRAM cell and the path for accessing its content.



The decoder is an electronic circuit that drives a horizontal line connected to the outlet transistor gates of thousands of cells. It can take a long time for the horizontal line to be fully charged or discharged to the desired level to activate the outlet gate. A more formidable challenge is for the cell, once its outlet gate is activated, to drive the vertical line to the sense amplifiers and allow the sense amplifier to detect its content. This is based on electrical charge sharing. The gate lets out the tiny amount of electrical charge stored in the cell. The tiny amount of charge must cause sufficient change to the electrical potential of the large capacitance of the long bit line in order to trigger the detection mechanism of the sense amplifier. A good analogy would be for someone to hold a small cup of coffee at one end of a long hallway connecting many offices for another person to smell the aroma propagated through the hallway to determine the flavor of the coffee.

One could speed up the process by using a larger, stronger capacitor in each cell. However, the DRAMs have been going in the opposite direction. The capacitors in each cell have been steadily reduced in size and thus reduced in their strength over time so that more bits can be stored in each chip. This is why the access latency of DRAMs has not decreased over time.

Each time a DRAM location is accessed, a range of consecutive locations that include the requested location are accessed as well. Many sensors are provided in each DRAM chip and they all work in parallel. Each sensor senses the content of a bit within these consecutive locations. Once detected by the sensors, the data from all these consecutive locations can be transferred at high speed to the processor. These consecutive locations accessed and delivered are referred to as *DRAM bursts*. If an application makes focused use of data from a burst, the DRAM can supply the data at a much higher rate than if an application accesses a truly random sequence of locations scattered across many bursts.

Modern GPUs also use SRAM-based L1 and L2 on-chip caches to help reduce the latencies and increase the throughput of global memory accesses. Since these caches are not a prominent part of the CUDA programming model, we have not covered these caches. However, their presence and behavior impact the observed performance of CUDA applications. Regarding the global memory access pattern, even when data has already been fetched from the DRAM and placed in the SRAM caches, making focused access to consecutive memory locations is still beneficial. When data in the L2 cache is accessed, a range of consecutive locations are moved from the L2 cache

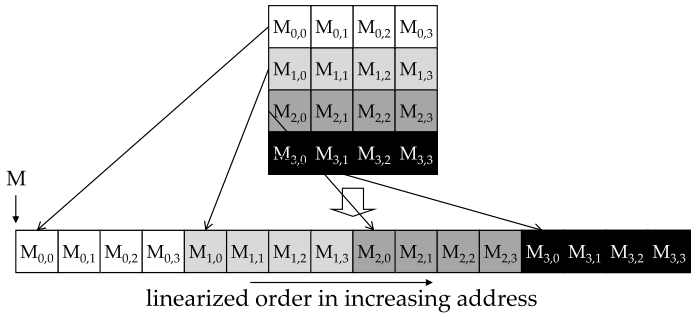


FIGURE 6.1

Placing matrix elements into a linear array based on row major order.

to the L1 cache. This granularity at which data is moved between different levels of cache is known as the *cache block* or *cache line*. If an application makes focused use of data from within a cache line, the caches can supply the data at a faster rate because fewer cache lines need to be moved.

Recognizing the burst organization of modern DRAMs and the cache line organization of modern cache hierarchies, current CUDA devices employ a technique that allows programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations in the same DRAM burst and/or cache line. In this case, the hardware combines, or *coalesces*, all these accesses into a consolidated access. For example, for a given load instruction of a warp, if thread 0 accesses global memory location X, thread 1 accesses location $X + 1$, thread 2 accesses location $X + 2$, and so on, all these accesses will be coalesced, or combined into a larger request for consecutive locations when accessing the DRAM or caches.

Besides ensuring that threads in a warp access consecutive global memory locations, another consideration is the *alignment* of these accesses in global memory. When the starting address of these consecutive global memory locations is aligned to the beginning of a burst or cache line, this alignment ensures that the smallest number of memory transactions are needed to perform the coalesced access. On the other hand, if the starting address is not aligned, the unaligned access could result in additional memory traffic because the accesses data wall span a larger number of bursts or cache lines.

To understand how to effectively use the coalescing hardware, we need to review how the memory addresses are formed in accessing C multi-dimensional array elements. Recall from Chapter 3 (Fig. 3.3, replicated as Fig. 6.1 for convenience) that multi-dimensional array elements in C and CUDA are placed into the linearly addressed memory space according to the row major convention. Recall that row major

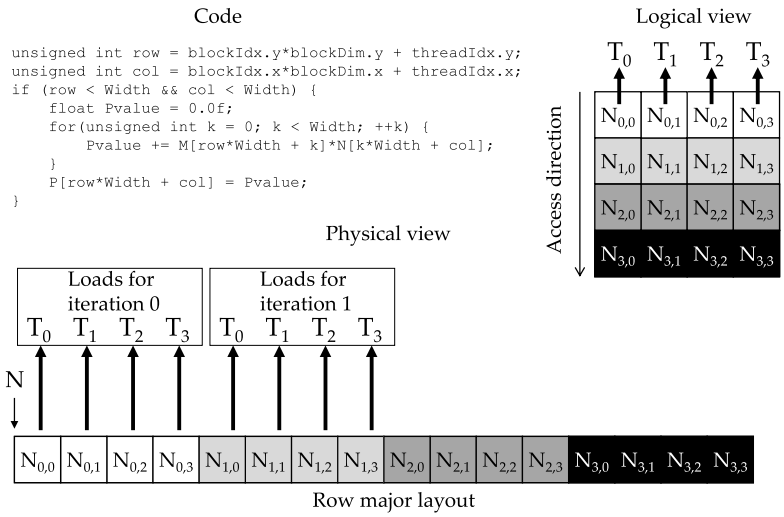


FIGURE 6.2

A coalesced access pattern.

refers to the fact that the placement of data preserves the structure of rows: all adjacent elements in a row are placed into consecutive locations in the address space. In Fig. 6.1, the four elements of row 0 are first placed in their order of appearance in the row. Elements in row 1 are then placed, followed by elements of row 2, followed by elements of row 3. It should be clear that $M_{0,0}$ and $M_{1,0}$, though appear to be consecutive in the two-dimensional matrix, are placed four locations away in the linearly addressed memory.

Let's assume that a multi-dimensional array `N` similar to the array `M` in Fig. 6.1 is used as the second input matrix in matrix multiplication. In this case, consecutive threads in a warp that are assigned to consecutive output elements and will iterate through consecutive columns of this input matrix. The top-left part of Fig. 6.2 shows the code for this computation and the top-right part shows the logical view of the access pattern: consecutive threads iterating through consecutive columns. One can tell by inspecting the code that the accesses to `N` can be coalesced. The index of the array `N` is `k*Width + col`. The variables `k` and `Width` have the same value across all threads in the warp. The variable `col` is defined as `blockIdx.x*blockDim.x+threadIdx.x`, which means that consecutive threads (with consecutive `threadIdx.x` values) will have consecutive values of `col`, and will therefore access consecutive elements of `N`.

The bottom part of Fig. 6.2 shows the physical view of the access pattern. In iteration 0, consecutive threads will access consecutive elements in row 0 which are adjacent in memory, shown as "Loads for iteration 0" in Fig. 6.2. In iteration 1, consecutive threads will access consecutive elements in row 1 which are also adjacent in memory, shown as "Loads for iteration 1" in Fig. 6.2. This process continues for all rows. As we can see, the memory access pattern formed by the threads during this

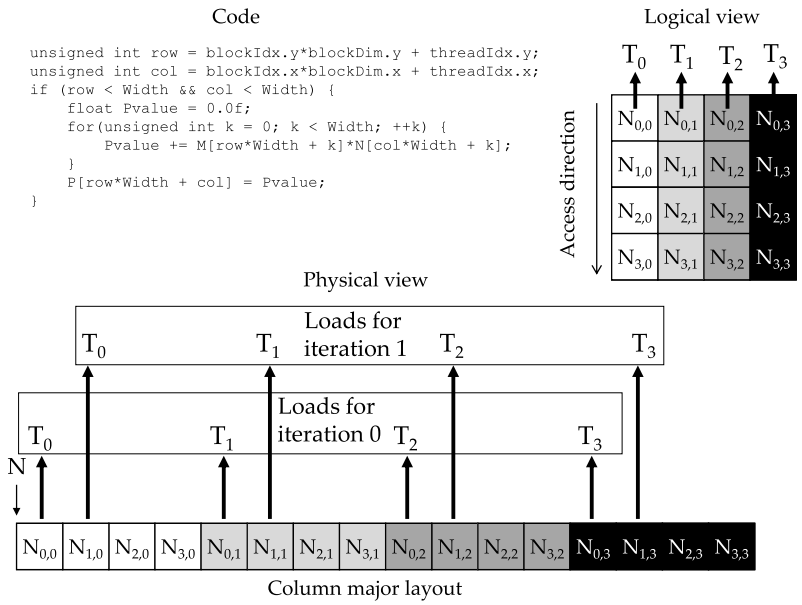


FIGURE 6.3

An un-coalesced access pattern.

process is a favorable one that can be coalesced. Indeed, in all the kernels we have implemented in the book so far, our memory accesses have been naturally coalesced.

Now, assume that the matrix was stored in column-major order instead of row-major order. There could be various reasons why this might be the case. For example, we might be accessing the transpose of a matrix that is stored in row-major order. In linear algebra, we often need to use both the original and transposed forms of a matrix. It would be better to avoid creating and storing both forms. A common practice is to create the matrix in one form, say the original form. When the transposed form is needed, its elements can be accessed by accessing the original form and switching the roles of the row and column indices. This is equivalent to viewing the transposed matrix as being in the column-major layout. Regardless of the reason, let's observe the memory access pattern achieved when the second input matrix to our matrix multiplication example is stored in column-major order.

Fig. 6.3 illustrates how consecutive threads iterate through consecutive columns when the matrix is stored in column-major order. The top-left part of Fig. 6.3 shows the code and the top-right part shows the logical view of the memory accesses. The program is still trying to have each thread to access a column of matrix `N`. However, the index calculation for accessing `N` in the for-loop is adjusted to reflect the assumed column-major layout. One can tell by inspecting the code that the accesses to `N` are not favorable for coalescing. The index of the array `N` is `col*Width+k`. As before, `col` is defined as `blockIdx.x*blockDim.x+threadIdx.x`, which means that consecutive threads

(with consecutive `threadIdx.x` values) will have consecutive values of `col`. However, in the index to `N`, `col` is multiplied by `Width`, which means that consecutive threads will access elements of `N` that are `Width` apart. Therefore, the accesses are not favorable for coalescing.

In the bottom portion of Fig. 6.3, we can see that the physical view of the memory accesses is quite different from that in Fig. 6.2. In iteration 0, consecutive threads will logically access consecutive elements in row 0, but this time they are not adjacent in memory because of the column major layout. These loads are shown as "Loads for iteration 0" in Fig. 6.3. Similarly, in iteration 1, consecutive threads will access consecutive elements in row 1 which are also not adjacent in memory. For a realistic matrix, there are typically hundreds or even thousands of elements in each dimension. The elements of `N` accessed in each iteration by neighboring threads can be hundreds or even thousands of elements apart. The hardware will determine that accesses to these elements are far away from each other and cannot be coalesced.

There are various strategies for optimizing code to achieve memory coalescing when the computation is not naturally amenable to it. One strategy is to rearrange how threads are mapped to the data and another strategy is to rearrange the layout of the data itself. We will discuss these strategies in Section 6.8 and see examples of how they can be applied throughout this book. Yet another strategy is to transfer the data between global memory and shared memory in a coalesced manner and carry out the unfavorable access pattern in shared memory which is less sensitive to the data layout and provides faster access latency. We will also see example optimizations that use this strategy throughout this textbook, including an optimization that we will apply now to matrix-matrix multiplication where the second input matrix is in column-major layout. This optimization is called *corner turning*.

Fig. 6.4 illustrates an example of how corner turning can be applied. In this example, `M` is an input matrix stored in row-major layout in global memory and `N` is an input matrix stored in column-major layout in global memory. They are multiplied to produce an output matrix `P` which is stored in row-major layout in global memory. The example illustrates how four threads that are responsible for the four consecutive elements at the top edge of the output tile load the input tile elements.

The access to the input tile in matrix `M` is similar to that in Chapter 5. The four threads load the four elements at the top edge of the input tile. Each thread loads an input element whose local row and column indices within the input tile are the same as that of the thread's output element within the output tile. These accesses are coalesced because consecutive threads access consecutive elements in the same row of `M` which are adjacent in memory according to the row major layout.

On the other hand, the access to the input tile in matrix `N` needs to be different from that in Chapter 5. Fig. 6.4(a) shows what the access pattern would be like if we use the same arrangement as in Chapter 5. Even though the first four threads in the block are logically loading the four consecutive elements at the top edge of the input tile, the elements loaded by consecutive threads are far away from each other in the memory because of the column major layout of the `N` elements. In other words, consecutive threads that are responsible for consecutive elements in the same row of the output

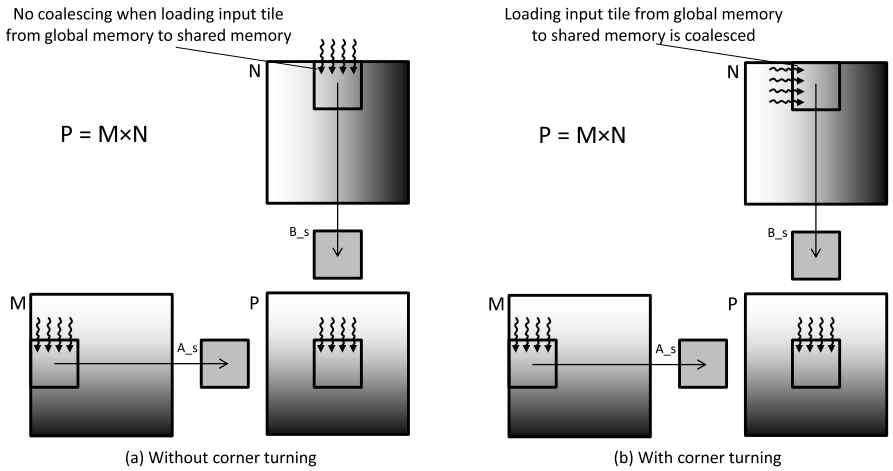


FIGURE 6.4

Applying corner turning to coalesce accesses to matrix `N` which is stored in column-major layout.

tile load non-consecutive locations in memory, which results in uncoalesced memory accesses.

This problem can be solved by assigning the first four consecutive threads of the block to load the four consecutive elements at the left edge (the same column) in the input tile, as shown in Fig. 6.4(b). Intuitively, we are exchanging the roles of `threadIdx.x` and `threadIdx.y` when each thread calculates the linearized index for loading the `N` input tile. Since `N` is in column-major layout, consecutive elements in the same column are adjacent in memory. Hence, consecutive threads load input elements that are adjacent in memory, which ensures that the memory accesses are coalesced. The code can be written to place the tile of `N` elements into the shared memory in either column-major layout or row-major layout. Either way, after the input tile has been loaded, each thread can access its inputs with little performance penalty. This is because shared memory is implemented with SRAM technology and does not require coalescing. Note that we have been using a toy example illustrating the first four threads in the block when accessing 4×4 tiles. In reality, the tile size for corner turning will be the warp size of 32, so the first 32 threads in the block will access 32 consecutive elements on the left edge of the tile.

While our example focuses on enabling the coalescing of global memory load operations, coalescing is also important for global memory store operations. In fact, the impact of uncoalesced stores can be more severe than the impact of uncoalesced loads. The reason is that uncoalesced stores generate partial store operations that perform read-modify-write accesses, which could require more memory bandwidth on GPUs with error correcting code (ECC) enabled.

The main advantage of coalescing memory accesses is that it reduces global memory traffic by combining multiple memory accesses into a single access. Accesses



FIGURE 6.5

Reducing traffic congestion in highway systems.

can be combined when they take place at the same time and access adjacent memory locations. Traffic congestion does not only arise in computing. Most of us have experienced traffic congestion in highway systems, as illustrated in Fig. 6.5. The root cause of highway traffic congestion is that there are too many cars all squeezing through a road that is designed for a much smaller number of vehicles. When congestion occurs, the travel time for each vehicle is greatly increased. Commute time to work can easily double or triple during traffic congestion.

Most solutions for reduced traffic congestion involve reduction of cars on the road. Assuming that the number of commuters is constant, people need to share rides in order to reduce the number of cars on the road. A common way to share rides is carpooling, where a group of commuters take turns to drive the group to work in one vehicle. Governments usually have policies to encourage carpooling. In some countries, the government simply disallows certain classes of cars to be on the road on a daily basis. For example, cars with odd license plates may not be allowed on the road on Monday, Wednesday, or Friday. This encourages people whose cars are allowed on different days to form a carpool group. In other countries, the government may provide incentives for behavior that reduces the number of cars on the road. For example, in certain countries, some lanes of congested highways are designated as carpool lanes; only cars with more than two or three people are allowed to use these lanes. There are also countries where the government makes gasoline so expensive that people form carpools to save money. All these measures for encouraging carpooling are designed to overcome the fact that carpooling requires extra effort, as we show in Fig. 6.6.

Carpooling requires workers who wish to carpool to compromise and agree on a common commute schedule. The top half of Fig. 6.6 shows a good schedule pattern for carpooling. Time goes from left to right. Worker A and Worker B have similar schedule for sleep, work, and dinner. This allows these two workers to easily go to work and return home in one car. Their similar schedules allow them to more easily agree on a common departure time and return time. This is however, not the case of the schedules shown in the bottom half of Fig. 6.6. Worker A and Worker B have very different habit in this case. Worker A parties until sunrise, sleeps during the day, and goes to work in the evening. Worker B sleeps at night, goes to work in the morning,

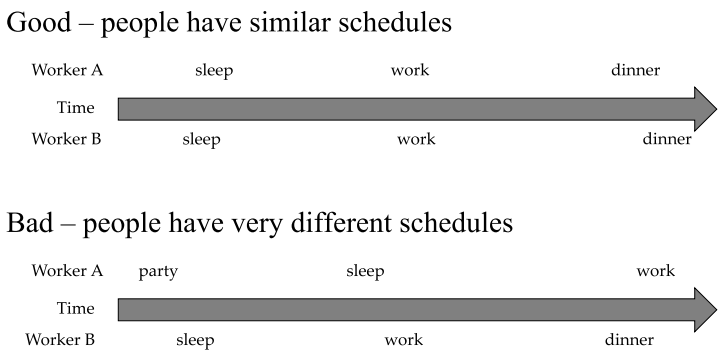


FIGURE 6.6

Carpooling requires synchronization among people.

and returns home for dinner at 6 pm. The schedules are so wildly different that there is no way these two workers can coordinate a common time to drive to work and return home in one car.

Memory coalescing is very similar to carpooling arrangements. We can think of the data as the commuters and the DRAM access requests as the vehicles. When the rate of DRAM requests exceeds the provisioned access throughput of the DRAM system, traffic congestion rises, and the arithmetic units become idle. If multiple threads access data from the same DRAM location, they can potentially form a "carpool" and combine their accesses into one DRAM request. This, however, requires the threads to have similar execution schedule so that their data accesses can be combined into one. Threads in the same warp are the perfect candidates because they all execute a load instruction simultaneously by virtue of SIMD execution.

Note that modern CPUs also recognize the DRAM burst organization in their cache memory design. A CPU cache line typically maps to a fraction of, one, or even multiple DRAM bursts. Applications that make full use of bytes in each cache line they touch tend to achieve much higher performance than those that randomly access memory locations. The techniques presented in this chapter can be adapted to help CPU programs achieve higher speed by reducing the congestion in accessing DRAM.

6.2 Hiding memory latency

As we explained in Section 6.1, DRAM bursting is a form of parallel organization: multiple locations are accessed in the DRAM core array in parallel. However, bursting alone is not sufficient to realize the level of DRAM access bandwidth required by modern processors. DRAM systems typically employ two more forms of parallel organization – banks and channels. At the highest level, a processor contains one or more channels. Each channel is a memory controller with a bus that connects a set of DRAM banks to the processor. Fig. 6.7 illustrates a processor that contains four

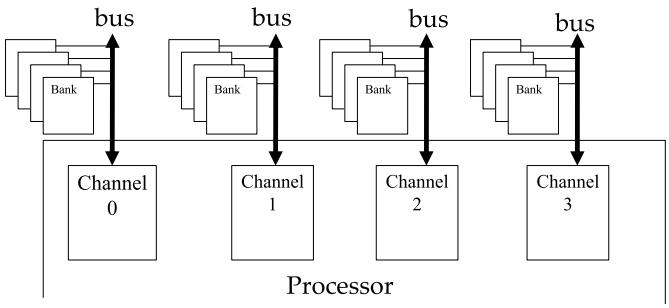


FIGURE 6.7

Channels and banks in DRAM systems.

channels, each with a bus that connects four DRAM banks to the processor. In real systems, a processor typically has one to eight channels and each channel is connected a large number of banks.

The data transfer bandwidth of a bus is defined by its width and clock frequency. Modern double data rate (DDR) buses perform two data transfers per clock cycle, one at the rising edge and one at the falling edge of each clock cycle. For example, a 64-bit DDR5-6400 bus with a clock frequency of 3.2 GHz has a bandwidth of 8 B $\times$ 2 $\times$ 3.2 GHz = 51.2 GB/sec. This seems to be a large number, but is often too small for modern CPUs and GPUs. For example, a modern CPU might require a memory bandwidth of at least 200 GB/s whereas a GPU might require more than 1000 GB/s, which would translate into four buses for the CPU and 20 buses for the GPU. Such large numbers of buses consume too much power when the processor needs to drive the wires on traditional circuit boards and have motivated the adoption of High Bandwidth Memory (HBM) technology. HBM co-packages the processors with DRAM to drastically reduce the size and capacity of connections between the two and enhance the clock frequency as well as the number of buses that connect the processor and the DRAM. For example, HBM3E supports a bus clock frequency of 4.9 GHz/s and 16 64-bit buses, resulting in 1 TB/s memory bandwidth.

For each channel, the number of banks connected to it is determined by the number of banks required to fully utilize the data transfer bandwidth of the bus. This is illustrated in Fig. 6.8. Each bank contains an array of DRAM cells, the sensing amplifiers for accessing these cells, and the interface for delivering bursts of data to the bus (Section 6.1).

Fig. 6.8(a) illustrates the data transfer timing when a single bank is connected to a channel. It shows the timing of two consecutive memory read accesses to the DRAM cells in the bank. Recall from Section 6.1 that each access involves long latency for the decoder to enable the cells and for the cells to share their stored charge with the sensing amplifier. This latency is shown as the light gray section at the left end of the time frame. Once the sensing amplifier completes its work, the burst data is delivered through the bus. The time for transferring the burst data through the bus is shown as the left dark section of the time frame in Fig. 6.8(a). The second memory read access

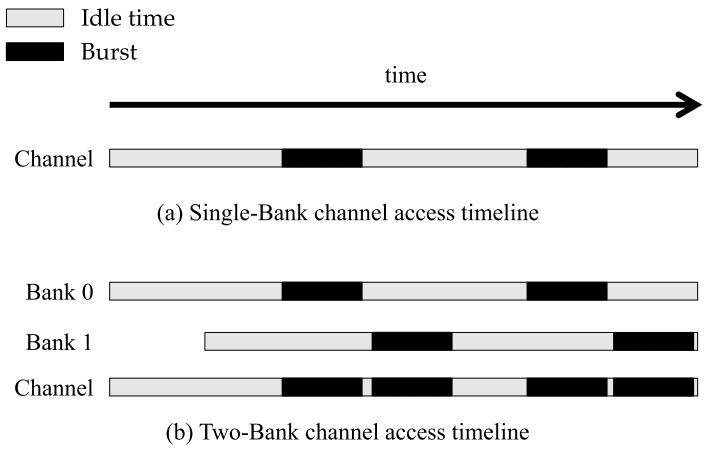


FIGURE 6.8

Banking improves the utilization of data transfer bandwidth of a channel.

will incur a similar long access latency (light section between the dark sections of the time frame) before its burst data can be transferred (right dark section).

In reality, the access latency (light gray sections) is much longer than the data transfer time (dark section). It should be apparent that the access-transfer timing of a one-bank organization would grossly underutilize the data transfer bandwidth of the channel bus. For example, if the ratio of DRAM cell array access latency to the data transfer time is 20:1, the maximal utilization of the channel bus would be 1/21 = 4.8%. That is a 16 GB/s channel would deliver data to the processor at a rate no more than 0.76 GB/s. This would be totally unacceptable. This problem is solved by connecting multiple banks to a channel bus.

When two banks are connected to a channel bus, an access can be initiated in the second bank while the first bank is serving another access. Therefore, one can overlap the latency for accessing the DRAM cell arrays. Fig. 6.8(b) shows the timing of a two-bank organization. We assume that the bank 0 started at a time earlier than the window shown in Fig. 6.8. Shortly after the first bank starts accessing its cell array, the second bank also starts access its cell array. When the access in bank 0 is complete, it transfers the burst data (leftmost dark section of the time frame). Once bank 0 completes its data transfer, bank 1 can transfer its burst data (second dark section). This pattern repeats for the next accesses.

From Fig. 6.8(b), we can see that by having two banks, we can potentially double the utilization of the data transfer bandwidth of the channel bus. In general, if the ratio of the cell array access latency and data transfer time is R, we need to have at least R + 1 banks if we hope to fully utilize the data transfer bandwidth of the channel bus. For example, if the ratio is 20, we will need at least 21 banks connected to each channel bus. In general, the number of banks connected to each channel bus needs to be larger than R for two reasons. One is that having more banks reduces

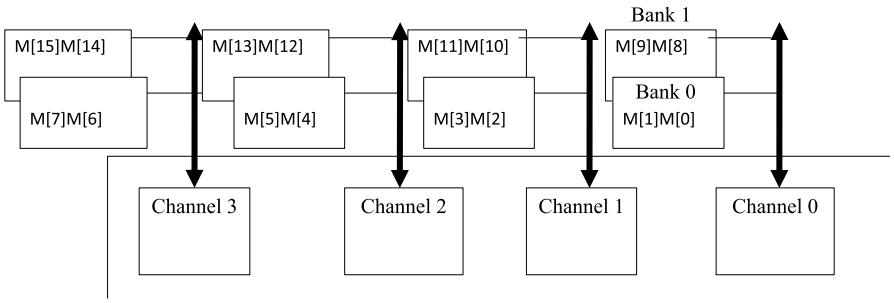


FIGURE 6.9

Distributing array elements into channels and banks.

the probability of multiple simultaneous accesses targeting the same bank, a situation referred to as a *bank conflict*. Since each bank can serve only one access at a time, the cell array access latency can no longer be overlapped for these conflicting accesses. Having a larger number of banks increases the probability that these accesses will be spread out among multiple banks. The second reason is that the size of each cell array is set to achieve reasonable latency and manufacturability. This limits the number of cells that each bank can provide. One may need many banks just to be able to support the memory capacity required.

Fig. 6.9 shows a toy example of distributing the elements of an array M to channels and banks. We assume a small burst size of two elements (eight bytes). The distribution is done by hardware design. The addressing of the channels and banks are such that the first eight bytes of the array (M[0] and M[1]) are stored in Bank 0 of Channel 0, the next eight bytes (M[2] and M[3]) in Bank 0 of Channel 1, the next eight bytes (M[4] and M[5]) in Bank 0 of Channel 2, and the next eight bytes (M[6] and M[7]) in Bank 0 of Channel 3.

At this point, the distribution wraps back to Channel 0 but will use Bank 1 for the next eight bytes (M[8] and M[9]). This way, elements M[10] and M[11] will be in Bank 1 of Channel 1, M[12] and M[13] in Bank 1 of Channel 2, and M[14] and M[15] in Bank 1 of Channel 3. Although not shown in the figure, any additional elements will be wrapped around and start with Bank 0 of Channel 0. For example, M[16] and M[17] will be stored in Bank 0 of Channel 0, M[18] and M[19] will be in Bank 0 or Channel 1, and so on.

The distribution scheme illustrated in Fig. 6.9, often referred to as interleaved data distribution, spreads the elements across the banks and channels in the system. This scheme ensures that even relatively small arrays are spread out nicely. That is, we only assign enough elements to fully utilize the DRAM burst of Bank 0 of Channel 0 before moving on to Bank 0 of Channel 1. In our toy example, as long as we have at least 16 elements, the distribution will involve all the channels and banks for storing the elements.

There is an important connection between the parallel execution of threads and the parallel organization of the DRAM system. In order to achieve the memory access bandwidth specified for device, there must be a sufficient number of threads

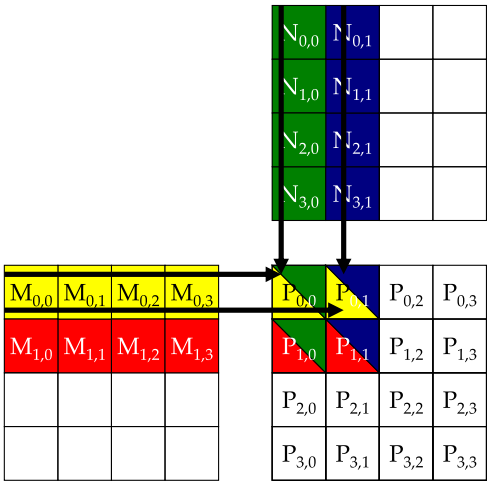


FIGURE 6.10

A small example of matrix multiplication (replicated from Fig. 5.5).

making simultaneous memory accesses. This observation reflects another benefit of maximizing occupancy. Recall that in Chapter 4, we saw that maximizing occupancy ensures that there are enough threads resident on the SMs to hide core pipeline latency, thereby utilizing the instruction throughput efficiently. As we see now, maximizing occupancy also has the additional benefit of ensuring that there are enough memory access requests made to hide DRAM access latency, thereby utilizing the memory bandwidth efficiently. Of course, to achieve the best bandwidth utilization, these memory accesses must be evenly distributed across channels and banks, and each access to a bank must also be a coalesced access.

We now illustrate the interaction between parallel thread execution and the parallel memory organization. We will use the example in Fig. 5.5, replicated as Fig. 6.10. We assume that the matrix multiplication will be performed with 2×2 thread blocks and 2×2 tiles.

During phase 0 of the kernel's execution, all four thread blocks load their first tile. The M elements involved in each tile are shown in Fig. 6.11. Row 2 shows the M elements accessed in Phase 0, with their 2D indices. Row 3 shows the same M elements with their linearized indices. Assume that all thread blocks are executed in parallel. We see that each block will make two coalesced accesses. According to the distribution in Fig. 6.9, these coalesced accesses in Phase 0 by all thread blocks will be made to the banks on Channel 0 and the banks on Channel 2. These accesses will be done in parallel to take advantage of two channels as well as improving the utilization of the data transfer bandwidth of each channel.

We also see that $Block_{0,0}$ and $Block_{0,1}$ will load the same M elements. Most of the modern devices are equipped with caches that will combine these accesses into

Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0], M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2], M[0][3], M[1][2], M[1][3]	M[0][2], M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

FIGURE 6.11

M elements loaded by thread blocks in each phase.

one as long as the execution timings of these blocks are sufficiently close to each other. In fact, the cache memories in GPU devices are mainly designed to combine such accesses and reduce the number of accesses to the DRAM system.

Rows 4 and 5 of Fig. 6.11 show the M elements loaded during phase 1 of the kernel execution. We see that the accesses are now done to the banks in Channel 1 and Channel 3. Once again, these accesses will be done in parallel. It should be clear to the reader that there is a symbiotic relationship between the parallel execution of the threads and the parallel structure of the DRAM system. On one hand, good utilization of the potential access bandwidth of the DRAM system requires that many threads simultaneously access data in the DRAM. On the other hand, the execution throughput of the device relies on good utilization of the parallel structure of the DRAM system, i.e., banks and channels. For example, if the simultaneously executing threads all access data in the same channel, the memory access throughput and the overall device execution speed will be greatly reduced.

The reader is invited to verify that multiplying two larger matrices, such as 8×8 with the same 2×2 thread block configuration, will make use all the four channels in Fig. 6.9. On the other hand, an increased DRAM burst size would require multiplication of even larger matrices to fully utilize the data transfer bandwidth of all the channels.

```
01    __global__ void vecadd_kernel(float* x, float* y, float* z, int N) {
02        int i = blockDim.x*blockIdx.x + threadIdx.x;
03        float4 x4 = ((float4*)x)[i];
04        float4 y4 = ((float4*)y)[i];
05        float4 z4;
06        z4.x = x4.x + y4.x;
07        z4.y = x4.y + y4.y;
08        z4.z = x4.z + y4.z;
09        z4.w = x4.w + y4.w;
10        ((float4*)z)[i] = z4;
11    }
```

FIGURE 6.12

A vector addition kernel that uses vector loads.

6.3 Vector loads and stores

We have seen in Section 6.2 that maximizing the occupancy of threads on the GPU allows these threads to issue enough memory accesses simultaneously to utilize the memory bandwidth efficiently. We can further increase the amount of memory that is accessed simultaneously by having each thread access a larger amount of continuous data in a single instruction. Such load and store instructions that access large amounts of continuous data per thread are referred to as *vector loads* and *vector stores*.

For example, consider the vector addition kernel presented in Fig. 2.10 of Chapter 2. In this kernel, each thread loads two 4 B (i.e., 4-byte) floating point values and stores one 4 B floating point value. Since adjacent threads access adjacent elements in memory with each access, the memory accesses are coalesced. Moreover, since the kernel does not overutilize any resources, it achieves maximum occupancy and issues enough memory loads and stores simultaneously to utilize the memory bandwidth efficiently. However, the memory bandwidth utilization of this kernel can still be further improved. As is, each thread in the kernel executes one load or store instruction for every 4 B of memory that it accesses. Instead, we can use vector loads and stores to have each thread access 16 B of memory with each instruction.

The code that embodies this optimization is shown in Fig. 6.12. In this example, each thread identifies its global index as before (line 02). However, the number of threads launched is only one quarter of the number of elements being added, and the index `i` refers to a chunk of four floating point values, not just a single one. To load a chunk of four floating point values, the pointers to the `x` and `y` arrays are cast to the `float4` datatype before being dereferenced (lines 03-04). The `float4` datatype represents four floating point values as one combined 16 B object. When a pointer to a `float4` datatype is dereferenced, the compiler converts the access to a vector load or store that accesses the entire 16 B object. The loaded objects, `x4` and `y4`, consist of four floating point values each that can be accessed via the fields `.x`, `.y`, `.z`, and `.w`. These fields are added individually and the results are placed in another `float4` object `z4` (lines 05-09). Finally, the pointer to the `z` array is cast to the `float4` datatype while storing `z4` (line 10). This cast ensures that the store is performed as a vector store.

In the case where the total number of elements in the `x` and `y` arrays is not divisible by four, a boundary condition is needed to make sure that the boundary thread

performs scalar loads and stores instead of vector loads and stores to avoid accessing data that is out of bounds. A boundary condition is also needed to make sure that the out of bounds threads in the last thread block are not active. We leave these boundary conditions as an exercise to the reader.

Using vector loads and stores reduces the instruction execution overhead associated with accessing global memory, allowing the memory bandwidth to be utilized more efficiently. In Fig. 6.12, two vector load instructions (lines 03-04) replace eight scalar load instructions, reducing the instruction execution overhead for loading the same amount of data by 75%. Vector loads and stores are also important in situations where kernels cannot execute at full occupancy, allowing such kernels to still be able to issue enough concurrent memory accesses to tolerate memory latency and fully utilize the memory bandwidth. We will see an example of such a situation in Chapter 15.

6.4 Shared memory bank conflicts

Unlike the global memory which is made from DRAM technology, the shared memory in GPUs is made from SRAM technology which has much shorter access latency. The shorter access latency allows the shared memory to avoid using the DRAM bursting technique and thus keep the banks narrow. For example, a typical shared memory structure on GPUs has an access latency of a few clock cycles and consists of 32 banks, with each bank being 32 bits (4 bytes) wide. Hence, when a GPU thread accesses a 32 bit integer or floating point value from shared memory, the thread alone utilizes the output of an entire bank.

When the threads in the same warp access different values from shared memory, it is desirable for each thread to access a different memory bank so that the values can be served in parallel. For this reason, the shared memory is organized such that consecutive 4 B chunks are placed in different banks in an interleaved fashion, similar to the placement of data in DRAM banks and channels described in Section 6.2. For example, the first 4 B are placed in bank 0, the next 4 B in bank 1, the next 4 B in bank 2, etc. When the 32 banks are exhausted, the placement restarts at bank 0. Accordingly, if the 32 threads in the warp access 32 consecutive 4 B integer of floating point values, each thread will be served by a different bank in parallel.

Unfortunately, if the accesses by threads in the same warp are less adjacent to each other, these accesses may hit the same bank at the same time, which results in a *bank conflict*. When a shared memory bank conflict occurs, the bank cannot serve multiple threads simultaneously. Instead, accesses to the same bank need to be served one at a time. The hardware detects when the shared memory accesses by threads in the same warp exhibit a bank conflict and divides these accesses into multiple conflict-free accesses.

For example, consider the corner turning optimization in Fig. 6.4(b) in Section 6.1. Although the threads load from global memory in a coalesced manner, their stores to shared memory have a strided pattern. The code for writing to the shared memory looks as follows:

```
__shared__ float a[TILE_DIM][TILE_DIM]; // TILE_DIM = 32
a[threadIdx.x][threadIdx.y] = ...;
```

Assuming a block dimension of 32×32, threads in the same warp have the same `threadIdx.y` value and consecutive `threadIdx.x` values. Accordingly, these threads write to consecutive elements in the same column. Thread 0 writes to `a[0][0]`, thread 1 writes to `a[1][0]`, thread 2 writes to `a[2][0]`, etc. Since array `a` is stored in row major order, the linear index of `a[i][j]` is $i \times 32 + j$. Hence, the linear indices of `a[0][0]`, `a[1][0]`, `a[2][0]`, etc. are 0, 32, 64, etc. That is, the threads in the same warp access elements that are 32 positions (128 B) apart.

Now assume that the element at index 0 is located in bank 0. In this case, the elements at indices 1 to 31 are located in banks 1 to 31. After reaching the last bank, the order restarts and the subsequent element at index 32 is located in bank 0. Hence, threads 0 and 1 both write to bank 0. Similarly, the element at index 64 written to by thread 2 is also located at bank 0 (64 mod 32). In fact, all threads in the same warp write to the same bank causing a 32-way bank conflict! The hardware serializes these writes, performing 32 successive accesses to the same bank, while the other banks are not used.

To avoid bank conflicts, we need to rearrange the data such that elements accessed by threads in the same warp are placed in different banks. Such a rearrangement can be done adding an extra column to the tile, which effectively adds extra elements between consecutive rows. These extra unused elements that are added to modify the data layout are referred to as *padding*. The following code shows how padding can be added to the corner turning example:

```
__shared__ float a[TILE_DIM][TILE_DIM + 1]; // TILE_DIM = 32
a[threadIdx.x][threadIdx.y] = ...;
```

The key change is that the number of columns in `a` is now `TILE_DIM + 1` instead of `TILE_DIM`.

Let us observe the impact of this change on how the shared memory banks are accessed. Since array `a` now has 33 columns, the linear index of `a[i][j]` is now $i \times 33 + j$. Accordingly, thread 0 accesses `a[0][0]` at linear index 0, which is placed in bank 0. Thread 1 accesses `a[1][0]` at linear index $1 \times 33 + 0 = 33$, which is placed in bank 1 (33 mod 32). Thread 2 accesses `a[2][0]` at linear index $2 \times 33 + 0 = 66$, which is placed in bank 2 (66 mod 32). It is clear that the bank conflicts are eliminated. Each of the 32 threads in the warp accesses a different bank. These accesses are served by the banks in parallel without the need for the hardware to serialize the accesses.

As is the case with many performance optimization techniques, padding comes with cost and limitations. The most obvious cost is that it consumes additional shared memory resources. Another important cost of padding is that the calculation of the linearized index for accessing a padded array can become more expensive than when the original dimensions of the array are of power of two. For example, when the number of columns is 32, which is a 2^5, the multiplication $i \times 32$ in the linearized index calculation can be simplified into a shift operation $i >> 5$. However, when the

number of columns is padded into 33, the multiplication will need to be implemented as $i \times (32 + 1) = i >> 5 + i$. The number of additional operations needed depends on the number of additional columns introduced by padding.

The observed benefit of padding an array depends on the access patterns and the original dimension of the array. For example, if the number of columns in the original array were 16 rather than 32, the bank conflicts among threads in the same warp would have spread across two banks, with a 16-way bank conflict on each bank. Thus, the performance benefit of padding would have been lower. This is intuitive as the smaller stride means that the accesses within the same warp are closer to each other.

6.5 Thread coarsening

So far, in all the kernels that we have seen, work has been parallelized across threads at the finest granularity. That is, each thread was assigned the smallest possible unit of work. For example, in the vector addition kernel in Chapter 2, each thread was assigned one output element. In the RGB to grayscale conversion and the image blur kernels in Chapter 3, each thread was assigned one pixel in the output image. In the matrix multiplication kernels in Chapter 3 and Chapter 5, each thread was assigned one element in the output matrix.

The advantage of parallelizing work across threads at the finest granularity is that it enhances transparent scalability, as discussed in Chapter 4. If the hardware has enough resources to perform all the work in parallel, then the application has exposed enough parallelism to fully utilize the hardware. Otherwise, if the hardware does not have enough resources to perform all the work in parallel, the hardware can simply serialize the work by executing it in multiple waves, delaying the execution of some of the thread blocks until sufficient resources become available due to the completion of the ones that are currently running.

The disadvantage of parallelizing work at the finest granularity comes when there is an overhead associated with parallelizing that work. This overhead can take many forms such as redundant loading of data by different thread blocks, redundant work, synchronization overhead, instruction execution overhead, and others. When the threads are executed in parallel by the hardware, this parallelization overhead is often worth incurring. However, if the hardware ends up serializing the work due to insufficient resources, then the overhead has been incurred unnecessarily. In this case, it is better for the programmer to partially serialize the work and reduce the overhead of parallelization. This can be done by assigning each thread multiple units of work, which is often referred to as *thread coarsening*.

For example, consider the following code where each thread in the kernel is responsible for performing some computation `foo` on different data distinguished by index `i`:

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
foo(i);
```

Here, each thread performs `foo` only once for one value of `i`. Alternatively, thread coarsening could be applied where each thread could be assigned to perform `foo` multiple times for multiple values of `i`. We refer to the number of units of work that the thread is assigned to perform as the *coarsening factor*. The following code shows how thread coarsening could be applied to the previous code example with a coarsening factor of 4:

```
unsigned int iStart = 4*(blockIdx.x*blockDim.x + threadIdx.x);
for(unsigned int c = 0; c < 4; ++c) {
    unsigned int i = iStart + c;
    foo(i);
}
```

Here, the global thread index is multiplied by 4 such that each thread receives the starting value of 4 consecutive integers. For example, thread 0 starts at an `iStart` of 0 and is responsible for integers 0, 1, 2, and 3; thread 1 starts at an `iStart` of 4 and is responsible for integers 4, 5, 6, and 7; etc. Next, each thread executes a loop that iterates over its 4 assigned values. This loop is referred to as a *coarsening loop*. Finally, in each loop iteration, the thread calculates the index `i` corresponding to the unit of work that iteration is responsible for and calls `foo`.

While we introduce thread coarsening using a toy example in this chapter, we will see more realistic examples of how to apply thread coarsening to real kernels throughout this book. We have already seen an example of thread coarsening in Section 6.3 where each thread of the vector addition kernel is assigned to add four pairs of elements instead of one pair to benefit from vector loads and stores. Here, the overhead of parallelizing vector addition across more threads is the instruction execution overhead associated with the load and store instructions. If the threads actually run in parallel, this overhead may be worth paying. However, if there are excess threads and the hardware serializes them, then it is preferable to serialize the additions ourselves within the threads as we have done in Fig. 6.12. Serializing multiple additions in a single thread enables us to combine the memory accesses into vector loads and stores, thereby reducing the instruction execution overhead associated with accessing memory.

Another example where thread coarsening could be beneficial is the tiled matrix multiplication kernel in Chapter 5. In this example, parallelizing the computation across many thread blocks means that the same input tiles are redundantly loaded by different thread blocks. Using fewer thread blocks and having each block process larger output tiles enables each input tile to be loaded fewer times. We will visit this example in depth in Chapter 15 where we apply thread coarsening to matrix multiplication to improve data reuse.

Thread coarsening is a powerful optimization that can result in substantial performance improvement for many applications. It is an optimization that is commonly applied, and that we will see recur in many chapters. However, there are several pitfalls to avoid when applying thread coarsening. The first pitfall is that one must be

careful not to apply the optimization when it is unnecessary. Recall that thread coarsening is beneficial if there is an overhead incurred when parallelizing a computation. Not all computations exhibit such an overhead. For some computations, threads operate completely independently without any coordination or sharing of data. Applying thread coarsening to such computations is not expected to make a substantial performance difference.

The second pitfall to avoid when applying thread coarsening is not to apply too much coarsening such that the hardware resources become underutilized. Recall that exposing as much parallelism as possible to the hardware enables transparent scalability. It provides the hardware with the flexibility of parallelizing or serializing work depending on the amount of execution resources it has. When programmers coarsen threads, they reduce the amount of parallelism exposed to the hardware. If the coarsening factor is too high, not enough parallelism will be exposed to the hardware resulting in some parallel execution resources being unutilized.

For example, if a kernel that has 1056 blocks runs on a GPU that can execute 264 blocks simultaneously, the kernel would have 4 waves that fully utilize the GPU. However, if the threads in this kernel are coarsening by a factor of 8, then the kernel would only have 132 blocks or 0.5 waves. In this case, half the GPU resources would be unutilized, which would likely negatively impact performance. Programmers should also be careful to pick a coarsening factor that does not result in partial waves. For example, if an uncoarsened kernel that has 1584 blocks runs on a GPU that can execute 264 blocks simultaneously, the kernel would have 6 waves that fully utilize the GPU. If the threads in this kernel are coarsening by a factor of 4, then the kernel would have 396 blocks or 1.5 waves. Hence, after the first full wave completes, the second partial wave underutilizes the GPU resources and has a tail effect as discussed in Chapter 4.

In practice, different devices have different amounts of execution resources, so the best coarsening factor is usually device-specific and dataset-specific and needs to be retuned for different devices and datasets. Hence, when thread coarsening is applied, scalability becomes less transparent. That is, the kernel may need to be recompiled from one generation of GPU to another generation using a different coarsening factor to achieve the best possible level of performance.

The third pitfall to avoid when applying thread coarsening is not to increase resource consumption to the extent that it substantially reduces occupancy and hurts performance. Depending on the kernel, thread coarsening may require using more registers per thread or more shared memory per thread block because the thread and thread block are responsible for performing more work. In such cases, if programmers use too many registers or too much shared memory, the occupancy can be reduced. The programmer must be careful that the performance penalty from reducing occupancy is not more detrimental than the performance benefit that thread coarsening may offer.

6.6 Loop unrolling

Recall from Chapter 1 that GPUs follow throughput-oriented design as opposed to latency-oriented design, which means that GPU cores are optimized to execute many threads concurrently at the expense of long latencies experienced by individual threads. In Chapter 4, we discussed an important optimization for tolerating latency by maximizing the occupancy of threads on an SM. Maximizing occupancy ensures that when a warp encounters a stall while waiting for the output of a long-latency instruction, there is a high probability of finding another warp that is ready to execute and keep the SM cores busy. In this section, we discuss another important optimization for tolerating latency, i.e., *loop unrolling*.

Loop unrolling is an optimization that transforms a loop by replicating the body of the loop by some factor and reducing the number of loop iterations by the same factor. For example, consider the following loop:

```
for(unsigned int i = 0; i < 16; ++i) {
    A(i)
    B(i)
}
```

where `A(i)` and `B(i)` are some operations or instructions performed in loop iteration `i`. If we apply loop unrolling with an unrolling factor of four, the loop will be transformed as follows:

```
for(unsigned int i = 0; i < 16; i += 4) {
    A(i)
    B(i)
    A(i + 1)
    B(i + 1)
    A(i + 2)
    B(i + 2)
    A(i + 3)
    B(i + 3)
}
```

There are two advantages of loop unrolling that relate to tolerating latency. The first advantage of loop unrolling is that by reducing the number of loop iterations, the number of branch instructions is reduced. While CPUs have sophisticated branch prediction to reduce the latency of branch instructions, GPUs do not have such sophisticated branch prediction. When a GPU thread encounters a branch instruction, it stalls until the outcome of the branch is resolved. Such stalls are reduced when the number of branch instructions is reduced by loop unrolling.

The second advantage of loop unrolling is that it exposes more independent instructions that can be reordered to avoid stalling. In our example, if `B(i)` depends on `A(i)`, then when a thread executes `A(i)`, it needs to stall and wait for `A(i)` to complete before it can execute `B(i)`. Alternatively, if we have other operations that we can perform that do not need to wait for `A(i)`, we can place those operations in between `A(i)` and `B(i)` to avoid having to stall. In our example, if `A(i + 1)` is independent of `A(i)` and `B(i)`, then the body of the unrolled loop can be further transformed as follows:

```
for(unsigned int i = 0; i < 16; i += 4) {
    A(i)
    A(i + 1)
    A(i + 2)
    A(i + 3)
    B(i)
    B(i + 1)
    B(i + 2)
    B(i + 3)
}
```

In this case, moving `A(i + 1)`, `A(i + 2)`, and `A(i + 3)` in between `A(i)` and `B(i)` ensures that the thread has plenty of instructions to execute while waiting for `A(i)` to finish so it can execute `B(i)`. Reordering instructions in such a way to avoid stalling is part of an important optimization called *instruction scheduling*.

In practice, loop unrolling and instruction scheduling are applied automatically and aggressively by the compiler at the compiler's discretion, and the programmer does not need to worry about them. In our example, since the loop body is small and the loop bound is constant and small, the compiler will most likely fully unroll the loop, eliminating all branches and exposing the largest number of instructions to schedule. If the programmer would like to have control over loop unrolling, this control can be exerted by placing the directive `#pragma unroll N` before the loop to instruct the compiler to unroll the loop by a factor of `N`. For example, the directive can be applied as follows:

```
#pragma unroll 4
for(unsigned int i = 0; i < 16; ++i) {
    A(i)
    B(i)
}
```

In this code, the loop will only be unrolled by a factor of 4. Setting the unrolling factor to 1 instructs the compiler not to perform any unrolling. Throughout this book, we will not annotate loops with the `#pragma unroll` directive except where it is particularly important to emphasize that a loop is expected to be unrolled.

A common category of loops that benefit from loop unrolling are loops that apply thread coarsening (see Section 6.5). Loop unrolling is essential to making thread coarsening efficient. For example, consider the following code:

```
int x;
foo(x);
```

In this code, each thread is responsible for processing some variable `x` which is placed in a register. If we apply thread coarsening to this code, each thread will be responsible for multiple `x`'s. The code after coarsening can be written as follows (assuming a coarsening factor of four):

```
int x[4];
for(unsigned int c = 0; c < 4; ++c) {
    foo(x[c]);
}
```

However, recall from Chapter 5 that local arrays such as `x[4]` are placed in global memory by default. It is necessary to place arrays in memory so that they can be accessed with a variable index, as is done with the access `x[c]`. Unfortunately, placing `x[4]` in global memory can introduce memory access overhead that outweighs the benefits of thread coarsening. However, the thread coarsening loop has a small and constant loop bound (which is typical of thread coarsening loops) so it can be fully unrolled. In this case, the code with the fully unrolled coarsening loop would look as follows:

```
int x[4];
foo(x[0]);
foo(x[1]);
foo(x[2]);
foo(x[3]);
```

Since all accesses to the local array `x` have constant indices after unrolling, as opposed to variable indices, the compiler can promote the local array `x` from memory to registers. Hence, not only does loop unrolling reduce branches and expose more instructions for scheduling, it also exposes opportunities for propagating constants and promoting variables from memory to registers for more efficient access.

6.7 Double buffering

Recall from Chapter 5 that threads in the same thread block use `__syncthreads()` to perform barrier synchronization in order to coordinate their accesses to shared memory. Consider the following example code where threads in the same block pass data to each other during a computation:

```
for(unsigned int i = 0; i < N; ++i) {
    ... = buffer[anotherThreadID];
    __syncthreads();
    buffer[myThreadID] = ...;
    __syncthreads();
}
```

In this example, in every loop iteration, every thread reads some value in memory from a location that was previously written by another thread, then writes a new value to memory at a location corresponding to its own ID. This new value may itself be read by other threads in future iterations. The code example includes two barrier synchronization calls to ensure that the code executes correctly. The second barrier synchronization ensures that all the threads finish writing their values to memory

before other threads read them. In other words, the barrier enforces a *read-after-write* dependence. This dependence is also referred to as a *true dependence* because it is unavoidable. We must wait for a value to be written before we can read it.

On the other hand, the first barrier synchronization ensures that all threads finish reading the values they need before other threads overwrite them on the next iteration. In other words, the barrier enforces a *write-after-read* dependence. This dependence is also referred to as a *false dependence* because we do not really need to wait to read a value before writing another value. The only reason we are forced to wait is because the value being read and the value being written occupy the same memory location. If these two values did not occupy the same memory location, the write-after-read or false dependence would be eliminated, and hence, the first barrier synchronization would not be needed.

Based on this observation, we can eliminate the second barrier synchronization by using different memory buffers on different loop iterations. One way to do so is to allocate a different buffer for each iteration. However, this approach would require a large amount of memory resources which is undesirable. A better approach is to simply allocate two buffers and alternate between them each iteration. This optimization is referred to as *double-buffering* because it requires allocating two buffers in total that keep trading places.

The following code example shows how double-buffering can be applied to the previous example:

```
for(unsigned int i = 0; i < N; ++i) {
    ... = inBuffer[anotherThreadID];
    outBuffer[myThreadID] = ...;
    __syncthreads();
    swap(inBuffer, outBuffer);
}
```

In this example, two different buffers are allocated, one pointed to by `inBuffer` and another pointed to by `outBuffer`. Since the read from memory is performed from `inBuffer` and the write is performed to `outBuffer`, there is no risk that the write will overwrite a value that another thread is reading. Accordingly, the call to `__syncthreads()` that separates these two memory accesses can be eliminated. However, after the write, the most up to date data is now located in `outBuffer`, so this buffer is the one we should read from in the next iteration. Swapping the two buffers at the end of the loop body guarantees that the right buffer is read from on the next iteration. This example is only a toy example, and we will see more concrete examples of how to apply double-buffering to real kernels throughout this book.

6.8 A checklist of optimizations

Throughout this first part of the textbook, we have covered various common optimizations that CUDA programmers apply to improve the performance of their code.

Category	Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Compute utilization	Occupancy tuning	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tune the usage of SM resources such as threads per block, shared memory per block, and registers per thread
	Loop unrolling	Fewer branch instructions and more independent instruction sequences with fewer stalls	May enable promoting local arrays to registers to reduce global memory traffic	Performed automatically by the compiler Use loops with constant bounds where possible to facilitate the compiler's job
	Reducing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	-	Rearrange the assignment of threads to work and/or data Rearranging the layout of the data
Memory utilization	Using coalesceable global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Rearranging the mapping of threads to data Rearranging the layout of the data (e.g., matrix transposition, converting an array-of-structures to a structure-of-arrays) Transfer data between global memory and shared memory in a coalesceable manner and perform irregular accesses in shared memory (e.g., corner turning, packing)
	Shared memory tiling	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Place data that is reused within a block in shared memory so that it is transferred between global memory and the SM only once
	Register tiling	Fewer pipeline stalls waiting for shared memory accesses	Less shared memory traffic	Place data that is reused within a warp or thread in registers so that it is transferred between shared memory and registers only once
	Vector loads and stores	Fewer instructions executed to manage memory accesses	More parallel memory accesses to hide DRAM latency	Assign each thread to load 16 B chunks of contiguous data
	Avoiding shared memory bank conflicts	Fewer pipeline stalls waiting for shared memory accesses	Less shared memory traffic and better utilization of shared memory banks	Rearranging the mapping of threads to data Rearranging the layout of the data (e.g., padding)
	Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Apply partial updates to a private copy of the data then update the public copy when done
Synchron-ization latency	Warp-level primitives	Reduce block-wide barrier synchronizations	Less global and shared memory traffic	Perform operations requiring barrier synchronization at the warp-level, then consolidate warp-level results at the block-level
	Double buffering	Eliminates barriers that enforce false dependences	-	Eliminate false (write-after-read) dependences by using different buffers for the writes and the preceding reads
General	Thread coarsening	Depends on the overhead of parallelization	Depends on the overhead of parallelization	Assign multiple units of parallelism to each thread in order to reduce the overhead of parallelization when it is incurred unnecessarily

FIGURE 6.13

A checklist of optimizations.

We consolidate these optimizations into a single checklist that is shown in Fig. 6.13. This checklist is not an exhaustive one, but it contains many of the universal optimizations that are common across different applications and that programmers should first consider. In the second and third parts of the textbook, we apply the optimizations in this checklist to various parallel patterns and applications to understand how they

manifest in different contexts. In this section, we give a brief overview of each optimization and the strategies for applying it.

The optimizations in Fig. 6.13 are categorized into three major categories based on the aspect of performance that they try to improve. The *compute utilization* category refers to optimizations that aim to reduce idleness in the GPU cores by making sure that cores have useful work to do instead of remaining idle. The *memory utilization* category refers to optimizations that improve the efficiency of how the memory subsystem is used to access data. The *synchronization latency* category refers to optimizations that reduce the need for threads to wait on barrier synchronizations. While some optimizations may fall into multiple categories, we place them in the category corresponding to the primary intent of when they are typically used. The final optimization, thread coarsening, is labeled as general because the category it falls into differs based on the context in which it is applied.

The compute utilization category consists of three main optimizations. The first optimization is tuning the occupancy of threads on SMs. This optimization was introduced in Chapter 4 where the importance of having many more threads than cores was emphasized as a way for having enough work available to hide long-latency operations in the core pipeline. To maximize occupancy, programmers can tune the resource usage of their kernels to ensure that the maximum number of blocks or registers allowed per SM do not limit the number of threads that can be assigned to the SM simultaneously. In Chapter 5, shared memory was also introduced as another resource whose usage should be carefully tuned so as not to limit occupancy. In Section 6.2 of this chapter, the importance of maximizing occupancy was discussed as a means for also hiding memory latency, not just core pipeline latency. Having many threads executing simultaneously ensures that enough memory accesses are generated to fully utilize the memory bandwidth.

Throughout this book, we will apply various optimizations that increase the shared memory or register usage of a kernel, and we will assume that the programmer can adequately tune their usage so as not to adversely affect occupancy. In Chapter 15, we will see an example of where the benefit of using more resources may outweigh the decrease in occupancy, and where other optimizations are used to improve compute utilization that compensate for the reduction in occupancy.

The second optimization for improving compute utilization is loop unrolling. In Section 6.6 of this chapter, we saw that loop unrolling reduces the number of long-latency branch instructions, exposes more independent instructions that can be scheduled to reduce stalls, and may convert local array indices to constants which allows them to be promoted to registers. Throughout this book, we will assume that the compiler does an adequate job at unrolling loops where profitable. In Chapter 11, which covers the prefix sum pattern, and Chapter 15, which covers advanced optimizations for matrix multiplication, we will emphasize how loop unrolling is essential to enable register tiling after thread coarsening. In Chapter 15, we will also comment on how loop unrolling is critical for compensating for the kernel executing with low occupancy.

The third optimization for improving compute utilization is reducing control divergence. Control divergence was introduced in Chapter 4, where the importance of

threads in the same warp taking the same control path was emphasized as a means for ensuring that all cores are productively utilized during SIMD execution. So far, the kernels we have looked at in this part of the textbook have not exhibited control divergence, except the inevitable divergence at boundary conditions. However, we will see many examples in the second and third parts of the textbook where control divergence can be a significant detriment to performance.

There are multiple strategies that can be employed for reducing control divergence. One strategy is to rearrange how the work and/or data is distributed across the threads to ensure that threads in one warp are all used before using threads in other warps. We will see an example of this strategy in Chapter 10, which covers the reduction pattern. The strategy of rearranging how work and/or data is distributed across threads can also be used to ensure that threads in the same warp have similar workload. We will see an example of this in Chapter 18, which covers graph traversal, where we discuss the tradeoffs between vertex-centric and edge-centric parallelization schemes. Another strategy for reducing control divergence is to rearrange how the data is laid out to ensure that threads in the same warp which process adjacent data have similar workload. We will see an example of this strategy in Chapter 17, which covers sparse matrix computation and storage formats, particularly when discussing the JDS format.

The memory utilization category in Fig. 6.13 consists of six main optimizations. The first optimization for improving memory utilization is using coalesced global memory accesses by ensuring that threads in the same warp access adjacent memory locations within the same cache line and/or DRAM burst. This optimization was introduced in Section 6.1 of this chapter, where the hardware's ability to combine accesses to adjacent memory locations into a single memory request was emphasized as a way for reducing global memory traffic and improving utilization of DRAM bursts and cache lines. So far, the kernels we have looked at in this part of the textbook have exhibited coalesced accesses naturally. However, we will see many examples in the second and third parts of the textbook where memory access patterns are more irregular, thereby requiring more effort to achieve coalescing.

There are multiple strategies that can be employed for achieving coalescing in applications with irregular access patterns. One strategy is to load data from global memory to shared memory in a coalesced manner, then perform the irregular accesses on shared memory. We have already seen an example of this strategy in Section 6.1, namely corner turning. We will see another example of this strategy in Chapter 12, which covers the filtering pattern, and Chapter 14, which covers the sorting pattern. In these patterns, threads write out results to an array in a scattered manner. Instead, they can collaborate to perform their scattered accesses in the shared memory, then write the result from the shared memory to the global memory with more coalescing enabled for elements with nearby destinations. This optimization is sometimes referred to as *packing*. We will also see an example of this strategy in Chapter 13, which covers the merge pattern. In this pattern, threads in the same block need to perform a binary search in the same array, so they collaborate to load that array from global memory to shared memory in a coalesced manner, and then each thread performs the binary search in shared memory.

Another strategy for achieving coalescing in applications with irregular access patterns is to rearrange how the threads are mapped to the data elements. We will see an example of this strategy in Chapter 10, which covers the reduction pattern, and Chapter 15, which covers advanced optimizations for matrix multiplication.

Yet another strategy for achieving coalescing in applications with irregular access patterns is to rearrange how the data itself is laid out. We will see an example of this strategy in Chapter 17, which covers sparse matrix computation and storage formats, particularly when discussing the ELL and JDS formats. Other common examples of rearranging the layout of the data are transposing a matrix or converting an array of structures into a structure of arrays.

The second optimization for improving memory utilization is tiling data that is reused within a thread block by placing it in shared memory and accessing it repetitively from there, such that it only needs to be transferred between global memory and the SM once. Shared memory tiling was introduced in Chapter 5 in the context of matrix multiplication, where threads processing the same output tile collaborate to load the corresponding input tiles to the shared memory then access these input tiles repeatedly from the shared memory. We will see shared memory tiling applied again in most of the parallel patterns in the second and third parts of the textbook. We will observe the challenges of applying tiling when the input and output tiles have different dimensions. This challenge arises in Chapter 7, which covers the convolution pattern, and Chapter 8, which covers the stencil pattern. We will also see more sophisticated applications of shared memory tiling to matrix multiplication in Chapter 15 and to sequence alignment using dynamic programming in Chapter 16.

The third optimization for improving memory utilization is tiling data that is reused within a warp or thread by placing it in registers and accessing it repetitively from there, such that it only needs to be transferred between global or shared memory and registers once. Register tiling was applied in Chapter 3 and Chapter 5 where each thread used a local variable or register to accumulate a result before writing it to global memory. One can think of the collective set of registers across all threads as jointly forming a tile of the output data. We will see register tiling again in Chapter 8, where a shared memory input tile is temporarily moved from shared memory to registers and back for efficient access and preservation of shared memory capacity. We will also see register tiling in Chapter 11, which covers the prefix sum pattern, and Chapter 15, which covers advanced optimizations for matrix multiplication, where each thread is responsible for multiple output elements with multiple corresponding input elements that it reuses.

The fourth optimization for improving memory utilization is using vector loads and stores to access larger chunks of memory with less instruction execution overhead. Vector loads and stores were introduced in Section 6.3 of this chapter and applied in the context of vector addition. The optimization can be applied to most computations discussed in this book. Due to the simplicity of the optimization, we will not apply it throughout the book. We will only mention it briefly where it is particularly relevant.

The fifth optimization for improving memory utilization is avoiding shared memory bank conflicts. This optimization was introduced in Section 6.4 of this chapter

in the context of corner turning. The main strategies to avoid bank conflicts are to rearrange how the threads are mapped to the data or to rearrange the data layout itself, often by adding padding elements. We will see an example of this optimization again in Chapter 11, which covers the prefix sum pattern, and in Chapter 15, which covers advanced optimizations for matrix multiplication, where threads make strided accesses to shared memory when they load their portions of the input data.

The sixth optimization for improving memory utilization is privatization. This optimization has not been introduced yet, but it is mentioned here for completeness. Privatization relates to when multiple threads or blocks need to update a common (public) output. To avoid the overhead of updating the same data concurrently, a private copy of the data can be created and partially updated, then a final update can be made from the private copy to the public copy when done. We will see an example of privatization in Chapter 9, which covers the histogram pattern, where multiple threads need to update the same histogram counters. We will also see examples of privatization in Chapter 12, which covers the filtering pattern, and Chapter 18, which covers graph traversal, where multiple threads need to insert entries into the same output list.

The synchronization latency category in Fig. 6.13 consists of two main optimizations. The first optimization for reducing synchronization latency is using warp-level primitives. Warp-level primitives were briefly mentioned in Chapter 4 as a means for threads within the same warp to synchronize and share data together without performing the more expensive block-wide barrier synchronization and accessing shared memory. The main strategy for reducing synchronization overhead with warp-level primitives is to decompose a block-wide computation across warps such that operations requiring barrier synchronization are first performed at the warp-level, then the warp-level results are consolidated at the block-level. We will see examples of using warp-level primitives to optimize computations that perform frequent synchronization in Chapter 10, Chapter 11, and Chapter 12 which cover the reduction, prefix sum, and filter patterns, respectively. We will also visit warp-level primitives in Chapter 15, which covers advanced optimizations for matrix multiplication, where warp-level primitives will be used to assist with register tiling.

The second optimization for reducing synchronization latency is double-buffering. Double buffering was first introduced in Section 6.7 of this chapter. Double-buffering eliminates false (write-after-read) dependences by using different buffers for the writes and the preceding reads, thereby allowing the barrier synchronizations that enforce the false dependences to be removed. We will see double-buffering applied in real contexts in Chapter 11, which covers the prefix sum pattern, and Chapter 15, which covers advanced optimizations for matrix multiplication.

The final optimization in Fig. 6.13 is thread coarsening, which is a general optimization whose performance objective depends on the computation at hand. Thread coarsening was first introduced in Section 6.5 of this chapter, where multiple units of parallel work are assigned to each thread to reduce the overhead of parallelization if the hardware is going to serialize the threads anyways. In the second and third parts of this textbook, we will see thread coarsening applied in different contexts with a

different overhead of parallelization each time. In Chapter 8, which covers the stencil pattern, and Chapter 15, which covers advanced optimizations for matrix multiplication, thread coarsening is applied to increase the size of the output tile, thereby reusing input data loaded from memory to compute more output values. In this case, the overhead of having finer-grain parallelism with smaller output tiles is needing to load the input data redundantly by more thread blocks. In Chapter 9, which covers the histogram pattern, thread coarsening helps reduce the number of private copies that need to be committed to the public copy in the context of the privatization optimization. In Chapter 10, which covers the reduction pattern, and Chapter 11, which covers the prefix sum pattern, thread coarsening is applied to reduce the overhead from synchronization and control divergence. Also in Chapter 11, which covers the prefix sum pattern, thread coarsening helps reduce the redundant work performed by the parallel algorithm compared to the sequential algorithm. In Chapter 12, which covers the filtering pattern, and Chapter 14, which covers the sorting pattern, thread coarsening helps improve memory coalescing.

Again, the checklist in Fig. 6.13 is not intended to be an exhaustive one, but it contains the major types of optimizations that are common across different computation patterns. These optimizations appear in multiple chapters in the second and third parts of the textbook. We will also see other optimizations that appear in specific chapters. For example, in Chapter 7, which covers the convolution pattern, we introduce the use of constant memory.

6.9 Optimization strategy

We have seen in Section 6.8 that different optimizations target different aspects of performance. Hence, when deciding which optimization to apply to a specific kernel, it is important to first understand what resource is limiting the performance of that kernel. The resource that limits the performance of a kernel is often referred to as a performance bottleneck. Optimizations typically use more of one resource to reduce the burden on another resource. If the optimization applied does not target the bottleneck resource, one may not see any benefit from the optimization. Worse yet, if the optimization applied uses more of the bottleneck resource, the optimization may even hurt performance.

For example, shared memory tiling increases the use of shared memory to reduce the pressure on the global memory bandwidth. This optimization is great when the bottleneck resource of a kernel is the global memory bandwidth. However, if the performance of a kernel is limited by occupancy and occupancy is already constrained by the use of too much shared memory, then applying shared memory tiling is likely to make things worse.

Performance optimization is an iterative process whereby one identifies the performance bottleneck of a kernel, applies an optimization to alleviate that bottleneck, identifies the new bottleneck, alleviates it, and keeps repeating. To identify the performance bottleneck of a kernel, GPU computing platforms typically provide various

profiling tools. We refer readers to the CUDA documentation for more information on how to use profiling tools to identify the performance bottlenecks of their kernels [1]. Performance bottlenecks may be hardware-specific, meaning that the same kernel may encounter different bottlenecks on different devices. For this reason, the process of identifying performance bottlenecks and applying performance optimizations requires a good understanding of the GPU architecture and the architectural differences across different GPU devices.

An important skill is knowing when to stop the iterative optimization process because the kernel's performance can no longer be further improved. In Chapter 5, the roofline model was discussed as means for assessing how far a kernel's performance is from the hardware limit. By estimating how many computations a kernel performs and how much data it accesses from memory, the kernel's computational intensity can be derived to infer whether the kernel is compute-bound or memory-bound. After that, the kernel's actual resource utilization reported by the profiler can be compared to the hardware's limit using the roofline model. A kernel that is close to the hardware limit has little room for further optimization. On the other hand, a kernel that has not reached the hardware limit can likely be further optimized.

6.10 Summary

In this chapter, we covered the off-chip memory (DRAM) architecture of a GPU and discussed related performance considerations such as global memory access coalescing and hiding memory latency with memory parallelism. We then presented a few more important optimizations, which are thread granularity coarsening, loop unrolling, and double-buffering. With the insights presented in this chapter and earlier chapters, readers should be able to reason about the performance of any kernel code they come across. We concluded this part of the textbook by presenting a checklist of common performance optimizations that are widely used to optimize many computations. We will continue to study practical applications of these optimizations in the parallel computation patterns and application case studies in the next two parts of the book.

6.11 Exercises

1. Consider the following CUDA kernel:

```
01    __global__ void foo_kernel(float* a, float* b, float* c,
                                 float* d, float* e) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        __shared__ float a_s[256];
04        __shared__ float bc_s[4*256];
05        a_s[threadIdx.x] = a[i];
06        for(unsigned int j = 0; j < 4; ++j) {
```

```
07            bc_s[j*256 + threadIdx.x] =
                        b[j*blockDim.x*gridDim.x + i] + c[i*4 + j];
08        }
09        __syncthreads();
10        d[i + 8] = a_s[threadIdx.x];
11        e[i*8] = bc_s[threadIdx.x*4];
12    }
```

For each of the following memory accesses, specify whether they are coalesced, uncoalesced, or if coalescing is not applicable.

a. The access to array a of line 05
b. The access to array a_s of line 05
c. The access to array b of line 07
d. The access to array c of line 07
e. The access to array bc_s of line 07
f. The access to array a_s of line 10
g. The access to array d of line 10
h. The access to array bc_s of line 11
i. The access to array e of line 11

2. Write a matrix multiplication kernel function that uses corner turning, corresponding to the design illustrated in Fig. 6.4.

3. For tiled matrix multiplication in Chapter 5, out of the possible range of values for `BLOCK_SIZE`, for what values of `BLOCK_SIZE` will the kernel completely avoid uncoalesced accesses to global memory? (You need to consider only square blocks.)

4. Implement a vector addition kernel that uses vector loads and handles the boundary conditions correctly.

5. Consider the following shared memory array and access to the array by a block with one warp (32 threads):

```
__shared__ float a[1024];
a[threadIdx.x*stride] = ...;
```

Assume that `a[0]` is placed in bank 0. For each of the following values of `stride`, specify the banks accessed by the warp and the number of bank conflicts on each bank (if any):

a. `stride` = 32
b. `stride` = 31
c. `stride` = 24
d. `stride` = 16
e. `stride` = 12
f. `stride` = 8
g. `stride` = 7

References

[1] NVIDIA, Nsight compute, https://docs.nvidia.com/nsight-compute/index.html, 2023.

This page intentionally left blank

PART

Parallel patterns

2

This page intentionally left blank

CHAPTER

7 Convolution

And constant memory

CONTENTS

7.1 Background 159
7.2 Parallel convolution – a basic kernel 164
7.3 Memory bandwidth considerations 167
7.4 Constant memory and caching 168
7.5 Tiled convolution with halo cells 172
7.6 Tiled convolution using caches for halo cells 178
7.7 Summary 179
7.8 Exercises 180

In the next several chapters, we discuss a set of important patterns of parallel computation. These patterns are the basis of a wide range of parallel algorithms that appear in many parallel applications. We start with convolution, which is a popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision. In these application areas, convolution is often performed as a filter that transforms signals and pixels into more desirable values. Our image blur kernel is such a filter that smooths out the signal values so that one can see the big-picture trend. For another example, Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images.

Convolution typically performs a significant number of arithmetic operations to generate each output element. For large data sets, such as high-definition images and videos where there are many output elements (pixels), the amount of computation can be huge. On one hand, the output data elements of the convolution computation can be calculated independently of each other, a desirable trait for parallel computing. On the other hand, there is a substantial amount of input data sharing when processing different output data elements with somewhat challenging boundary conditions. These properties make convolution an important use case for sophisticated tiling methods and input data staging methods, which are the focus of this chapter.

7.1 Background

Convolution is an array operation where each output element is a weighted sum of the corresponding input element and a collection of input elements that are centered

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00017-2

Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

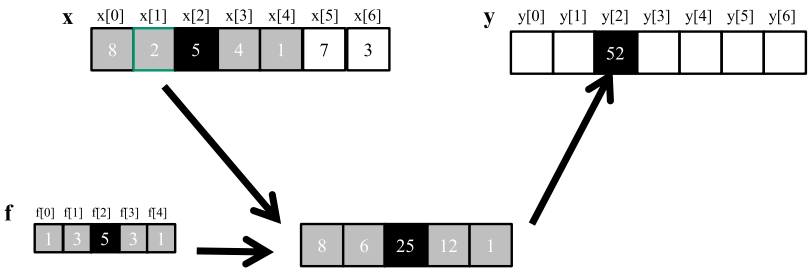


FIGURE 7.1

A 1D convolution example, inside elements.

around it. The weights used in the weighted sum calculation are defined by a filter array, commonly referred to as the convolution kernel. Since there is an unfortunate name conflict between the CUDA kernel functions and convolution kernels, we will simply refer to these filter arrays as convolution filters to avoid confusion.

Convolution can be performed on input data of different dimensionality: 1D (e.g., audio), 2D (e.g., photo), 3D (e.g., video), etc. In audio digital signal processing, the input 1D array elements are sampled signal volume over time. That is, the input data element x_i is the ith sample of the audio signal volume. A convolution on 1D data, referred to as 1D convolution, is mathematically defined as a function that takes an input data array of n elements $[x_0, x_1, ..., x_{n-1}]$ and a filter array of $2r + 1$ elements $[f_0, f_1, ..., f_{2r}]$ and returns an output data array y such that:

$$y_i = \sum_{(j=-r)}^{r} f_{j+r} \times x_{i+j}$$

Since the size of the filter is an odd number $(2r+1)$, the weighted sum calculation is symmetric around the element being calculated. That is, the weighted sum involves r input elements on each side of the position being calculated, which is the reason why r is referred to as the radius of the filter.

Fig. 7.1 shows a 1D convolution example where a 5-element ($r = 2$) convolution filter f is applied to a 7-element input array x. We will follow the C++ language convention where x and y elements are indexed from 0 to 6 and f elements are indexed from 0 to 4. Since the filter radius is two, each output element is calculated as the weighted sum of the corresponding input element, two elements on the left, and two elements on the right.

For example, the value of $y[2]$ is generated as the weighted sum of $x[0]$ (i.e., $x[2 + (-2)]$) through $x[4]$ (i.e., $x[2 + 2]$). In this example, we arbitrarily assume that the values of the x elements are $[8, 2, 5, 4, 1, 7, 3]$. The f elements define the weights, whose values are $[1, 3, 5, 3, 1]$ in this example. Each f element is multiplied to the corresponding x element values before the products are summed together. As shown in Fig. 7.1, the calculation for $y[2]$ is as follows:

$$y[2] = f[0] * x[0] + f[1] * x[1] + f[2] * x[2] + f[3] * x[3] + f[4] * x[4]$$

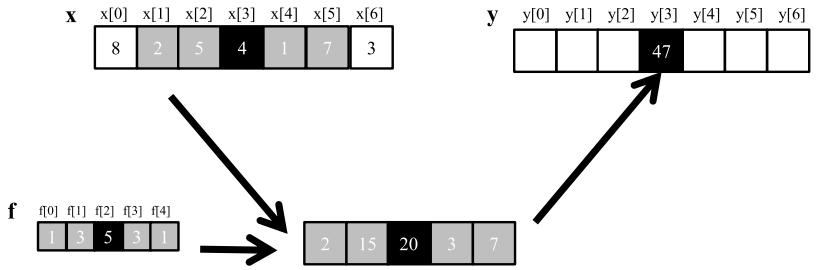


FIGURE 7.2

1D convolution, calculation of y[3].

$$= 1 * 8 + 3 * 2 + 5 * 5 + 3 * 4 + 1 * 1$$
$$= 52$$

In Fig. 7.1, the calculation for $y[i]$ can be viewed as an inner product between the sub-array of x that starts at $x[i-2]$ and the f array[1] Fig. 7.2 shows the calculation for $y[3]$. The calculation is shifted by one x element from that of Fig. 7.1. That is, the value of $y[3]$ is the weighted sum of $x[1]$ (i.e., $x[3+(-2)]$), through $x[5]$ (i.e., $x[3+2]$). We can think of the calculation for $x[3]$ as the following inner product:

$$y[3] = f[0] * x[1] + f[1] * x[2] + f[2] * x[3] + f[3] * x[4] + f[4] * x[5]$$
$$= 1 * 2 + 3 * 5 + 5 * 4 + 3 * 1 + 1 * 7$$
$$= 47$$

Because convolution is defined in terms of neighboring elements, boundary conditions naturally arise when computing output elements that are close to the ends of an array. As shown in Fig. 7.3, when we calculate $y[1]$, there is only one x element to the left of $x[1]$. That is, there are not enough x elements to calculate $y[1]$ according to our definition of convolution. A typical approach to handling such boundary condition is to define a default value to these missing x elements. For most applications, the default value is 0, which is what we use in Fig. 7.3. For example, in audio signal processing, we can assume that the signal volume is 0 before the recording starts and after it ends. In this case, the calculation of $y[1]$ is as follows.

$$y[1] = f[0] * 0 + f[1] * x[0] + f[2] * x[1] + f[3] * x[2] + f[4] * x[3]$$
$$= 1 * 0 + 3 * 8 + 5 * 2 + 3 * 5 + 1 * 4$$
$$= 53$$

[1] Since each output element of convolution can be viewed as an inner product between a sub-array of x and the f array, the reader may wonder if one can formulate convolution in terms of matrix multiplication. There is indeed such a formulation, which will be discussed in Chapter 19 (Convolutional Neural Networks).

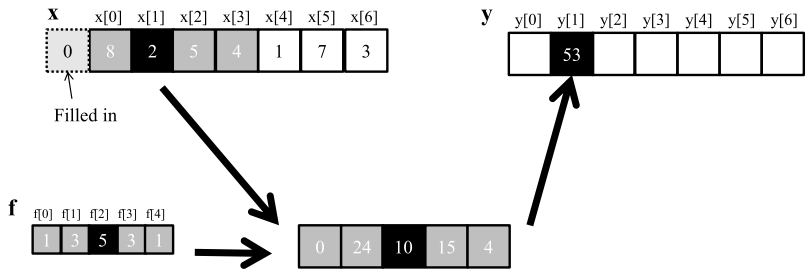


FIGURE 7.3

1D convolution boundary condition.

The x element that does not exist in this calculation is illustrated as a dashed box in Fig. 7.3. It should be clear that the calculation of $y[0]$ would involve two missing x elements and both would be assumed to be 0 for this example. We leave the calculation of $y[0]$ as an exercise. These missing elements are typically referred to as "ghost cells" in the literature. There are also other types of ghost cells due to the use of tiling in parallel computation. These ghost cells can have significant impact on the effectiveness and/or efficiency of tiling. We will come back to this point soon.

Not all applications assume that the ghost cells contain 0. For example, some applications might assume that each ghost cell contains the same value as the closest valid data element on the edge. Others might assume that the ghost cell values can be derived by concatenating a copy of the input array with the input array, creating a circular view of the input array.

For image processing and computer vision, input data is typically represented as two-dimensional arrays, with pixels in an x-y space. Image convolutions are therefore 2D convolutions, as illustrated in Fig. 7.4. In a 2D convolution, the filter f is also a 2D array. Its x- and y-dimensions determine the range of neighbors to be included in the weighted sum calculation. Let us assume that the dimension of the filter is $2r_x + 1$ in the x-dimension and $2r_y + 1$ in the y-dimension. The calculation of each P element can be expressed as:

$$P_{y,x} = \sum_{(j=-r_y)}^{(r_y)} \sum_{(k=-r_x)}^{(r_x)} f_{j+r_y,k+r_x} \times N_{y+j,x+k}$$

In Fig. 7.4, we use a 5×5 filter for simplicity, i.e., $r_y = 2$ and $r_x = 2$. In general, the filter does not have to be but is typically a square array. To generate an output element, we take the sub-array whose center is at the corresponding location in the input array N. We then perform pairwise multiplication between elements of the filter array and those of the image array. For our example, the result is shown as the 5×5 product array below N and P in Fig. 7.4. The value of the output element is the sum of all elements of the product array.

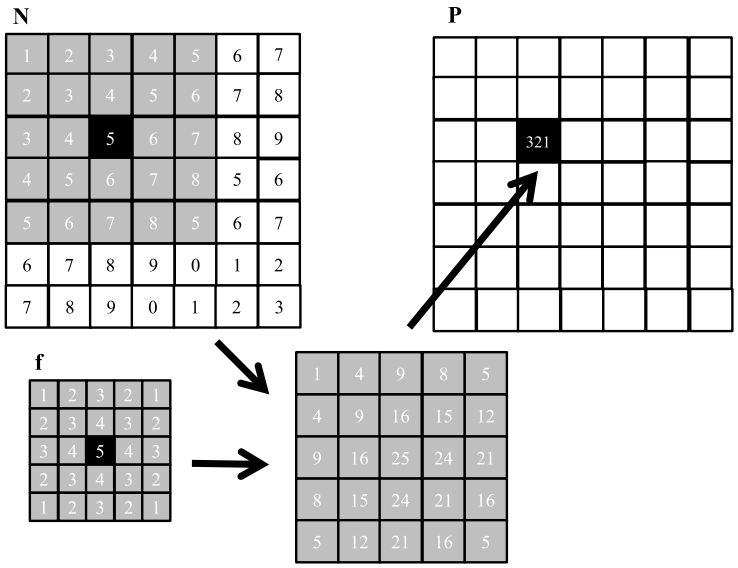


FIGURE 7.4

A 2D convolution example.

The example in Fig. 7.4 shows the calculation of $P_{2,2}$. For brevity, we use $N_{y,x}$ to denote $N[y][x]$ in addressing a C++ array. Since N and P are most likely dynamically allocated arrays, we will be using linearized indices in our actual code examples. The calculation is as follows:

$$
\begin{aligned}
P_{2,2} = {} & N_{0,0} * M_{0,0} + N_{0,1} * M_{0,1} + N_{0,2} * M_{0,2} + N_{0,3} * M_{0,3} + N_{0,4} * M_{0,4} \\
& + N_{1,0} * M_{1,0} + N_{1,1} * M_{1,1} + N_{1,2} * M_{1,2} + N_{1,3} * M_{1,3} + N_{1,4} * M_{1,4} \\
& + N_{2,0} * M_{2,0} + N_{2,1} * M_{1,1} + N_{2,2} * M_{2,2} + N_{2,3} * M_{2,3} + N_{2,4} * M_{2,4} \\
& + N_{3,0} * M_{3,0} + N_{3,1} * M_{3,1} + N_{3,2} * M_{3,2} + N_{3,3} * M_{3,3} + N_{3,4} * M_{3,4} \\
& + N_{4,0} * M_{4,0} + N_{4,1} * M_{4,1} + N_{4,2} * M_{4,2} + N_{4,3} * M_{4,3} + N_{4,4} * M_{4,4} \\
= {} & 1 * 1 + 2 * 2 + 3 * 3 + 4 * 2 + 5 * 1 \\
& + 2 * 2 + 3 * 3 + 4 * 4 + 5 * 3 + 6 * 2 \\
& + 3 * 3 + 4 * 4 + 5 * 5 + 6 * 4 + 7 * 3 \\
& + 4 * 2 + 5 * 3 + 6 * 4 + 7 * 3 + 8 * 2 \\
& + 5 * 1 + 6 * 2 + 7 * 3 + 8 * 2 + 5 * 1 \\
= {} & 1 + 4 + 9 + 8 + 5 \\
& + 4 + 9 + 16 + 15 + 12 \\
& + 9 + 16 + 25 + 24 + 21 \\
& + 8 + 15 + 24 + 21 + 16
\end{aligned}
$$

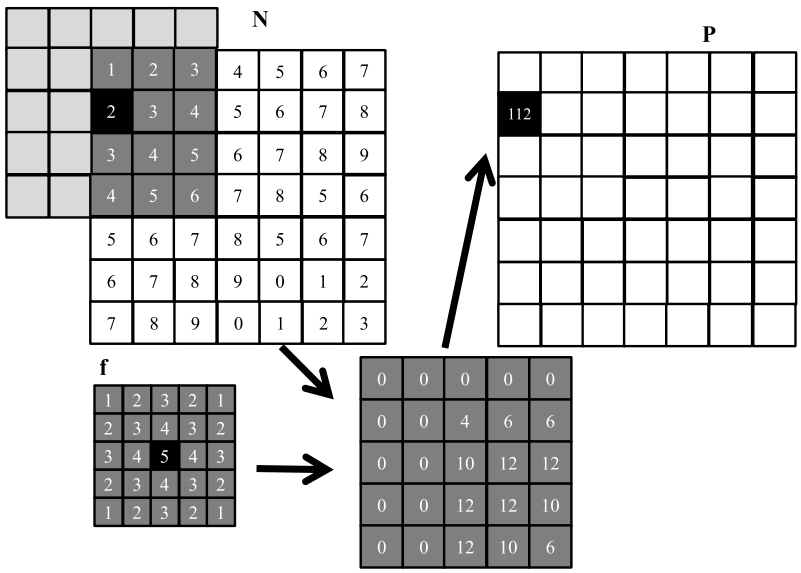


FIGURE 7.5

A 2D convolution boundary condition.

$$+5+12+21+16+5$$
$$=321$$

Like 1D convolution, 2D convolution must also deal with boundary conditions. With boundaries in both the x and y dimensions, there are more complex boundary conditions: the calculation of an output element may involve boundary conditions along a horizontal boundary, a vertical boundary, or both. Fig. 7.5 illustrates the calculation of a P element that involves both boundaries. From Fig. 7.5, the calculation of $P_{1,0}$ involves two missing columns and one missing row in the sub-array of N. Like in 1D convolution, different applications assume different default values for these missing N elements. In our example, we assume that the default value is 0. These boundary conditions also affect the efficiency of tiling. We will come back to this point soon.

7.2 Parallel convolution – a basic kernel

The fact that the calculation of all output elements can be done in parallel in convolution makes convolution an ideal use case for data parallel computing. Based on our experience in matrix multiplication, we can quickly write a simple parallel convolution kernel. We show code examples for 2D convolution and the reader is encouraged

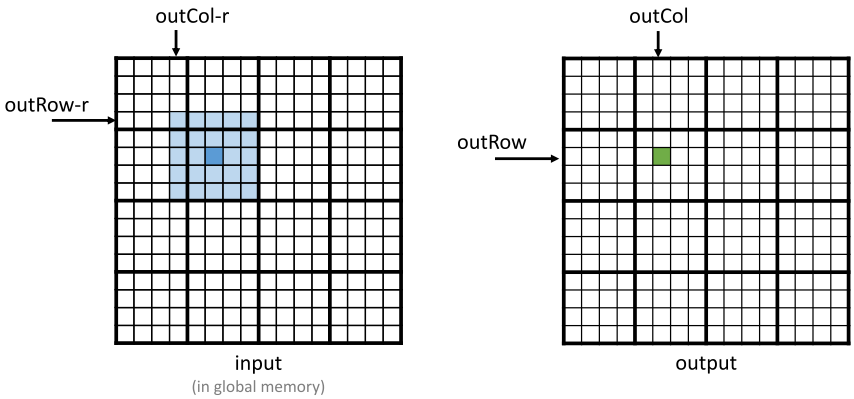


FIGURE 7.6

Parallelization and thread organization for 2D convolution.

to adapt these code examples to 1D and 3D as exercises. Also, for simplicity, we assume the convolution filters are square.

The first step is to define the major input parameters for the kernel. We assume that the 2D convolution kernel receives five arguments: a pointer to input array `N`, a pointer to the filter `F`, a pointer to output array `P`, the radius of the square filter `r`, the width of the input and output arrays `width`, and the height of the input and output arrays `height`. Thus, we have the following kernel signature:

```
__global__ void
convolution_2D_basic_kernel(float *N, float *F, float *P, int r,
                            int width, int height) {
  // kernel body
}
```

The second step is to determine and implement the mapping of threads to output elements. Since the output array is 2D, a simple and good approach is to organize the threads into a 2D grid and have each thread in the grid to calculate one output element. Each block, with up to 1024 threads in a block, can calculate up to 1024 output elements, typically organized as a 32×32 square. Fig. 7.6 shows a toy example where the input and output are 16×16 images. We assume in this toy example that each thread block is organized as a 4×4 array of threads, four threads in the x dimension and four in the y dimension. The grid in this example is organized as a 4×4 array of blocks. The assignment of threads to output elements, i.e., output pixels in this example, is simple: every thread is assigned to calculate an output pixel whose x and y indices are the same as thread's x and y indices.

The reader should recognize that the parallelization arrangement in Fig. 7.6 is the same as that of the `ColorToGrayScaleConversion` example in Chapter 3. Therefore, we can use the statements in lines 2-3 of the kernel in Fig. 7.7 to calculate the output element indices from the block index, block dimension, and thread index for each thread.

```
01 __global__ void convolution_2D_basic_kernel(float *N, float *F, float *P,
      int r, int width, int height) {
02    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
03    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
04    float Pvalue = 0.0f;
05    for (int fRow = 0; fRow < 2*r+1; fRow++) {
06       for (int fCol = 0; fCol < 2*r+1; fCol++) {
07          inRow = outRow - r + fRow;
08          inCol = outCol - r + fCol;
09          if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
10             Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
11          }
12       }
13    }
14    P[outRow][outCol] = Pvalue;
15 }
```

FIGURE 7.7

A 2D convolution kernel with boundary condition handling.

For example, thread$_{1,1}$ of block$_{1,1}$, is mapped to output element $P_{1*4+1,1*4+1} = P_{5,5}$ which is marked as a shaded cell on the right side of Fig. 7.6.

Once we have determined the output element indices for each thread, we can identify the input N elements needed for calculating the output element. As illustrated in Fig. 7.6, the calculation of $P_{5,5}$ by thread$_{1,1}$ of block$_{1,1}$ uses the input elements whose x indices range from $outCol - r = 5 - 2 = 3$ to $outCol + r = 5 + 2 = 7$, and whose y indices range from $outRow - r = 5 - 2 = 3$ to $outRow + r = 5 + 2 = 7$. For all threads, $outCol - r$ and $outRow - r$ define the upper-left corner of the patch of input elements (shaded square on the left side of Fig. 7.6) needed for $P_{outRow,outCol}$. Therefore, we can use a doubly nested loop to iterate through all these index values and perform this calculation (lines 05-13 of Fig. 7.7).

The register variable `Pvalue` accumulates all intermediate results to save DRAM bandwidth. The if-statement in the inner for-loop (line 09) tests if any of the input `N` elements used are ghost cells, on the left, right, top, or bottom side of the `N` array. Since we assume that 0 values will be used for ghost cells, we can simply skip the multiplication and accumulation of the ghost cell element and its corresponding filter element. After the end of the loop, we release the `Pvalue` into the output `P` element (line 14).

We make two observations from the kernel in Fig. 7.7. First, there is control flow divergence on lines 09-10. The threads that calculate the output elements near the four edges of the `P` array need to handle ghost cells. As shown in Section 7.1, each of these threads encounters a different number of ghost cells. Therefore, they all make somewhat different decisions in the if-statement (line 09). The thread that calculates $P_{0,0}$ skips the multiply-accumulate statement most of the time, whereas the one that calculates $P_{0,1}$ skips fewer times, and so on. The cost of control divergence depends on the width and height of the input array and the radius of the filter. For large input arrays and relatively small filters, control divergence only occurs when computing a small portion of the output elements, which keeps the effect of control divergence small. Since convolution is often applied to large images, we expect that the effect of control divergence to be from modest to insignificant. A more important performance consideration is the memory bandwidth.

7.3 Memory bandwidth considerations

Following the same approach introduced in Chapter 5 in the context of matrix multiplication, let us analyze the arithmetic intensity of convolution and reason about its demand for memory bandwidth. For simplicity, let us assume that the input array is square and has dimensions $n \times n$. Let us also assume that the filter is square and has dimensions $m \times m$ ($m = 2r + 1$ where r is the filter radius). All array and filter elements are 32-bit (4 B) single-precision floating point values.

Let us first calculate the number of arithmetic operations performed by convolution. For each of the $n \times n$ output elements, the convolution operation iterates through the $m \times m$ filter and performs two arithmetic operations for each filter element, a multiply and an add. Hence, the total number of arithmetic operations performed by the convolution operation is $2 \cdot n^2 \cdot m^2$ FLOP.

Now, let us calculate the amount of memory data that the convolution operation needs to access. At a minimum, the operation needs to load the convolution filter and the input array, and needs to store the output array. The filter consists of m^2 elements. The input and output arrays consist of n^2 elements each. Since each element is 4 B, the total number of bytes that convolution needs to access is $8 \cdot n^2 + 4 \cdot m^2$. Since the filter is typically much smaller than the array ($m << n$), we can ignore the accesses to the filter and approximate the number of bytes accessed as $8 \cdot n^2$ B.

Taking the ratio of the number of arithmetic operations performed to the number of bytes accessed, we find the arithmetic intensity of convolution to be $\frac{2 \cdot n^2 \cdot m^2}{8 \cdot n^2} = \frac{1}{4} \cdot m^2$ FLOP/B. In other words, a perfect implementation of convolution that accesses each input and output element only once performs $\frac{1}{4} \cdot m^2$ floating point operations for every byte it accesses in memory.

An interesting observation is that the arithmetic intensity of convolution is dependent on the convolution filter size m. Convolutions with small filter sizes have low arithmetic intensity, which makes them memory bound. For example, a convolution with a 3×3 filter has an arithmetic intensity of $\frac{1}{4} \cdot 3^2 = 2.25$ FLOP/B, which makes it memory-bound on an H100 GPU. It is impossible for this computation to reach the peak compute throughput of the GPU because its performance is limited by the memory bandwidth. On the other hand, convolutions with large filter sizes have a high arithmetic intensity, which makes them compute-bound. For example, a convolution with an 11×11 filter has an arithmetic intensity of $\frac{1}{4} \cdot 11^2 = 30.25$ FLOP/B, which makes it compute-bound on an H100 GPU. The performance of this computation can come close to the peak compute throughput of the GPU provided that the right optimizations are applied.

Let us now turn our attention to our basic convolution kernel in Fig. 7.7 of Section 7.2. On line 10 of Fig. 7.7, each thread in each loop iteration loads two elements (8 B) from the global memory and performs two arithmetic operations on them. We ignore the store operation on line 14 because it is negligible in this kernel since it occurs outside the loop nest. Accordingly, the arithmetic intensity of this implementation is (2 FLOP)/(8 B) = 0.25 FLOP/B. Clearly, this arithmetic intensity is very low regardless of the filter size.

In practice, the actual arithmetic intensity of the kernel is higher because of on-chip caching. The same filter and input elements are accessed by multiple threads and are likely to be found in the L1 or L2 caches. Still, to more reliably achieve a higher arithmetic intensity and perform closer to the device's peak throughput, programmers can use other on-chip memory structures such as shared memory or the constant cache. We discuss constant memory and the constant cache in Section 7.4 and shared memory tiling in Section 7.5.

7.4 Constant memory and caching

There are three interesting aspects of the way the filter array F is used in convolution. First, the size of F is typically small; the radius of most convolution filters is 7 or smaller. Even in 3D convolution, the filter typically contains only less than or equal to $7^3 = 343$ elements. Second, the contents of F do not change throughout the execution of the convolution kernel. Third, all threads access the filter elements. Even better, all threads access the filter elements in the same order, starting from $F_{0,0}$ and moving by one element a time through the iterations of the doubly nested for-loop in Fig. 7.7. These three aspects make the filter an excellent candidate for constant memory and caching (Fig. 7.8).

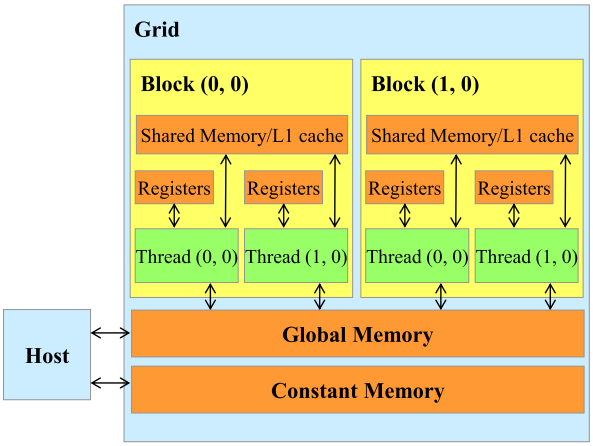


FIGURE 7.8

A review of the CUDA memory model.

As we discussed in Chapter 5 (Fig. 5.4), CUDA allows programmers to declare variables to reside in the constant memory. Like global memory variables, constant memory variables are also visible to all thread blocks. The main difference is that the value of a constant memory variable cannot be modified by threads during kernel execution. Furthermore, the size of the constant memory is quite small, currently at 64 KB.

To use constant memory, the host code needs to allocate and copy constant memory variables in a different way than global memory variables. We assume that the radius of the filter is specified in the compile-time constant `FILTER_RADIUS`. To declare a filter array `F` in constant memory, the host code declares it as a global variable as follows:

```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

This global variable declaration should be outside any function in the source file. The keyword `__constant__` (two underscores on each side) tells the compiler that array `F` should be placed into the device constant memory.

Assume that the host code has already allocated and initialized the filter in an array `F_h` in the host memory with $(2 * FILTER\_RADIUS + 1)^2$ elements. The contents of `F_h` can be transferred from the host memory to `F` in the device constant memory as follows:

```
cudaMemcpyToSymbol(F, F_h, (2*FILTER_RADIUS + 1)*(2*FILTER_RADIUS + 1)
                           *sizeof(float));
```

The `cudaMemcpyToSymbol()` API function is a special memory copy function that informs the CUDA runtime that the data being copied into the constant memory will not be changed during kernel execution. In general, the use of `cudaMemcpyToSymbol()` function is as follows:

```
cudaMemcpyToSymbol(dest, src, size)
```

where `dest` is a pointer to the destination location in the constant memory, `src` is a pointer to the source data in the host memory, and `size` is the number of bytes to be copied.[2]

Kernel functions access constant memory variables as global variables. Thus, their pointers do not need to be passed to the kernel as arguments. We can revise our kernel to use the constant memory as shown in Fig. 7.9. Note that the kernel looks almost identical to that in Fig. 7.7. The only difference is that `F` is no longer accessed through a pointer passed in as a parameter. It is now accessed as a global variable. Keep in mind that all the C++ language scoping rules for global variables apply here. If the host code and kernel code are in different files, the kernel code file must include the relevant external declaration information to ensure that the declaration of `F` is visible to the kernel.

Like global memory variables, constant memory variables are also located in DRAM. However, because the CUDA runtime knows that constant memory variables are not modified during kernel execution, it directs the hardware to aggressively cache the constant memory variables during kernel execution. To understand the benefit of

[2] The function can take two more arguments, offset and kind but these are seldom used and often omitted. The reader is referred to CUDA C++ Programming Guide for details of these arguments.

```
01 __global__ void convolution_2D_const_mem_kernel(float *N, float *P, int r,
int width, int height) {
02    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
03    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
04    float Pvalue = 0.0f;
05    for (int fRow = 0; fRow < 2*r+1; fRow++) {
06    for (int fCol = 0; fCol < 2*r+1; fCol++) {
07         inRow = outRow - r + fRow;
08         inCol = outCol - r + fCol;
09         if (inRow >= 0 && inRow < height && inCol >= 0 && inCol < width) {
10            Pvalue += F[fRow][fCol]*N[inRow*width + inCol];
11         }
12       }
13    }
14    P[outRow*width+outCol] = Pvalue;
15 }
```

FIGURE 7.9

A 2D convolution kernel using constant memory for F.

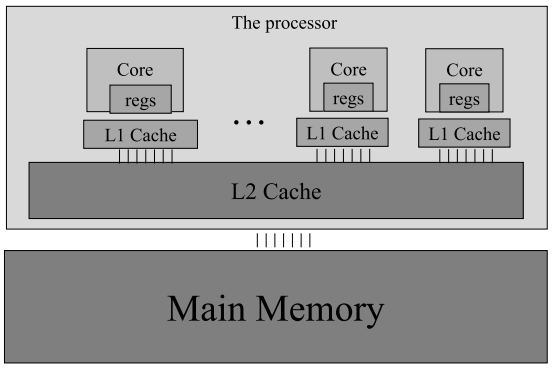


FIGURE 7.10

A simplified view of the cache hierarchy of modern processors.

constant memory usage, we need to first understand more about modern processor memory and cache hierarchies.

As we discussed in Chapter 5 and Chapter 6, the long latency and limited bandwidth of DRAM form a bottleneck in virtually all modern processors. To mitigate the effect of this memory bottleneck, modern processors commonly employ on-chip cache memories, or caches, to reduce the number of variables that need to be accessed from the main memory (DRAM), as shown in Fig. 7.10.

Unlike the CUDA shared memory, or scratchpad memories in general, caches are "transparent" to programs. That is, to use CUDA shared memory to hold the value of a global variable, a program needs to declare variables as `__shared__` and explicitly copy the value of the global memory variable into the shared memory variable. On the other hand, when using caches, the program simply accesses the original global memory variables. The processor hardware automatically retains the most recently or frequently used variables in the cache and remembers their original global memory

address. When one of the retained variables is used later, the hardware detects from their addresses that a copy of the variable is available in cache. The value of the variable is then served from the cache, eliminating the need to access DRAM.

There is a tradeoff between the size of a memory and the speed of a memory. As a result, modern processors often employ multiple levels of caches. The numbering convention for these cache levels reflects the distance to the processor. The lowest level, Level 1 or L1, is the cache that is directly attached to a processor core, as shown in Fig. 7.10. It runs at a speed close to the processor in both latency and bandwidth. However, an L1 cache is small, typically up to 128 KB in capacity. L2 caches are larger, in the range of a few hundred KBs to a few tens of MBs, but can take tens to hundreds of cycles to access. They are typically shared among multiple CPU cores, or SMs in a CUDA device, so the access bandwidth is shared among SMs. In some high-end processors today, there are even L3 caches that can be of hundreds of MBs in size.

Constant memory variables play an interesting role in designing and using memories in massively parallel processors. Since these constant memory variables are not modified during kernel execution, there is no need to support writes by threads when caching them in an SM. Supporting high-thought writes into a general cache requires sophisticated hardware logic and is costly in terms of chip area and power consumption. Without the need for supporting writes, a specialized cache for constant memory variables can be designed in a highly efficient manner in terms of chip area and power consumption. Furthermore, since the constant memory is quite small (64 KB), a small and specialized cache can be highly effective in capturing the heavily used constant memory variables for each kernel. This specialized cache is called a constant cache in modern GPUs.

When all threads in a warp access the same constant memory variable, as is the case of `F` in Fig. 7.9 where the indices for accessing `F` are independent of the thread indices, the constant caches can provide a tremendous amount of bandwidth to satisfy the data needs of these threads. Also, since the size of `F` is typically small, we can assume that all `F` elements are effectively always accessed from the constant cache. Therefore, we can simply assume that no DRAM bandwidth is spent on accesses to the `F` elements.

With this consideration in mind, let us calculate the arithmetic intensity of the kernel in Fig. 7.9. On line 10 of Fig. 7.9, each thread in each loop iteration loads one element (4 B) from the global memory and performs two arithmetic operations on them. The load from `F` in constant memory is not counted since it almost always hits the constant cache. We ignore the store operation on line 14 because it is negligible in this kernel since it occurs outside the loop nest. Accordingly, the arithmetic intensity of this implementation is (2 FLOP)/(4 B) = 0.5 FLOP/B. In other words, it is double that of the basic kernel in Fig. 7.7. In practice, the actual arithmetic intensity of the kernel in Fig. 7.9 is even higher because the elements of `N` are accessed by multiple threads and will sometimes be found in the L1 or L2 cache. Still, we can more reliably optimize the accesses to `N` using shared memory tiling, as we show in Section 7.5.

There is another way to place the filter weights in constant memory without declaring a global array variable. The CUDA compiler actually places kernel parameters in the constant memory as well. Hence, if the filter weights were passed as kernel parameters, they would also be placed in constant memory. A concise way of passing the filter weights as kernel parameters is to place them in a C++ structure and pass the structure by value. This method would allow the kernel to be called many times with different filter weights without the need to explicitly copy these weights every time using `cudaMemcpyToSymbol`. The CUDA runtime will still need to copy them as part of the kernel launch process, but the copy operation is done transparently to the programmer. There is yet another way to leverage the constant nature of the filter weights to avoid accessing them from the global memory. If the filter weights do not change across program executions and were known at compile time, their values could be hard-coded in the code itself. However, this approach would make the kernel not reusable for other filter weight values.

7.5 Tiled convolution with halo cells

We can address the memory bandwidth bottleneck of convolution by applying the shared memory tiling optimization. Recall from Chapter 5 that in shared memory tiling, threads collaborate to load input elements into an on-chip memory for subsequent repeated use of these elements. We start by establishing the definitions of input tiles and output tiles since these definitions are important for understanding the implementation of the optimization. We refer to the collection of output elements processed by each block as an *output tile*. Recall that Fig. 7.6 shows a toy example of a 16×16 2D convolution using $4 \times 4 = 16$ blocks with $4 \times 4 = 16$ threads each. In that example, there are 16 output tiles. Keep in mind that we use 16 threads per block to keep the example small. In practice, there should be at least 32 threads, or one warp, per block, and typically more to achieve a good level of occupancy and data reuse.

We define an *input tile* as the collection of input elements that are needed to calculate the elements in an output tile. Fig. 7.11 shows the input tile (the shaded patch on the left side) that corresponds to the output tile (the shaded patch on the right side) processed by $\text{block}_{1,1}$. Note that the dimensions of the input tile need to be extended by the radius of the filter (2 in this example) in each direction to ensure that it includes all the halo input elements needed for calculating the output elements at the edges of the output tile. This extension can make the input tiles significantly larger than output tiles. In this toy example, each output tile consists of $4^2 = 16$ elements whereas each input tile consists of $(4 + 2 + 2)^2 = 8^2 = 64$ elements. In this case, the input tiles are or $4\times$ larger than the output tiles. However, this large ratio is because we assume a tiny output tile dimension for ease of visualization in the toy example. In practice, the output tile dimensions would be much larger and the ratio between input tile size and output tile size would be smaller. For example, if the output tile size is $16 \times 16 = 256$, with the same 5×5 filter, the input tile size

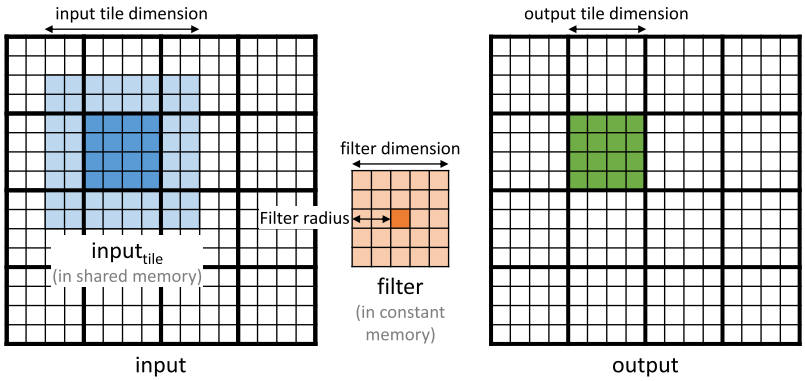


FIGURE 7.11

Input tile and output tile in 2D convolution.

would be $(16 + 2 + 2)^2 = 400$. In this case, the ratio between the input tile size and the output size is about 1.6. Although this ratio is much less than 4, it shows that the input tile size can still be significantly larger than output tiles even for practical output tile dimensions.

In this section, we present one method for applying shared memory tiling to convolution. This method entails having all threads in a block collaboratively loading the entire input tile to the shared memory, then calculating the elements of the output tile by accessing the input elements from the shared memory. This method should sound familiar to the reader: the strategy resembles that of the tiled matrix multiplication implementation discussed in Chapter 5. The main difference is that the tiled matrix multiplication implementation in Chapter 5 assumes that the input tiles are of the same dimension as the output tiles, whereas the convolution input tiles are larger than the output tiles. This difference between input tile size and output tile size complicates the design of tiled convolution kernels.

There are two simple thread organizations for addressing the discrepancy between the input tile size and the output tile size. The first one launches thread blocks whose dimension matches that of the input tiles. This simplifies the loading of the input tiles as each thread needs to load just one input element. However, since the block dimension is larger than that of the output tile, some of the threads need to be disabled during the calculation of output elements. The second approach launches blocks whose dimension matches that of the output tiles. On one hand, this second strategy makes the input tile loading more complex as the threads need to iterate to ensure that all input tile elements are loaded. On the other hand, it simplifies the calculation of the output elements since the dimension of the block is the same as the output tile and there is no need to disable any threads during the calculation of output elements. We present a kernel based on the first thread organization and leave the second organization as an exercise to the reader.

```
01  #define IN_TILE_DIM 32
02  #define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
03  __constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
04  __global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
05                                                        int width, int height) {
06    int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
07    int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
08    //loading input tile
09    __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
10    if(row>=0 && row<height && col>=0 && col<width) {
11      N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
12    } else {
13      N_s[threadIdx.y][threadIdx.x] = 0.0;
14    }
15    __syncthreads();
16    // Calculating output elements
17    int tileCol = threadIdx.x - FILTER_RADIUS;
18    int tileRow = threadIdx.y - FILTER_RADIUS;
19    // turning off the threads at the edges of the block
20    if (col >= 0 && col < width && row >=0 && row < height) {
21      if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
22                     && tileRow<OUT_TILE_DIM){
23        float Pvalue = 0.0f;
24        for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
25          for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
26            Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
27          }
28        }
29        P[row*width+col] = Pvalue;
30      }
31    }
32  }
```

FIGURE 7.12

A tiled 2D convolution kernel using constant memory for `F`.

Fig. 7.12 shows a kernel that is based on the first thread organization. Each thread first calculates the column index (`col`) and row index (`row`) of the input/output element that it is responsible for loading/computing (lines 06-07). The kernel allocates a shared memory array `N_s` whose size is the same as an input tile (line 09) and loads the input tile to the shared memory array (lines 10-15). The conditions on line 10 are used by each thread to check if the input tile element it is attempting to load is a ghost cell. If so, the thread does not perform a memory load. Rather, it places a zero into the shared memory. All threads perform a barrier synchronization (line 15) to ensure that the entire input tile is in place in the shared memory before any thread is allowed to proceed with the calculation of output elements.

Now that all the input tile elements are in the `N_s` array, each thread can calculate its output `P` element value using the `N_s` elements. Keep in mind that the output tile is smaller than the input tile and the blocks are of the same size as the input tiles. For this reason, only a subset of the threads in each block are used to calculate the output tile elements.

There are multiple ways we can select a subset of the block's threads to calculate the output tile elements. Our implementation deactivates `FILTER_RADIUS` exterior layers of threads, as illustrated in Fig. 7.13. Fig. 7.13 shows a small example of convolution using a 3×3 filter (`FILTER_RADIUS` $= 1$), 8×8 input tiles, 8×8 blocks, and

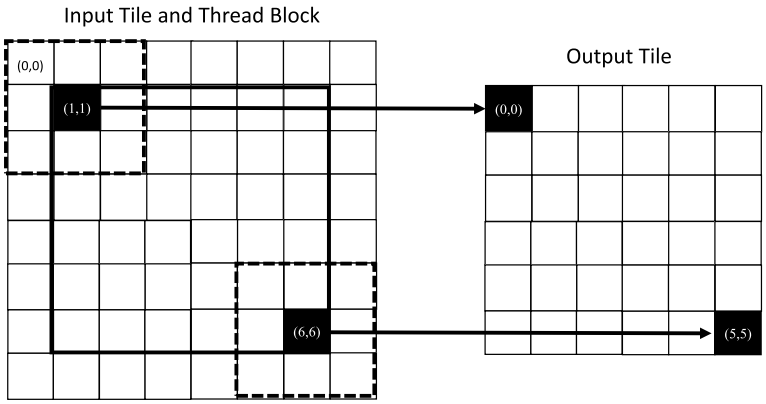


FIGURE 7.13

A small example that illustrates the thread organization for using the input tile elements in the shared memory to calculate the output tile elements.

6 × 6 output tiles. The left side of Fig. 7.13 shows the input tile and the thread block. Since they are of the same size, they are overlayed on top of each other. In our implementation, we deactivate `FILTER_RADIUS` = 1 exterior layer of threads. The heavy-line box at the center of the left side encloses the active threads for calculating the output tile elements. In this example, the `threadIdx.x` and `threadIdx.y` values of the active threads both range from 1 to 6.

Fig. 7.13 also shows the mapping of the active threads to the output tile elements. Active thread (tx, ty) calculates output element (tx – `FILTER_RADIUS`, ty – `FILTER_RADIUS`) using a patch of input tile elements whose upper-left corner is element (tx – `FILTER_RADIUS`, ty – `FILTER_RADIUS`) of the input tile. This is reflected in lines 17-18 of Fig. 7.12, where the column index (`tileCol`) and row index (`tileRow`) of the beginning of the input tile patch are assigned `threadIdx.x - FILTER_RADIUS` and `threadId.y - FILTER_RADIUS`, respectively.

In our small example in Fig. 7.13, `tileCol` and `tileRow` of thread (1,1) receive 0 and 0 respectively. Thus, thread (1, 1) calculates element (0, 0) of the output tile using the 3 × 3 patch of input tile elements highlighted with the dashed box at the upper left corner of the input tile. The loop nest on lines 24-28 of Fig. 7.12 iterates through the input patch and generates the output element. Thread (1, 1) in the block iterates through the patch whose upper left corner is `N_s[0][0]`, whereas Thread (5,5) iterates through the patch whose upper left corner is `N_s[5][5]`.

In lines 06-07, `blockIdx.x*OUT_TILE_DIM` and `blockIdx.y*OUT_TILE_DIM` are the horizontal and vertical `P` array indices of the beginning of the output tile assigned to the block. As we discussed above, `threadIdx.x - FILTER_RADIUS` and `threadIdx.y - FILTER_RADIUS` give the offset into the tile. Thus, the `row` and the `col` variables provide the index of the output element assigned to each active thread. Each thread uses these two indices to write the final value of the output element in line 29.

The tiled 2D convolution kernel in Fig. 7.12 is significantly longer and more complex than the basic kernel in Fig. 7.9. We introduced the additional complexity to reduce the number of DRAM accesses for the `N` elements. The goal is to improve the arithmetic intensity of the kernel so that the achieved performance is not limited or less limited by the memory bandwidth. Recall from Section 7.4 that the arithmetic intensity of the kernel in Fig. 7.9 is 0.5 FLOP/B. Let us now derive the arithmetic intensity for the kernel in Fig. 7.12.

For the blocks that handle tiles at the edges of the data, the threads that handle ghost cells do not perform any memory access for these ghost cells. This reduces the number of memory accesses for these blocks. We can calculate the reduced number of memory accesses by enumerating the number of threads that use each ghost cell. However, it should be clear that for large input arrays, the effect of ghost cells for small filter sizes will be insignificant. So, we will ignore the effect of ghost cells when we calculate the arithmetic intensity of tiled convolution kernels, and only consider the internal thread blocks whose halo cells are not ghost cells.

We now calculate the arithmetic intensity for the tiled kernel in Fig. 7.12. Let t represent the size of the square output tile along one dimension (i.e., `OUT_TILE_DIM`) and m represent the filter dimension (i.e., `2*FILTER_RADIUS+1`). Every thread assigned to an output tile element performs one multiplication and one addition for every element of the filter. Therefore, the threads in an internal block collectively perform $t^2 \cdot m^2 \cdot 2$ arithmetic operations.

As for the global memory accesses, all the global memory loads have been shifted outside the loop nest to the code that loads the input elements into the shared memory. Each thread assigned to an input tile element loads one 4 B input value. Since the input tile dimension (`IN_TILE_DIM`) is $t + m - 1$, the total number of bytes that are loaded by each internal block is $(t + m - 1)^2 \cdot 4$. Since the loads are outside the loop nest, they no longer significantly outnumber the stores so we must also consider the stores in our analysis. The threads of an internal thread block collectively store an entire $t \times t$ output tile of elements that are 4 B each. Hence, the total number of bytes stored is $t^2 \cdot 4$.

Taking the ratio of the arithmetic operations performed by an internal block to the number of bytes the block loads, we find the arithmetic intensity of the tiled convolution kernel in Fig. 7.12 to be:

$$\frac{t^2 \cdot m^2 \cdot 2}{((t + m - 1)^2 + t^2) \cdot 4} = \frac{1}{4} \cdot m^2 \cdot \frac{1}{\frac{1}{2} \cdot (1 + \frac{m-1}{t})^2 + \frac{1}{2}}$$

It is notable that this arithmetic intensity is closer to the ideal arithmetic intensity of $\frac{1}{4} \cdot m^2$ derived in Section 7.3. It is attenuated by a factor that approaches 1.0 as the output tile dimension t increases. Intuitively, a larger output tile enables us to reuse input tile elements that are loaded to shared memory to compute a larger number of output tile elements. Hence, a larger t provides a higher arithmetic intensity. Of course, t in our kernel implementation is limited by multiple hardware resource constraints such as the thread block size limit and the shared memory capacity limit. We cover optimization techniques to overcome these challenges in Chapter 8.

Filter dimension (m)	Output tile dimension (t) = 8	16	28	32	64	Ideal	
3	1.76	1.99	2.10	2.11	2.18	2.25	Memory bound
5	3.85	4.88	5.42	5.52	5.87	6.25	
7	6.03	8.48	9.90	10.17	11.16	12.25	
9	8.10	12.46	15.27	15.80	17.88	20.25	
11	9.98	16.62	21.29	22.22	25.89	30.25	Compute bound
13	11.66	20.80	27.79	29.23	35.06	42.25	

FIGURE 7.14

Arithmetic intensity as a function of tile size and filter size for 2D tiled convolution.

Fig. 7.14 shows how the value of the expression we derived for the arithmetic intensity of the tiled convolution kernel in Fig. 7.12 varies for different filter sizes and tile dimensions. It is clear that the computation is memory bound for small filter and tile dimensions, and compute bound for large filter and tile dimensions. For our example with a 5×5 filter and 28×28 output tiles (32×32 input tiles), the arithmetic intensity is 5.42 FLOP/B. In practice, it could be larger because some values used by different thread blocks can be found in the caches. This value is reasonably close to the ideal arithmetic intensity with a 5×5 filter, which is 6.25 FLOP/B.

The tiled convolution kernel implemented in Fig. 7.12 has a number of inefficiencies. Much of the inefficiency has to do with the fact that the input tile dimensions and block dimensions are larger than the output tile dimensions because of the loading of halo cells. The first inefficiency is that the implementation launches threads that only load data from global memory to shared memory, but do not participate in the computation stage, thereby wasting compute resources. While this approach may be beneficial for memory-bound kernels where having more parallelism when accessing memory is useful, it may become a drawback in compute-bound kernels where the resources wasted by the excluded threads are more valuable. The second inefficiency is that the implementation has output tile dimensions that are not powers of two in size, which may cause misaligned memory accesses when storing the final result to the global memory. Since the maximum size of the thread block (and input tile) is 32×32, the output tile for a 5×5 filter is 28×28. Hence, the output tiles stored by each thread block are not perfectly aligned in memory which can cause some inefficiency when storing them. The third inefficiency is that supporting larger filter dimensions requires shrinking the output tile because the size of the input tile cannot be expanded. As we have seen, shrinking the output tile results in reducing the arithmetic intensity of the kernel, not to mention exacerbating the two aforementioned inefficiencies.

To overcome these issues, one could match the block size to the output tile dimension and have threads iterate to load the larger input tile to shared memory. This alternative corresponds to the second thread organization mentioned earlier in this section and left as an exercise for the reader. Another possibility is to avoid loading the halo cells to shared memory entirely and rely on the caches for accessing them quickly. This alternative is introduced in Section 7.6

```
01  #define TILE_DIM 32
02  __constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
03  __global__ void convolution_cached_tiled_2D_const_mem_kernel(float *N,
                                          float *P, int width, int height) {
04    int col = blockIdx.x*TILE_DIM + threadIdx.x;
05    int row = blockIdx.y*TILE_DIM + threadIdx.y;
      //loading input tile
06    __shared__ N_s[TILE_DIM][TILE_DIM];
07    if(row<height && col<width) {
08      N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
09    } else {
10      N_s[threadIdx.y][threadIdx.x] = 0.0;
11    }
12    __syncthreads();
      // Calculating output elements
      // turning off the threads at the edges of the block
13    if (col < width && row < height) {
14      float Pvalue = 0.0f;
15      for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
16        for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
17          if (threadIdx.x-FILTER_RADIUS+fCol >= 0 &&
18              threadIdx.x-FILTER_RADIUS+fCol < TILE_DIM &&
19              threadIdx.y-FILTER_RADIUS+fRow >= 0 &&
20              threadIdx.y-FILTER_RADIUS+fRow < TILE_DIM){
21            Pvalue += F[fRow][fCol]*N_s[threadIdx.y+fRow][threadIdx.x+fCol];
22          }
23          else {
24            if (row-FILTER_RADIUS+fRow >= 0 &&
25                row-FILTER_RADIUS+fRow < height &&
26                col-FILTER_RADIUS+fCol >=0 &&
27                col-FILTER_RADIUS+fCol < width) {
24              Pvalue += F[fRow][fCol]*
25                    N[(row-FILTER_RADIUS+fRow)*width+col-FILTER_RADIUS+fCol];
26            }
27          }
28        }
29        P[row*width+col] = Pvalue;
30      }
31    }
32  }
```

FIGURE 7.15

A tiled 2D convolution kernel using caching for halos and constant memory for F.

7.6 Tiled convolution using caches for halo cells

Recall that the halo cells of an input tile of a block are also the internal elements of neighboring tiles. For example, in Fig. 7.11, the lightly shaded halo cells of an input tile are also internal elements of the input tiles of neighboring blocks. There is a significant probability that by the time a block needs to its halo cells, they are already in the L2 cache due to the accesses by its neighboring blocks. As a result, the memory accesses to these halo cells may be naturally served from L2 cache without causing additional DRAM traffic. That is, we can leave the accesses to these halo cells in the original `N` elements rather than loading them into the `N_s`. Doing so enables us to use the same dimension for the shared memory input tile and the output tile, and only load the internal elements of each input tile into the shared memory.

Fig. 7.15 shows a 2D convolution kernel using caching for halo cells. In this tiled kernel, the shared memory `N_s` array only needs to hold the internal elements of the tile. Thus, the input tiles and output tiles are of the same dimension, which is defined as constant `TILE_DIM` (line 1). With this simplification, `N_s` is declared to have `TILE_DIM` elements in both the x and y dimensions (line 6).

Because the input tiles and output tiles are of the same size, the thread blocks can be launched with the same size of the input/output tiles. Loading of the `N_s` elements becomes simpler since each thread can simply load the input element that has the same row and column index as its assigned output element (lines 4-5, 7-11). The condition for loading an input element is also simplified in line 7 because the kernel no longer loads halo cells into shared memory so there is no danger of loading ghost cells. Thus, the condition only needs to check for the usual boundary condition that a tile may extend beyond the valid range of the input data.

The body of the loop that calculates the elements of `P`, however, becomes more complex. It needs to add conditions to check for use of both halo cells and ghost cells. The handling of halo cells are done with the conditions on lines 17-20, which test if the input element falls within the interior of the input tile. If so, the element is accessed from the shared memory. If not, the conditions on lines 24-27 check if the halo cells are ghost cells. If so, no action is taken for the element since we assume that the ghost values are 0. Otherwise, the element is accessed from the global memory. The reader should verify that the conditions for handling the ghost cells are similar to those used in Fig. 7.7.

A subtle advantage of the kernel in Fig. 7.15 compared to the kernel in Fig. 7.12 is that its block size, input tile size, and output tile size can be the same and can be a power of two. Because the input tile size and output tile size are different for the kernel in Fig. 7.12, there is likely more memory divergence and control divergence during the execution of that kernel.

7.7 Summary

In this chapter, we have studied convolution as an important parallel computation pattern. While convolution is used in many applications such as computer vision and video processing, it also represents a general pattern that forms the basis of many parallel algorithms. For example one can view the stencil algorithms in partial differential equation (PDE) solvers as a special case of convolution, and this will be the subject of Chapter 8. We will also apply much of what we learned in this chapter on convolutional neural networks in Chapter 20.

We presented a basic parallel convolution implementation whose performance is limited by DRAM bandwidth for accessing both the input and filter elements. We then introduced constant memory and a simple modification to the kernel and host code to take advantage of constant caching and eliminate practically all DRAM accesses for the filter elements. We further introduced a tiled parallel convolution implementation that reduces DRAM bandwidth consumption by leveraging the shared memory while

introducing more control flow divergence and programming complexity. Finally, we presented a tiled parallel convolution implementation that takes advantage of the L1 and L2 caches for handling halo cells.

We presented an analysis of the benefit of tiling in terms of elevated arithmetic intensity. This analysis is an important skill that was introduced in Chapter 5 and will be useful in understanding the benefit of tiling for other patterns as well. Through the analysis, we learned about the limitation of small filter and tile sizes.

Although we have shown examples for only 1D and 2D convolution, the techniques are directly applicable to 3D convolutions as well. In general, the index calculation for the input and output arrays is more complex due to higher dimensionality. Also, one will have more loop nesting for each thread since multiple dimensions need to be traversed when loading tiles and/or calculating output values. We encourage the reader to complete these higher dimension kernels as homework exercises.

7.8 Exercises

1. Calculate the `P[0]` value in Fig. 7.3.
2. Consider performing a 1D convolution on array $N = \{4, 1, 3, 2, 3\}$ with filter $F = \{2, 1, 4\}$. What is the resulting output array?
3. What do you think the following 1D convolution filters are doing?
   a. $[0, 1, 0]$
   b. $[0, 0, 1]$
   c. $[1, 0, 0]$
   d. $[-\frac{1}{2}, 0, \frac{1}{2}]$
   e. $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$
4. Consider performing a 1D convolution on an array of size N with a filter of size M:
   a. How many ghost cells are there in total?
   b. How many multiplications are performed if ghost cells are treated as multiplications (by 0)?
   c. How many multiplications are performed if ghost cells are not treated as multiplications?
5. Consider performing a 2D convolution on a square matrix of size $N \times N$ with a square filter of size $M \times M$:
   a. How many ghost cells are there in total?
   b. How many multiplications are performed if ghost cells are treated as multiplications (by 0)?
   c. How many multiplications are performed if ghost cells are not treated as multiplications?
6. Consider performing a 2D convolution on a rectangular matrix of size $N_1 \times N_2$ with a rectangular filter of size $M_1 \times M_2$:
   a. How many ghost cells are there in total?

b. How many multiplications are performed if ghost cells are treated as multiplications (by 0)?

c. How many multiplications are performed if ghost cells are not treated as multiplications?

7. Consider performing a 2D tiled convolution with the kernel shown in Fig. 7.12 on an array of size $N \times N$ with a filter of size $M \times M$ using an output tile of size $T \times T$.

a. How many thread blocks are needed?

b. How many threads are needed per block?

c. How much shared memory is needed per block?

d. Repeat the same questions if you were using the kernel in Fig. 7.15.

8. Revise the 2D kernel in Fig. 7.7 to perform 3D convolution.

9. Revise the 2D kernel in Fig. 7.9 to perform 3D convolution.

10. Revise the tiled 2D kernel in Fig. 7.12 to perform 3D convolution.

11. Revise the tiled 2D kernel in Fig. 7.12 to use as many threads per block as the output tile size, and have each thread load multiple elements when loading the input tile to shared memory.

This page intentionally left blank

CHAPTER

Stencil computation

And thread coarsening

8

CONTENTS

8.1 Background 184
8.2 Parallel stencil – a basic kernel 187
8.3 Memory bandwidth considerations 189
8.4 Shared-memory tiling for stencil sweep 189
8.5 Thread coarsening 193
8.6 Register tiling 196
8.7 Summary 198
8.8 Exercises 198

Stencils are foundational to numerical methods for solving partial differential equations in application domains like fluid dynamics, heat conductance, combustion, weather forecast, climate simulation, and electromagnetics. The data processed by stencil-based algorithms are discretized quantities of physical significance, such as mass, velocity, force, acceleration, temperature, electrical field, and energy whose relations with each other are governed by differential equations. A common use of stencils is to approximate the derivative values of a function based on the function values within a range of input variable values. Stencils bear strong resemblance to convolution in that they both calculate the new value of an element of a multidimensional array based on the current value of the element at the same position and those in the neighborhood in another multidimensional array. Therefore, stencils also need to deal with halo cells and ghost cells. Unlike convolution, a stencil computation is used to iteratively solve the values of continuous and differentiable functions within the domain of interest. The data elements and the weight coefficients used for the elements in the stencil neighborhood are governed by the differential equations being solved. Some stencil patterns are amenable to optimizations that are not applicable to convolution. In the solvers where initial conditions are iteratively propagated through the domain, the calculations of output values may have dependences and need to be performed according to some ordering constraints. Furthermore, because of the numerical accuracy requirements in solving differential equations, the data processed by stencils tend to be high-precision floating data that consume more on-chip memory for tiling techniques. Due to these differences, stencils tend to motivate different optimizations than convolution.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00018-4
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

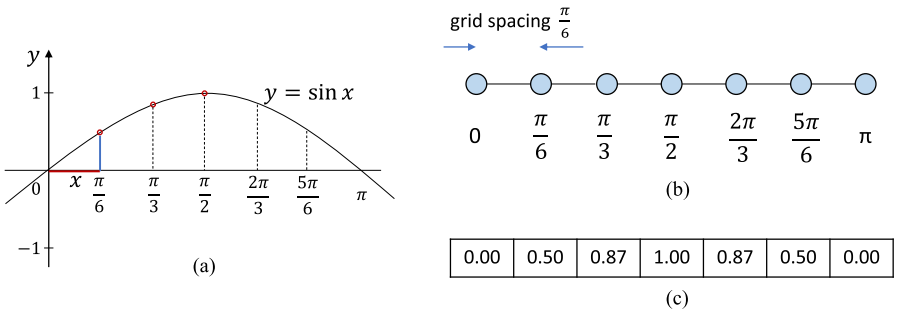


FIGURE 8.1

(a) Sine as a continuous and differentiable function for $0 \leq x \leq \pi$, (b) Design of a regular grid with constant spacing ($\frac{\pi}{6}$) between grid point for discretization, (c) Resulting discrete representation of the sine function for $0 \leq x \leq \pi$.

8.1 Background

The first step in using computers to numerically evaluate and solve functions, models, variables, and equations is to convert them into a discrete representation. For example, Fig. 8.1(a) shows the sine function $y = sin(x)$ for $0 \leq x \leq \pi$. Fig. 8.1(b) shows the design of a 1D regular (structured) grid whose seven grid points correspond to x values that are of constant spacing ($\frac{\pi}{6}$) apart. In general, structured grids cover an n-dimensional Euclidean space with identical parallelotopes (e.g., segments in 1D, rectangles in 2D, bricks in 3D). As we will see, with structured grids, derivatives of variables can be conveniently expressed as finite differences. Therefore, structured grids are mainly used in finite-difference methods. Unstructured grids are more complex and are used in finite-element and finite-volume methods. For simplicity, we only use regular grids and thus finite-difference methods in this chapter.

Fig. 8.1(c) shows the resulting discrete representation where the sine function is represented with its values at the seven grid points. In this case, the representation is stored in a 1D array F. Note that the x values are implicitly assumed to be $i \cdot \frac{\pi}{6}$ where i is the index of the array element. For example, the x value that corresponds to element $F[2]$ is 0.87, which is the sine value of $2 \cdot \frac{\pi}{6}$.

In a discrete representation, one needs to use an interpolation technique such as linear or splines to derive the approximate value of the function for x values that do not correspond to any of the grid points. The fidelity of the representation, or how accurate the function values from these approximate interpolation techniques are, depends on the spacing between grid points: the smaller the spacing, the more accurate the approximations. By decreasing the spacing, one can improve the accuracy of the representation at the cost of more storage and, as we will see, more compute when we solve a partial differential equation.

The fidelity of a discrete representation also depends on the precision of the numbers used. Since we are approximating continuous functions, floating-point numbers are typically used for the grid point values. Mainstream CPUs and GPUs support

double-precision (64-bit), single-precision (32-bit), half-precision (16-bit), and other representations (see Appendix A). Among these, double-precision numbers offer the best precision and the most fidelity in discrete representations. However, modern CPUs and GPUs typically have much higher arithmetic computation throughput for single-precision and half-precision arithmetic. Also, since double precision numbers consist of more bits, reading and writing double precision numbers consumes more memory bandwidth. Storing these double precision numbers also requires more memory capacity. This can pose a significant challenge for tiling techniques that require a significant number of grid point values to be stored in on-chip memory and registers in order to achieve high arithmetic intensity, as we learned in Chapter 7.

Let us discuss stencils with a little more formalism. In mathematics, a stencil is a geometric pattern of weights applied at each point of a structured grid. The pattern specifies how the values of the grid point of interest can be derived from the values at neighboring points using a numerical approximation routine. For example, a stencil may specify how the derivative value of a function at a point of interest is approximated using finite differences between the values of the function at the point and its neighbors. Since partial differential equations express the relations between functions, variables, and their derivatives, stencils offer a convenient basis for specifying how finite difference methods numerically compute the solutions to partial differential equations.

For example, assume that we have a function $f(x)$ discretized into a 1D grid array F and we would like to calculate the discretized derivative of $f(x)$, $f'(x)$. We can use the classic finite difference approximation for the first derivative:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

That is, the derivative of a function at point x can be approximated with the difference of the function values at the two neighboring points divided by the difference of the x values of these neighboring points. The value h is the spacing between neighboring points in the grid. The error is expressed by the term $O(h^2)$, meaning that the error is proportional to the square of h. Obviously, the smaller the h value, the better the approximation. In our example in Fig. 8.1, the h value is $\frac{\pi}{6}$ or 0.52. The value is not small enough to make the approximation error negligible but should result in a reasonably close approximation.

Since the grid spacing is h, the current estimated $f(x-h)$, $f(x)$, $f(x+h)$ values are in $F[i-1]$, $F[i]$, and $F[i+1]$ where $x = i \cdot h$. So, we can calculate the derivative values of $f(x)$ at each grid point into an output array FD:

$$FD[i] = \frac{F[i+1] - F[i-1]}{2 \cdot h}$$

for all the grid points. This expression can be rewritten as:

$$FD[i] = \frac{-1}{2 \cdot h} \cdot F[i-1] + \frac{1}{2 \cdot h} \cdot F[i+1]$$

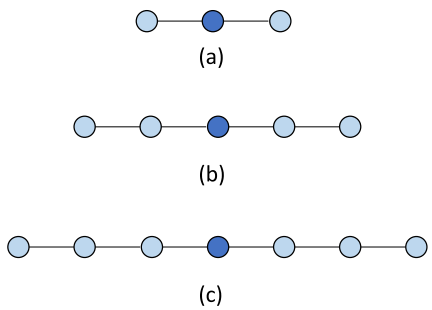


FIGURE 8.2

1D stencil examples (a) 3-point (order 1) stencil, (b) 5-point (order 2) stencil, (c) 7-point (order 3) stencil.

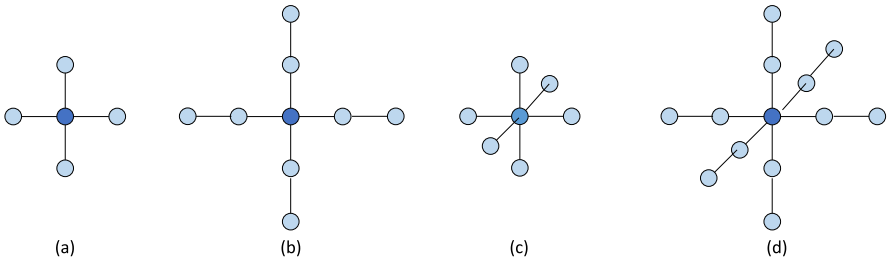


FIGURE 8.3

(a) 2D 5-point stencil (order 1), (b) 2D 9-point stencil (order 2), (c) 3D 7-point stencil (order 1), (d) 3D 13-point stencil (order 2).

That is, the calculation of the estimated function derivative value at grid point i involves the current estimated function values at grid points $[i-1, i, i+1]$ with coefficients $[\frac{-1}{2 \cdot h}, 0, \frac{1}{2 \cdot h}]$, which defines a 1D 3-point stencil, as shown in Fig. 8.2(a). If we are approximating higher derivative values at grid points, we would need to use higher-order finite differences. For example, if the differential equation includes up to the second derivative of $f(x)$, we will use a stencil involving grid points $[i-2, i-1, i, i+1, i+2]$, which is a 1D 5-point stencil, as shown in Fig. 8.2(b). In general, if the equation involves up to the nth derivative, the stencil will involve n grid points on each side of the center grid point. Fig. 8.2(c) shows a 1D 7-point stencil. The number of grid points on each side of the center point is called the order of the stencil, as it reflects the order of the derivative being approximated. According to this definition, the stencils in Fig. 8.3(a), (b), and (c) are of order 1, 2, and 3, respectively.

It should be obvious that solving a partial differential equation of two variables would require the function values to be discretized into a 2D grid, and we need to use a 2D stencil to calculate the approximate partial derivatives. If the partial differential equation involves exclusively the partial derivatives by only one of the variables, e.g., $\frac{\partial f(x,y)}{\partial x}$, $\frac{\partial f(x,y)}{\partial y}$, but not $\frac{\partial f(x,y)}{\partial x \partial y}$, we can use 2D stencils whose selected grid points

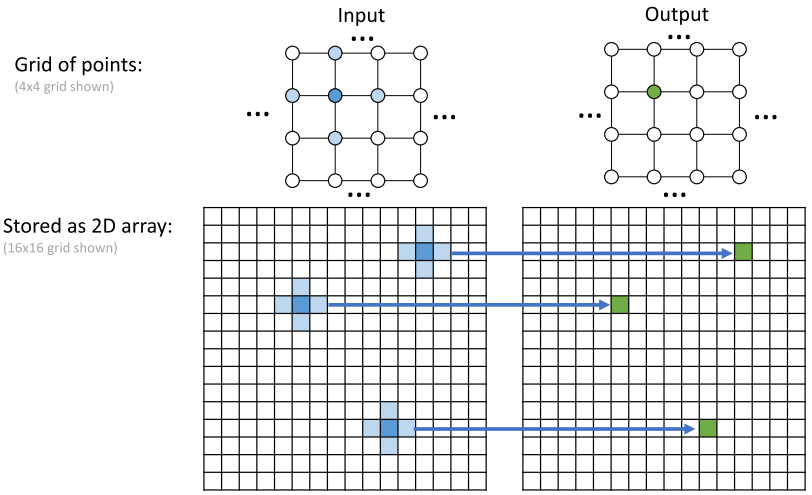


FIGURE 8.4

A 2D grid example and a 5-point (order 1) stencil used to calculate the approximate derivative values at different grid points.

are all along the x axis and y axis. For example, for a partial differential equation involving only first derivatives by x and first derivatives by y, we can use a 2D stencil that involves two grid points on each side of the center point along the x axis and the y-axis, which results in the 2D 5-point stencil in Fig. 8.3(a). If the equation involves up to second order derivatives only by x or by y, we will use a 2D 9-point stencil, as shown in Fig. 8.3(b).

Fig. 8.4 uses a 5-point stencil example to illustrate the concepts of numerical grids, and application of stencils on the grid points. Functions are discretized into their grid point values which are stored in multi-dimensional arrays. In Fig. 8.4, a function of two variables is discretized as a 2D grid which is stored as a 2D array. The stencil used in Fig. 8.4 is 2D and is used to calculate an approximate derivative value (output) at each grid point from function values at the neighboring grid points and the grid point itself. In this chapter, we focus on the computation pattern where a stencil is applied to the all the relevant input grid points to generate the output values at all grid points, which will be referred to as a stencil sweep.

8.2 Parallel stencil – a basic kernel

We first present a basic kernel for stencil sweep. For simplicity, we assume that there is no dependence between output grid points when generating the output grid point values within a stencil sweep. We further assume that the grid point values on the boundaries store boundary conditions and do not change from input to output, as

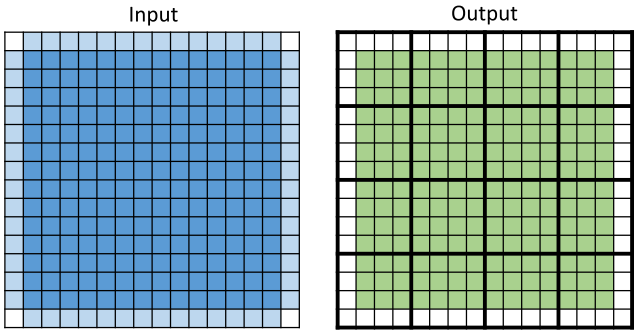


FIGURE 8.5

Simplifying boundary condition: The boundary cells contain boundary conditions that are not updated from one iteration to the next. Thus, only the inner output grid points need to be calculated during each stencil sweep.

```
01  __global__ void stencil_kernel(float* in, float* out, unsigned int N) {
02    unsigned int i = blockIdx.z*blockDim.z + threadIdx.z;
03    unsigned int j = blockIdx.y*blockDim.y + threadIdx.y;
04    unsigned int k = blockIdx.x*blockDim.x + threadIdx.x;
05    if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1&& k >= 1 && k < N - 1) {
06        out[i*N*N + j*N + k] = c0*in[i*N*N + j*N + k]
07                             + c1*in[i*N*N + j*N + (k - 1)]
08                             + c2*in[i*N*N + j*N + (k + 1)]
09                             + c3*in[i*N*N + (j - 1)*N + k]
10                             + c4*in[i*N*N + (j + 1)*N + k]
11                             + c5*in[(i - 1)*N*N + j*N + k]
12                             + c6*in[(i + 1)*N*N + j*N + k];
13    }
14  }
```

FIGURE 8.6

A basic stencil sweep kernel.

illustrated in Fig. 8.5. That is, the shaded inside area in the output grid is calculated whereas the unshaded boundary cells are not. This is a reasonable assumption since stencils are mainly used to solve differential equations with boundary conditions.

Fig. 8.6 shows a basic kernel that performs a stencil sweep. This kernel assumes that each thread block is responsible for calculating a tile of output grid values and each thread is assigned to one of the output grid points. A 2D tiling example of the output grid where each thread block is responsible for a 4×4 output tile is shown in Fig. 8.5. However, since most real-world applications solve 3D differential equations, the kernel in Fig. 8.6 assumes a 3D grid and a 3D 7-point stencil like the one in Fig. 8.3(c). The assignment of threads to grid points is done with the familiar linear expressions involving the `x`, `y`, and `z` fields of `blockIdx`, `blockDim`, and `threadIdx` (lines 02-04). Once each thread has been assigned to a 3D grid point, the input values at that grid point and all neighboring grid points are multiplied by different coefficients (c0 to c6 on lines 06-12) and added. The values of these coefficients may be hard-coded

in the code, or they may be placed in constant memory depending on the desired level of flexibility. These values depend on the specific differential equation that is being solved, as we explained in Section 8.1.

8.3 Memory bandwidth considerations

Following the same approach introduced in Chapter 5 and used in Chapter 7, let us analyze the arithmetic intensity of the 3D 7-point stencil sweep and reason about its demand for memory bandwidth. Let us assume that the size of the grid is $n \times n \times n$. Since we only perform a stencil computation for the internal points in the grid, the total number of grid point values we compute is $(n-2)^3$. To compute each of these values, we must perform 13 floating-point operations (seven multiplications and six additions). Overall, the total number of arithmetic operations performed is $13 \cdot (n-2)^3$ FLOP.

The data that needs to be accessed to perform these computations are the values of the input grid to be loaded and the values of the output grid to be stored. For the input grid, all the points in the grid except the points in the edges and corners of the 3D space are accessed, which total $n^3 - 12 \cdot n + 16$ values. For the output grid, all the internal grid points are accessed, which total $(n-2)^3$ values. Each value is a single-precision floating point number (4 B). Therefore, the total number of bytes accessed is $4 \cdot (n^3 - 12 \cdot n + 16 + (n-2)^3)$.

Overall, the arithmetic intensity of an ideal 3D 7-point stencil computation, where each input grid point is loaded only once, is $\frac{13 \cdot (n-2)^3}{4 \cdot (n^3 - 12 \cdot n + 16 + (n-2)^3)}$. For a very large value of n, this ratio approaches $\frac{13}{8} = 1.625$ FLOP/B. Clearly, this ratio is very small, which makes the 3D 7-point stencil computation memory-bound on modern GPUs.

Let us now calculate the arithmetic intensity for the basic stencil sweep kernel in Fig. 8.6. Each thread in the kernel computes one output element, performing 13 floating-point operations (seven multiplications and six additions), loading seven 4 B input values, and storing one 4 B output value. Therefore, the arithmetic intensity for this kernel is $\frac{13}{(7+1) \cdot 4} = 0.41$ FLOP/B. This value is significantly lower than the ideal intensity of 1.625 FLOP/B. In practice, if we measure the arithmetic intensity of the kernel, it may be somewhat higher because the elements may be found in the L1 or L2 cache. Still, we can more reliably improve the arithmetic intensity of the kernel by applying optimizations to improve its memory bandwidth utilization. In Section 8.4, we apply shared memory tiling like we did in Chapter 7 to elevate the arithmetic intensity of the stencil kernel.

8.4 Shared-memory tiling for stencil sweep

As we have seen in Chapter 5, the arithmetic intensity of a computation can be significantly elevated with shared memory tiling. As the reader probably suspected, the

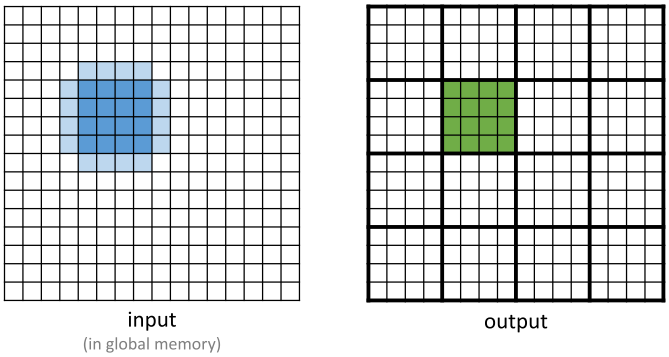


FIGURE 8.7

Input and output tiles for a 2D 5-point stencil.

design of shared memory tiling for stencils is almost identical to that for convolutions. However, there are a few subtle but important differences.

Fig. 8.7 shows the input and output tiles for a 2D 5-point stencil applied to a small grid example. A quick comparison with Fig. 7.11 shows a small difference between convolution and stencil sweep: the input tiles of the 5-point stencil do not include the corner grid points. This property will become important when we explore register tiling later in this chapter. For the purpose of shared memory tiling, we can expect that the input data reuse for a 2D 5-point stencil to be significantly lower than a 3×3 convolution. As we discussed in Chapter 7, the upper bound of the ideal arithmetic intensity of a 2D 3×3 convolution is $\frac{1}{4}\cdot 3^2 = 2.25$ FLOP/B. However, for a 2D 5-point stencil, the ideal arithmetic intensity is only 1.125 FLOP/B. This is because each output grid point value only uses 5 input grid values, compared to the 9 input pixel values in a 3×3 convolution.

The difference is even more pronounced when the number of dimensions and the order of the stencil increases. For example, if we increase the order of the 2D stencil from 1 (1 grid point on each side, 5-point stencil) to 2 (2 grid points on each side, 9-point stencil), the upper bound on the ratio is 2.125 FLOP/B, compared 6.25 FLOP/B for its counterpart 2D 5×5 convolution. This discrepancy is further pronounced when the order of the 2D stencil is increased to 3 (3 grid points on each side, 13-point stencil). The upper bound on the ratio is 3.125 FLOP/B, compared to 12.25 FLOP/B for its counterpart 2D 7×7 convolution.

When we move into 3D, the discrepancy between the arithmetic intensity of stencil sweep and convolution is even more pronounced. For example, a 3D 3rd order stencil (3 points on each side, 19-point stencil) has an upper bound of 4.625 FLOP/B, compared to 85.75 FLOP/B for its counterpart 3D $7 \times 7 \times 7$ convolution. That is, the benefit of loading of an input grid point value into the shared memory for stencil sweep can be significantly lower than that for convolution, especially for 3D which is the prominent use case for stencils. As we will see later, this small but significant difference motivates the use of thread coarsening and register tiling in the 3rd dimension.

```
01 __global__ void stencil_kernel(float* in, float* out, unsigned int N) {
02    int i = blockIdx.z*OUT_TILE_DIM + threadIdx.z - 1;
03    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
04    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;
05    __shared__ float in_s[IN_TILE_DIM][IN_TILE_DIM][IN_TILE_DIM];
06    if(i >= 0 && i < N && j >= 0 && j < N && k >= 0 && k < N) {
07        in_s[threadIdx.z][threadIdx.y][threadIdx.x] = in[i*N*N + j*N + k];
08    }
09    __syncthreads();
10    if(i >= 1 && i < N-1 && j >= 1 && j < N-1 && k >= 1 && k < N-1) {
11      if(threadIdx.z >= 1 && threadIdx.z < IN_TILE_DIM-1 && threadIdx.y >= 1
12        && threadIdx.y<IN_TILE_DIM-1 && threadIdx.x>=1 && threadIdx.x<IN_TILE_DIM-1){
13          out[i*N*N + j*N + k] = c0*in_s[threadIdx.z][threadIdx.y][threadIdx.x]
14                               + c1*in_s[threadIdx.z][threadIdx.y][threadIdx.x-1]
15                               + c2*in_s[threadIdx.z][threadIdx.y][threadIdx.x+1]
16                               + c3*in_s[threadIdx.z][threadIdx.y-1][threadIdx.x]
17                               + c4*in_s[threadIdx.z][threadIdx.y+1][threadIdx.x]
18                               + c5*in_s[threadIdx.z-1][threadIdx.y][threadIdx.x]
19                               + c6*in_s[threadIdx.z+1][threadIdx.y][threadIdx.x];
20     }
21    }
22  }
```

FIGURE 8.8

A 3D 7-point stencil sweep kernel with shared-memory tiling.

Since all the strategies for loading input tiles for convolution directly apply to stencil sweep, we present in Fig. 8.8 a kernel like the convolution kernel in Fig. 7.12 where blocks are of the same size as the input tiles and some of the threads are turned off when calculating output grid point values. The kernel is adapted from the basic stencil sweep kernel in Fig. 8.6 so we will only focus on the changes made during the adaptation. Like the tiled convolution kernel, the tiled stencil sweep kernel first calculates the x, y, and z coordinates of the grid point value that each thread is responsible for loading and possibly computing with. The value 1 is subtracted in each expression because the kernel assumes a 3D 7-point stencil with one grid point on each side (lines 02-04). In general, the value subtracted should be the order of the stencil.

The kernel allocates an `in_s` array in shared memory to hold the input tile for each block (line 05). Every thread loads one input element. Like the tiled convolution kernel, each thread loads the beginning element of the cubic input patch that contains the stencil grid point pattern. Because of the subtraction in lines 02-04, it is possible that some of the threads may attempt to load ghost cells of the grid. The conditions `i >= 0`, `j >= 0`, and `k >= 0` (line 06) guard against these out-of-bound accesses. Because the blocks are larger than the output tiles, it is also possible that some of the threads at the end of the x, y, and z dimensions of a block attempt to access the ghost cells outside the upper bound of each dimension of the grid array. The conditions `i < N`, `j < N`, and `k < N` (line 6) guard against these out-of-bound accesses. All threads in the thread block collaboratively load the input tile into the shared memory (line 07) and use the barrier synchronization to wait until all input tile grid points are in the shared memory (line 09).

Each block calculates its output tile on lines 10-21. The conditions on line 10 reflect the simplifying assumption that the boundary points of both the input grid and

the output hold initial condition values and do not need to be calculated from iteration to iteration by the kernel. Therefore, threads whose output grid points fall on these boundary positions are turned off. Note that the boundary grid points form a layer at the surface of the grid.

The conditions in lines 11-12 turn off the extra threads that were launched just to load the input tile grid points. The conditions allow those threads whose `i`, `j`, and `k` index values fall within the output tile to calculate the output grid points selected by these indices. Finally, each active thread calculates its output grid point value using the input grid points specified by the 7-point stencil while accessing the input values from shared memory instead of global memory.

We can evaluate the effectiveness of shared memory tiling by calculating the arithmetic intensity achieved by this kernel. Recall that the original stencil sweep kernel achieved a ratio of 0.41 FLOP/B when caching is not considered. With the shared memory tiling kernel, let us assume that each input tile is a cube with t grid points in each dimension and each output tile has $t - 2$ grid points in each dimension. Therefore, each block has $(t-2)^3$ active threads calculating output grid point values and each active thread performs 13 floating-point multiplication or addition operations, which is a total of $13 \cdot (t-2)^3$ floating-point arithmetic operations. Moreover, each block loads an input tile by performing t^3 loads and stores an output tile by performing $(t-2)^3$ stores that are 4 bytes each. Therefore, the arithmetic intensity of the tiled kernel can be calculated as:

$$\frac{13 \cdot (t-2)^3}{4 \cdot t^3 + 4 \cdot (t-2)^3} = \frac{13}{8} \cdot \frac{1}{\frac{1}{2} \cdot \frac{t^3}{(t-2)^3} + \frac{1}{2}} \text{ FLOP/B}$$

That is, the larger the value of t, the higher the arithmetic intensity. Intuitively, the larger the output tile, the more the input grid point values are reused. The upper bound on the arithmetic intensity as t increases asymptotically is $\frac{13}{8} = 1.625$ FLOP/B, which is the ideal arithmetic intensity derived in Section 8.3.

Unfortunately, the 1024 limit on the block size on current hardware makes it difficult to have large t values for the kernel in Fig. 8.8. The practical limit on t is 8 because an $8 \times 8 \times 8$ thread block has a total of 512 threads. Furthermore, the amount of shared memory used for the input tile is proportional to t^3. Thus, a larger t dramatically increases the consumption of the shared memory. These hardware constraints force us to use small tile sizes for the tiled stencil kernel.

There are two major disadvantages to the small limit on the t value imposed by the hardware. The first disadvantage is that it limits the arithmetic intensity. For t = 8, the arithmetic intensity for a 7-point stencil is only 0.96 FLOP/B, which is much less than the ideal arithmetic intensity of 1.625 FLOP/B. The arithmetic intensity decreases as the t value decreases because of the halo overhead. As we discussed in Chapter 7, halo elements are less reused than the non-halo elements. As t decreases, the portion of halo elements in the input tile increases so the arithmetic intensity decreases. For example, for a convolution filter with radius 1, a 32×32 2D input tile has 1024 input elements. The corresponding output tile has $30 \times 30 = 900$ elements,

which means that $1024 - 900 = 124$ of the input elements are halo elements. The portion of halo elements in the input tile is about 12%. In contrast, for a 3D stencil of order 1, an $8 \times 8 \times 8$ 3D input tile has 512 elements. The corresponding output tile has $6 \times 6 \times 6 = 216$ elements, which means that $512 - 216 = 296$ of the input elements are halo elements. The portion of halo elements in the input tile is about 58%!

The second disadvantage of a small tile size is that it has an adverse impact on memory coalescing. For an $8 \times 8 \times 8$ tile, every warp which consists of 32 threads is responsible for loading 4 different rows of the tile that each have 8 input values. Hence, on the same load instruction, the threads of the warp access at least 4 distant locations in global memory. These accesses cannot be coalesced and underutilize the DRAM bandwidth. This second disadvantage can be overcome by using non-square tiles such that the x-dimension of the thread block is 32 and threads in the same warp access a contiguous chunk of 32 elements. For example, the block dimension could be set to 32, 8, and 4 in the x, y, and z dimensions. Such a block dimension would result in better performance than the cubic blocks with 8 threads in each dimension because it would improve the coalescing of memory accesses. Nevertheless, even with such a rearrangement of the block dimensions, the size of the output tile remains small overall and the arithmetic intensity remains low. This need for a larger tile size to improve arithmetic intensity motivates the approach that we cover in Section 8.5.

8.5 Thread coarsening

As we mentioned in the previous section, the fact that stencils are typically applied to 3D grids and the sparse nature of the stencil patterns can make stencil sweep a less profitable target of shared memory tiling than convolution. This section presents the use of thread coarsening to overcome the block size limitation by coarsening the work done by each thread from calculating one grid point value to a column of grid point values, as illustrated in Fig. 8.9. Recall from Chapter 6 that with thread coarsening, the programmer partially serializes parallel units of work into each thread and reduces the overhead of parallelization. In this case, the overhead of fine-grain parallelization is having to load the halo elements (which have low data reuse) into more thread blocks.

In Fig. 8.9, we assume that each input tile consists of $t^3 = 6^3 = 216$ grid points. Note that to make the inside of the input tile visible, we have peeled away the front, left, and top layers of the tile. We also assume that each output tile consists of $(t - 2)^3 = 4^3 = 64$ grid points. The x, y, and z directions in the illustration are shown with the coordinate system for the input and the output. Each x-y plane of the input tile consists of $6^2 = 36$ grid points and each x-y plane of the output tile consists of $4^2 = 16$ grid points. The thread block assigned to process this tile consists of the same number of threads as one x-y plane of the input tile (i.e., 6×6). In the illustration, we only show the internal threads of the thread block that are active in computing output tile values (i.e., 4×4).

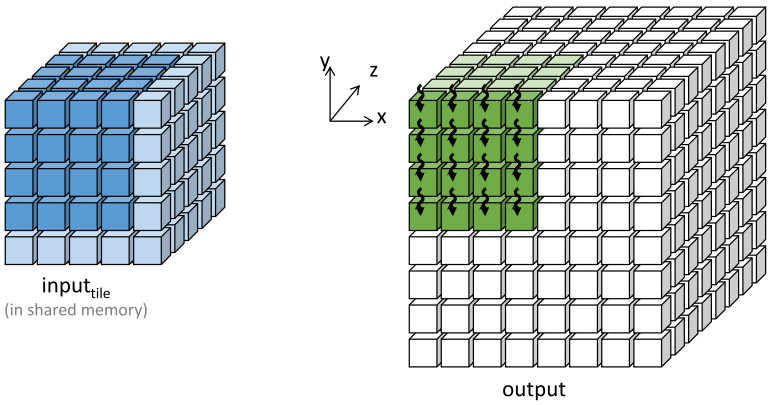


FIGURE 8.9

Thread coarsening in the z direction for a 3D 7-point stencil sweep.

```
01 __global__ void stencil_kernel(float* in, float* out, unsigned int N) {
02    int iStart = blockIdx.z*OUT_TILE_DIM;
03    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
04    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;
05    __shared__ float inPrev_s[IN_TILE_DIM][IN_TILE_DIM];
06    __shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
07    __shared__ float inNext_s[IN_TILE_DIM][IN_TILE_DIM];
08    if(iStart-1 >= 0 && iStart-1 < N && j >= 0 && j < N && k >= 0 && k < N) {
09        inPrev_s[threadIdx.y][threadIdx.x] = in[(iStart - 1)*N*N + j*N + k];
10    }
11    if(iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
12        inCurr_s[threadIdx.y][threadIdx.x] = in[iStart*N*N + j*N + k];
13    }
14    for(int i = iStart; i < iStart + OUT_TILE_DIM; ++i) {
15      if(i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
16            inNext_s[threadIdx.y][threadIdx.x] = in[(i + 1)*N*N + j*N + k];
17      }
18      __syncthreads();
19      if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
20        if(threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1
21           && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
22            out[i*N*N + j*N + k] = c0*inCurr_s[threadIdx.y][threadIdx.x]
23                                 + c1*inCurr_s[threadIdx.y][threadIdx.x-1]
24                                 + c2*inCurr_s[threadIdx.y][threadIdx.x+1]
25                                 + c3*inCurr_s[threadIdx.y+1][threadIdx.x]
26                                 + c4*inCurr_s[threadIdx.y-1][threadIdx.x]
27                                 + c5*inPrev_s[threadIdx.y][threadIdx.x]
28                                 + c6*inNext_s[threadIdx.y][threadIdx.x];
29        }
30      }
31      __syncthreads();
32      inPrev_s[threadIdx.y][threadIdx.x] = inCurr_s[threadIdx.y][threadIdx.x];
33      inCurr_s[threadIdx.y][threadIdx.x] = inNext_s[threadIdx.y][threadIdx.x];
34    }
35 }
```

FIGURE 8.10

Kernel with thread coarsening in the z direction for a 3D 7-point stencil sweep.

Fig. 8.10 shows a kernel with thread coarsening in the z direction for a 3D 7-point stencil sweep. The idea is for the thread block to iterate in the z direction (into the

figure), calculating the values of grid points in one x-y plane of the output tile during each iteration. The kernel first assigns each thread to a grid point in an x-y plane of the output (lines 03-04). Note that `i` is the `z` index of the output tile grid point calculated by each thread. During each iteration, all threads in a block process an x-y plane of an output tile and thus they all calculate output grid points whose `z` indices are identical.

Initially, each block needs to load into the shared memory the three input tile planes that contain all the points needed to calculate the values of the output tile plane that is the closest to the reader (marked with threads) in Fig. 8.9. This is done by having all the threads in the block to load the first layer into the shared memory array `inPrev_s` (lines 08-10) and the second layer into the shared memory array `inCurr_s` (lines 11-13). For the first layer, `inPrev_s` is loaded from the front layer of the input tile that has been peeled off for visibility into the internal layers.

During the first iteration, all threads in a block collaborate to load the third layer needed for the current output tile layer into the shared memory array `inNext_s` (lines 15-17). All threads then wait on a barrier (line 18) until everyone has completed loading the input tile layers. The conditions in lines 19-21 serve the same purpose as their counterparts in the shared memory kernel in Fig. 8.8.

Each thread then calculates its output grid point value in the current output tile plane using the four x-y neighbors stored in `inCurr_s`, the z neighbor in `inPrev_s`, and the z neighbor in `inNext_s`. All threads in the block then wait at the barrier to ensure that every thread completes its calculation before they move on to the next output tile plane. Once off the barrier, all threads collaborate to move the contents of `inCurr_s` to `inPrev_s` and the contents of `inNext_s` to `inCurr_s`. This is because the roles played by the input tile planes change when the threads move by one output plane in the z direction. So, by the end of each iteration, the block has two of the three input tile planes needed for calculating the output tile plane of the next iteration. All threads then move into the next iteration and load the third plane of the input tile needed for the output plane of the iteration. The updated mappings of `inPrev_s`, `inCurr_s`, and `inNext_s` in preparation for the calculation of the second output tile plane are illustrated in Fig. 8.11.

The advantage of the thread coarsening kernel is that it increases the tile size without increasing the number of threads, and it does not require all planes of the input tile to be present in the shared memory simultaneously. The thread block size is now only t^2 instead of t^3, where t is the dimension of the input tile, so we can use a much larger t value. For example, we can use $t = 32$ that would result in a block size of $t^2 = 1024$ threads. With this t value, the arithmetic intensity of the kernel is 1.52 FLOP/B, which is a significant improvement over the 0.96 FLOP/B intensity of the original shared memory tiling kernel with $8 \times 8 \times 8$ input tiles, and closer to the 1.625 FLOP/B ideal intensity. Moreover, at any point in time, only three layers of the input tile need to be in the shared memory. The shared memory capacity requirement is now $3 \cdot t^2$ elements instead of t^3 elements. For $t = 32$, the shared memory consumption is now at a reasonable level of $3 \cdot 32^2 \cdot 4 = 12$ KB per block.

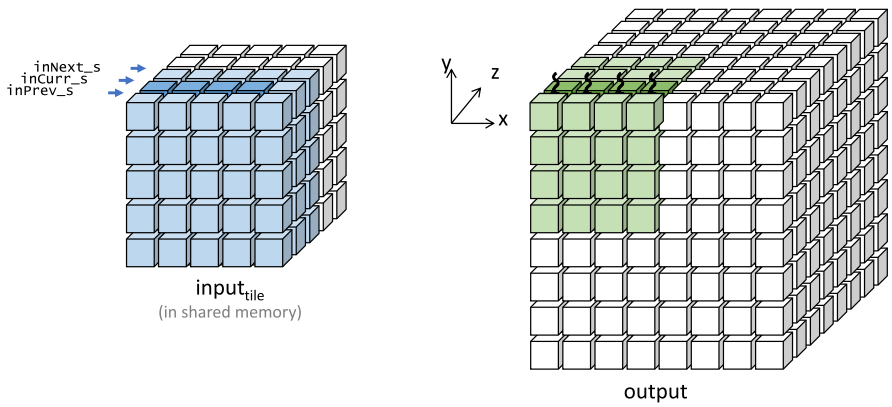


FIGURE 8.11

The mapping of the shared memory arrays to the input tile after the first iteration.

8.6 Register tiling

The special characteristics of some stencil patterns can give rise to new optimization opportunities. Here we present an optimization that can be especially effective for stencil patterns that involve only neighbors along the x, y, and z directions of the center point. All stencils in Fig. 8.3 match this description. The 3D 7-point stencil sweep kernel in Fig. 8.10 reflects this property. Each `inPrev_s` and `inNext_s` element is only used by one thread in the calculation of the output tile grid point with the same x-y indices. Only the `inCurr_s` elements are accessed by multiple threads and truly need to be in the shared memory. The z neighbors in `inPrev_s` and `inNext_s` can instead stay in the registers of the single user thread. Keeping data in registers to reduce shared or global memory accesses is called register tiling, as discussed in Chapter 5 and Chapter 6.

Fig. 8.12 shows a stencil sweep kernel with register tiling applied to the previous and next planes of the input tile. The kernel is built on the thread coarsening kernel in Fig. 8.10 with some simple but important modifications. We focus our discussion on these modifications. First, we create three register variables `inPrev`, `inCurr`, and `inNext` (lines 05, 07, 08). Register variables `inPrev` and `inNext` replace the shared memory arrays `inPrev_s` and `inNext_s`, respectively. In comparison, we keep `inCurr_s` to allow the x-y neighbor grid point values to be shared among threads. Therefore, the amount of shared memory used by this kernel is reduced to one third of that by the kernel in Fig. 8.12.

The initial loading of the previous and current input tile planes (lines 09-15) and the loading of the next plane of the input tile before each new iteration (lines 17-19) are all performed with register variables as destinations. Thus, the three planes of the "active part" of the input tile are held in registers across the threads of the same block. In addition, the kernel always maintains a copy the current plane of the input tile in

```
01 __global__ void stencil_kernel(float* in, float* out, unsigned int N) {
02    int iStart = blockIdx.z*OUT_TILE_DIM;
03    int j = blockIdx.y*OUT_TILE_DIM + threadIdx.y - 1;
04    int k = blockIdx.x*OUT_TILE_DIM + threadIdx.x - 1;
05    float inPrev;
06    __shared__ float inCurr_s[IN_TILE_DIM][IN_TILE_DIM];
07    float inCurr;
08    float inNext;
09    if(iStart-1 >= 0 && iStart-1 < N && j >= 0 && j < N && k >= 0 && k < N) {
10        inPrev = in[(iStart - 1)*N*N + j*N + k];
11    }
12    if(iStart >= 0 && iStart < N && j >= 0 && j < N && k >= 0 && k < N) {
13        inCurr = in[iStart*N*N + j*N + k];
14        inCurr_s[threadIdx.y][threadIdx.x] = inCurr;
15    }
16    for(int i = iStart; i < iStart + OUT_TILE_DIM; ++i) {
17      if(i + 1 >= 0 && i + 1 < N && j >= 0 && j < N && k >= 0 && k < N) {
18            inNext = in[(i + 1)*N*N + j*N + k];
19      }
20      __syncthreads();
21      if(i >= 1 && i < N - 1 && j >= 1 && j < N - 1 && k >= 1 && k < N - 1) {
22        if(threadIdx.y >= 1 && threadIdx.y < IN_TILE_DIM - 1
23           && threadIdx.x >= 1 && threadIdx.x < IN_TILE_DIM - 1) {
24            out[i*N*N + j*N + k] = c0*inCurr
25                                 + c1*inCurr_s[threadIdx.y][threadIdx.x-1]
26                                 + c2*inCurr_s[threadIdx.y][threadIdx.x+1]
27                                 + c3*inCurr_s[threadIdx.y+1][threadIdx.x]
28                                 + c4*inCurr_s[threadIdx.y-1][threadIdx.x]
29                                 + c5*inPrev
30                                 + c6*inNext;
31        }
32      }
33      __syncthreads();
34      inPrev = inCurr;
35      inCurr = inNext;
34      inCurr_s[threadIdx.y][threadIdx.x] = inNext_s;
35    }
35 }
```

FIGURE 8.12

Kernel with thread coarsening and register tiling in the z direction for a 3D 7-point stencil sweep.

the shared memory (lines 14 and 34). That is, the x-y neighbors of the active input tile plane are always available to all threads that need to access these neighbors.

The kernels in Fig. 8.10 and Fig. 8.12 both keep only the "active part" of the input tile in the on-chip memory. The number of planes in the active part depends on the order of the stencil and is 3 for the 3D 7-point stencil. The coarsening and register tiling kernel in Fig. 8.12 has two advantages over the coarsening kernel in Fig. 8.10. First, many reads from and writes to the shared memory are now performed on registers. Since registers have significantly lower latency and higher bandwidth than the shared memory, we expect the code to run faster. Second, each block consumes only one third of the shared memory it did previously. This is, of course, achieved at the cost of three more registers used by each thread, or 3072 more registers per block assuming 32×32 blocks. The reader should keep in mind that register use will become even higher for higher order stencils. If the register usage becomes a problem, one can go back to storing some of the planes in shared memory. This scenario represents

a common trade-off that often needs to be made between shared memory and register usage.

Overall, the data reuse is now spread across registers and the shared memory. The number of global memory accesses has not changed. The overall data reuse if we consider both registers and the shared memory remains the same as the thread coarsening kernel that uses only the shared memory for the input tile. Thus, there is no impact on the consumption of the global memory bandwidth when we add register tiling to thread coarsening.

8.7 Summary

In this chapter, we dived into stencil sweep computation, which seems to be just convolution with special filter patterns. However, because the stencils come from discretization and numerical approximation of derivatives in solving differential equations, they have two characteristics that motivate and enable new optimizations. First stencil sweeps are typically done on 3D grids whereas convolution is typically done on 2D images or a small number of time slices of 2D images. This makes the tiling considerations different between the two and motivate thread coarsening for 3D stencils to enable larger input tiles and higher arithmetic intensity. Second, the stencil patterns can sometimes enable register tiling of input data to further improve data access throughput and alleviate shared memory pressure.

8.8 Exercises

1. Consider a 3D stencil computation on a grid of size $120 \times 120 \times 120$ including boundary cells.
   a. What is the number of output grid points computed during each stencil sweep?
   b. For the basic kernel in Fig. 8.6, what is the number of thread blocks needed assuming a block size of $8 \times 8 \times 8$?
   c. For the kernel with shared memory tiling in Fig. 8.8, what is the number of thread blocks needed assuming a block size of $8 \times 8 \times 8$?
   d. For the kernel with shared memory tiling and thread coarsening in Fig. 8.10, what is the number of thread blocks needed assuming a block size of 32×32?
2. Consider an implementation of 7-point (3D) stencil with shared memory tiling and thread coarsening applied. The implementation is similar to those in Fig. 8.10 and Fig. 8.12, except that the tiles are not perfect cubes. Instead, a thread block size of 32×32 is used as well as a coarsening factor of 16 (i.e., each thread block processes 16 consecutive output planes in the z-dimension).
   a. What is the size of the input tile (in number of elements) that the thread block loads throughout its lifetime?

b. What is the size of the output tile (in number of elements) that the thread block processes throughout its lifetime?
c. What is the arithmetic intensity (in FLOP/B) of the kernel?
d. How much shared memory (in B) is needed by each thread block if register tiling is not used, as in Fig. 8.10?
e. How much shared memory (in B) is needed by each thread block if register tiling is used, as in Fig. 8.12?

This page intentionally left blank

CHAPTER

9 Histogram

An introduction to atomic operations and privatization

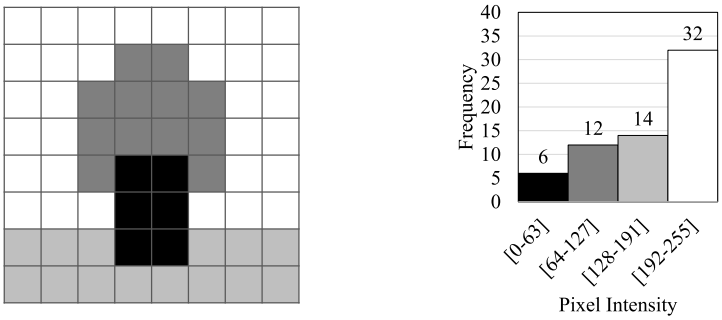
CONTENTS

9.1 Background 202
9.2 Atomic operations and a basic histogram kernel 203
9.3 Latency and throughput of atomic operations 209
9.4 Privatization 211
9.5 Thread coarsening 214
9.6 Thread-level privatization 217
9.7 Summary 219
9.8 Exercises 220
References 220

The parallel computation patterns that we have presented so far all allow the task of computing each output element to be exclusively assigned to, or owned by, a thread. Therefore, these patterns are amenable to the owner-computes rule, where every thread can write into their designated output element(s) without any concern about interference from other threads. This chapter introduces the parallel histogram pattern, where each output element can potentially be updated by any thread. As such, one must take care to coordinate among threads as they update output elements and avoid any interference that corrupts the final result. In practice, there are many other important parallel computation patterns where output interference cannot be easily avoided. The parallel histogram pattern provides an example of such output interference. We will first examine a baseline approach that uses atomic operations to serialize the updates to each element. This baseline approach is simple but inefficient, often resulting in disappointing execution speed. We will then present some widely used optimization techniques, most notably privatization, to significantly enhance execution speed while preserving correctness. The cost and benefit of these techniques depend on the underlying hardware as well as the characteristics of the input data. It is therefore important for a developer to understand the key ideas of these techniques and reason about their applicability under different circumstances.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00019-6

Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.



(a) Image of a tree

(b) Image histogram

FIGURE 9.1

An example of an image histogram.

9.1 Background

A histogram is a display of occurrences of data values in a data set. In the most common form of a histogram, the value intervals are plotted along the horizontal axis and the frequency of data values in each interval is represented as the height of a rectangle, or bar, rising from the horizontal axis. For example, Fig. 9.1(b) shows the histogram of pixel intensity values of the grayscale image of a tree in Fig. 9.1(a). Grayscale images typically consist of pixels with intensity values ranging between 0 and 255 where 0 is black and 255 is white. The histogram in Fig. 9.1(b) splits this range into four intervals with 64 values each. The six black pixels depicting the trunk of the tree are counted in the range [0-63]. The 12 dark gray pixels depicting the leaves of the tree are counted in the range [64-127]. The 14 light gray pixels depicting the grass around the tree are counted in the range [128-191]. The 32 white pixels depicting the sky are counted in the range [192-255].

Histograms provide useful summaries of data sets. In our example, we can see that the image being represented consists of pixels that are heavily concentrated in the bright (higher) intensity intervals and lightly concentrated in the dark (lower) intensity intervals. Such a shape of the histogram is sometimes referred to as a feature of the data set and provides a quick way to determine if there are significant phenomena in the data set, such as if an image has low or high light exposure. As another example, the shape of a histogram of the purchase categories and locations of a credit card account can be used to detect fraudulent usage. When the shape of the histogram deviates significantly from the norm, the system raises a flag of potential concern.

Many application domains rely on histograms to summarize data sets for data analysis. One such area is computer vision. Histograms of different types of object images, such as faces vs. cars, tend to exhibit different shapes. One can plot the histogram of pixel intensity values in an image or an area of an image, as we have seen. A histogram of the sky on a sunny day might have only a small number of very tall bars in the high-intensity intervals of the luminous spectrum. By dividing an image into

```
01  void histogram_sequential(unsigned char* image, unsigned int* bins,
02      unsigned int width, unsigned int height) {
03      for(unsigned int i = 0; i < width*height; ++i) {
04          unsigned char b = image[i];
05          ++bins[b];
06      }
07  }
```

FIGURE 9.2

A simple C function for calculating the histogram of a grayscale image.

subareas and analyzing the histograms for these subareas, one can quickly identify the interesting subareas of an image that potentially contain the objects of interest. The process of computing histograms of image subareas is an important approach to feature extraction in computer vision, where a feature refers to a pattern of interest in images. In practice, whenever there is a large volume of data that needs to be analyzed to distill interesting events, histograms are likely used as a foundational computation. Credit card fraudulence detection and computer vision obviously meet this description. Other application domains with such needs include speech recognition, website purchase recommendations, and scientific data analysis such as correlating heavenly object movements in astrophysics.

Histograms can be easily computed in a sequential manner. Fig. 9.2 shows a sequential function that calculates an image histogram. The C code assumes that the input image comes in the unsigned char array `image` where each element specifies the grayscale intensity of a pixel (line 01). Additionally, the histogram is to be generated into the unsigned int array `bins` (line 01). The number of pixels in the image is specified by the function parameters `width` and `height` (line 02). The for-loop (line 03) sequentially iterates over the image pixels. The image is treated as a 1D array of size `width*height` since the 2D indices of the pixels are not relevant to the computation. It loads the pixel value from the image (line 04) and increments the histogram bin corresponding to that pixel value (line 05). Here, we assume that each histogram bin represents a single pixel value. If the bins represent an interval of pixel values as in Fig. 9.1, then we would divide the pixel value by the interval width before indexing the array of bins.

The C code in Fig. 9.2 is quite simple and efficient. The computational complexity of the algorithm is $O(N)$ where N is the number of input data elements. The `image` array elements are accessed sequentially in the for-loop so the CPU cache lines are well used whenever they are fetched from the system DRAM. The `bins` array is so small that it fits well in the level-one (L1) data cache of the CPU, which ensures fast updates to the `bins` elements. For most modern CPUs, one can expect that execution speed of this code to be memory bound, i.e., limited by the rate at which the data elements can be brought from DRAM into the CPU cache.

9.2 Atomic operations and a basic histogram kernel

The most straightforward approach to parallelizing a histogram computation is to launch as many threads as input elements and have each thread to process one input

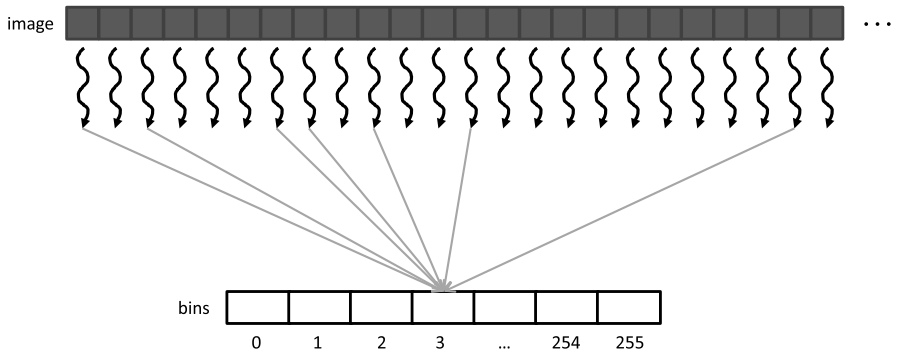


FIGURE 9.3

Basic parallelization of image histogram.

element. Each thread reads its assigned pixel value and increments the appropriate bin for that pixel value. Fig. 9.3 illustrates an example of this parallelization strategy. Note that multiple threads may be assigned to pixels with the same value and need to update the same bin, which is a conflict referred to as output interference. Programmers must understand the concepts of race conditions and atomic operations in order to confidently handle such output interferences in their parallel code.

An increment to an interval counter in the `bins` array is an update, or read-modify-write, operation on a memory location. The operation involves reading the memory location (read), adding one to the original value (modify), and writing the new value back to the memory location (write). Read-modify-write is a frequently used operation for coordinating collaborative activities.

For example, when we make a flight reservation with an airline, we bring up the seat map and look for available seats (read), we pick a seat to reserve (modify), and change the seat status to unavailable in the seat map (write). A bad potential scenario can happen as follows:

- Two customers simultaneously bring up seat map of the same flight.
- Both customers pick the same seat, say 9C.
- Both customers change the status of seat 9C to unavailable.

After the sequence, both customers thought that they have seat 9C. We can imagine that they will have an unpleasant situation when they board the flight and find out that one of them cannot take the reserved seat! Believe it or not, such unpleasant situation indeed happens in real life due to flaws in airline reservation software.

For another example, some stores allow customers to wait for service without standing in line. They ask each customer to take a number from one of the kiosks. There is a display that shows the number that will be served next. When a service agent becomes available, the agent asks the customer to present the ticket that matches the number, verify the ticket, and update the display number to the next higher number. Ideally, all customers will be served in the order they enter the store.

Time	Thread 1	Thread 2
1	(0) Old ← bins[x]	
2	(1) New ← Old + 1	
3	(1) bins[x] ← New	
4		(1) Old ← bins[x]
5		(2) New ← Old + 1
6		(2) bins[x] ← New

(a)

Time	Thread 1	Thread 2
1	(0) Old ← bins[x]	
2	(1) New ← Old + 1	
3		(0) Old ← bins[x]
4	(1) bins[x] ← New	
5		(1) New ← Old + 1
6		(1) bins[x] ← New

(b)

FIGURE 9.4

Race condition in updating a `bins` array element.

An undesirable outcome would be that two customers simultaneously sign in at two kiosks and both receive tickets with the same number. When a service agent calls for that number, both customers expect that they are the one who should receive service.

In both examples, undesirable outcomes are caused by a phenomenon called a read-modify-write race condition, where the outcome of two or more simultaneous update operations varies depending on the relative timing of the operations involved. Some outcomes are correct, and some are incorrect. Fig. 9.4 illustrates a race condition when two threads attempt to update the same `bins` element in our image histogram example. Each row in Fig. 9.4 shows the activity during a time period, with time progressing from top to bottom.

Fig. 9.4(a) depicts a scenario where Thread 1 completes all three parts of its read-modify-write sequence during time periods 1 through 3 before Thread 2 starts its sequence at time period 4. The value in the parenthesis in front of each operation shows the value being written into the destination, assuming the value of `bins[x]` was initially 0. Under this scenario, the value of `bins[x]` afterwards is 2, exactly what one would expect. That is, both threads successfully incremented `bins[x]`. The element value starts with 0 and becomes 2 after the operations complete.

In Fig. 9.4(b), the read-modify-write sequences of the two threads overlap. Note that Thread 1 writes the new value into `bins[x]` at time period 4. When Thread 2 reads `bins[x]` at time period 3, it still has the value 0. As a result, the new value it calculates and eventually writes to `bins[x]` is 1 rather than 2. The problem is that Thread 2 read `bins[x]` too early, before Thread 1 completes its update. The net outcome is that the value of `bins[x]` afterwards is 1, which is incorrect. The update by Thread 1 is lost.

During parallel execution, threads can run in any order relative to each other. In our example, Thread 2 can easily start its update sequence ahead of Thread 1. Fig. 9.5 shows two such scenarios. In Fig. 9.5(a), Thread 2 completes its update before Thread 1 starts its. In Fig. 9.5(b), Thread 1 starts its update before Thread 2 completes its. It should be obvious that the sequences in Fig. 9.5(a) result in correct outcome for `bins[x]` but those in Fig. 9.5(b) produce incorrect outcome.

The fact that the final value of `bins[x]` varies depending on the relative timing of the operations involved indicates that there is a race condition. We can eliminate such variation by eliminating the possible interleaving of operation sequences of Thread 1 and Thread 2. That is, we would like to allow the timings shown in Figs. 9.4(a)

Time	Thread 1	Thread 2
1		(0) Old ← bins[x]
2		(1) New ← Old + 1
3		(1) bins[x] ← New
4	(1) Old ← bins[x]	
5	(2) New ← Old + 1	
6	(2) bins[x] ← New	

(a)

Time	Thread 1	Thread 2
1		(0) Old ← bins[x]
2		(1) New ← Old + 1
3	(0) Old ← bins[x]	
4		(1) bins[x] ← New
5	(1) New ← Old + 1	
6	(1) bins[x] ← New	

(b)

FIGURE 9.5

Race condition scenarios where Thread 2 runs ahead of Thread 1.

and 9.5(a) while eliminating the possibilities shown in Figs. 9.4(b) and 9.5(b). This can be accomplished using atomic operations.

An atomic operation on a memory location is an operation that performs a read-modify-write sequence on the memory location in such a way that no other read-modify-write sequence to the location can overlap with it. That is, the read, modify, and write parts of the operation form an undividable unit, thus the name atomic operation. In practice, atomic operations are realized with hardware support to lock out other operations to the same location until the current operation is complete. In our example, such support eliminates the possibility depicted in Figs. 9.4(b) and 9.5(b) since the trailing thread cannot start its update sequence until the leading thread completes its.

It is important to remember that atomic operations do not enforce particular execution order between threads. In our example, both orders shown in Figs. 9.4(a) and 9.5(a) are allowed by atomic operations. Thread 1 can run either ahead of or behind Thread 2. The rule being enforced is that if both threads perform atomic operations on the same memory location, the atomic operation performed by the trailing thread cannot start until the atomic operation of the leading thread completes. This effectively serializes the atomic operations being performed on a memory location.

Atomic operations can be performed in CUDA using C++ atomic objects or atomic reference objects and functions on those objects.[1] CUDA extends the C++ atomic library, and these extensions can be accessed by including the relevant library as follows:

```
#include <cuda/atomic>
```

[1] We encourage readers who are not familiar with the concepts of the C++ objects, classes, and namespaces to refer to the C++ documentation [1] for a deeper understanding. Broadly, a C++ class defines how to create a reference to an object that consists of a collection of data and a set of method functions that can operate on the data. The class is *generic* as it is defined as a *template*. Intuitively, defining a class as a template enables library developers to specify *generic objects* that can be specialized to meet the needs of user applications in terms of the types of data and the options for the functions in the objects thus created.

```
01  __global__ void histogram_kernel(unsigned char* image, unsigned int* bins,
02      unsigned int width, unsigned int height) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < width*height) {
05          unsigned char b = image[i];
06          cuda::atomic_ref<unsigned int, cuda::thread_scope_device> bins_ref(bins[b]);
07          bins_ref.fetch_add(1, cuda::memory_order_relaxed);
08      }
09  }
```

FIGURE 9.6

A CUDA kernel for calculating a histogram.

With the `atomic` library included, the first step for performing an atomic operation on an object is to create an atomic reference to that object using the `cuda::atomic_ref` class.[2] An atomic reference enables us to perform atomic operations on the object. The `cuda::atomic_ref` class is defined in the CUDA C++ library as follows:

```
template <typename T, cuda::thread_scope>
class cuda::atomic_ref;
```

The template for the `atomic_ref` class takes two parameters that reside between angle brackets ("<" and ">"). The first template parameter `T` specifies the type of the object. Examples of commonly used types are `int`, `unsigned int`, and `float`. The second template parameter `cuda::thread_scope` specifies the scope of threads that can synchronize with one another using the method functions defined for the class. Common examples of scopes include within the same thread block, within the same GPU device, and across all GPU devices within the same multi-GPU system. The `"cuda::"` prefix indicates that `atomic_ref` is declared in the CUDA namespace. Interested readers are referred to the CUDA programming guide for more details.

We are now ready to implement a basic parallel histogram approach as illustrated in Fig. 9.3. Fig. 9.6 shows such a CUDA kernel that uses atomic operations to ensure the correctness of concurrent increments to the `bins` elements. The code is similar to the sequential code in Fig. 9.2 with two key distinctions. The first distinction is that the loop over input elements is replaced with a thread index calculation (line 03) and a boundary check (line 04) to assign a thread to each input `image` element. The second distinction is that the following increment expression in Fig. 9.2:

```
++bins[b];
```

is replaced with the code on lines 06-07 of Fig. 9.6. This code ensures that any simultaneous updates to the same `bins` array element by different threads are performed atomically and properly serialized.

[2] Alternatively, an object can be declared atomic from the moment it is created using the `cuda::atomic` class. Using `cuda::atomic` ensures that all accesses to the object are atomic from its creation. In our case, we use atomic references because we may wish to access the bins in non-atomic ways in other parts of the code.

The first step in performing the update operation atomically is to construct an atomic reference to `bins[b]` named `bins_ref` (line 06). Recall that the `cuda::atomic_ref` class is defined as a template. On line 06, the first template parameter, `unsigned int` specifies that the atomic operation is to be performed on an unsigned integer object. In this case, the unsigned integer object is one of the elements of the `bins` array. The second template parameter, `cuda::thread_scope_device`, is a predefined compile-time constant indicating that the atomic operation needs to allow threads in different thread blocks to synchronize with each other. That is, the increments to the same `bins` element are serialized, whether these increments are performed by threads within the same block or across different blocks in the same GPU device. As a result, an atomic reference `bins_ref` is constructed for performing atomic operations on the selected `bins[b]`. Although the constructor on line 06 appears to be complex, it actually just serves as a means to instruct the CUDA C++ compiler on how to generate code for line 07 and adds little execution cost at run time.

Having constructed the atomic reference, we can invoke its method functions to perform atomic operations on the object that it references. The CUDA atomic reference class provides functions that perform different kinds of operations such as: addition, minimum, maximum, and, or, xor, etc. In our histogram kernel, we need to perform an integer increment operation, i.e., fetching the old value, adding 1 to generate the new value, and writing the new value to the `bins` elements. The method function that performs atomic addition operations is `fetch_add`, which is defined as a generic method function of the `cuda::atomic_ref` class in the CUDA C++ library as follows:

```
T fetch_add(T arg, cuda::std::memory_order order)
```

The `fetch_add` function takes two parameters. The first parameter `arg` specifies the value to be added, which is 1 on our case. Note that the value must be of the same type (`T`) as specified for the first template parameter of the atomic reference class that defines the `fetch_add` function. This ensures that the value to be added to the object is of the same type as the object itself. On line 07 of Fig. 9.6, the `fetch_add` function is invoked on an unsigned integer object `bins[b]`. The compiler will treat the first argument passed, i.e., the constant value 1, as an unsigned integer constant.

The second parameter `order` is a compile-time constant defined in the C++ standard library for specifying the restrictions on how the atomic operation may be reordered with respect to other memory accesses performed by the invoking thread. The compiler or the hardware may reorder the execution of memory accesses to optimize the performance of the code. The `order` parameter specifies which kind of reordering is allowed based on the intent of the programmer. In our case, there is only one other memory access performed by the invoking thread: the one that reads the `image[i]` element into local variable `b`. Since `b` is used to select the object on which the atomic operation is to be performed, the compiler will not be able to reorder the atomic operation with respect to the memory access. At run time, the use of `b` to select the object presents instruction-level data dependencies to the hardware scheduler. That is, the atomic operation will naturally be forced to wait for the completion of

the memory access that reads the `image[i]` element. Thus, there is no need to specify any additional restriction to prevent them from being reordered with respect to each other. The compile-time constant `cuda::memory_order_relaxed` used in line 07 of Fig. 9.6 indicates that there is no further restrictions beyond what is already present at the instruction-level. In Chapter 11, we will encounter a use case where specifying additional restrictions on reordering memory accesses is needed.

The `fetch_add` function is defined to return a value, which is of the same type (`T`) as the first template parameter. The returned value is the old value of the object before the first parameter value is atomically added to it. This return value is not used on line 07 of Fig. 9.6 because the thread is only tasked to increment the selected `bins` element. It is not concerned with the old value of the element before the increment operation. In Chapter 12 and Chapter 18, we will see other use cases where the return value of the `fetch_add` function may be needed.

9.3 Latency and throughput of atomic operations

The atomic operation used in the kernel of Fig. 9.6 ensures the correctness of updates by serializing any simultaneous updates to a location. As we all know, serializing any portion of a massively parallel program can drastically increase the execution time and reduce the execution speed of the program. Therefore, it is important that such serialized operations account for as little execution time as possible.

As we learned in Chapter 5, the access latency to data in DRAMs can take hundreds of clock cycles. In Chapter 4, we learned that GPUs use zero-cycle context switching to tolerate such latency. In Chapter 6, we learned that as long as we have many threads whose memory access latencies can overlap with each other, the execution speed is limited by the throughput of the memory system. Thus, it is important that GPUs make full use DRAM bursts, banks, and channels to achieve high memory access throughput.

It should be clear to the reader at this point that the key to high memory access throughput is to have many DRAM accesses that are simultaneously in progress. Unfortunately, this strategy breaks down when many atomic operations update the same memory location. In this case, the read-modify-write sequence of a trailing thread cannot start until the read-modify-write sequence of a leading thread is complete. As shown in Fig. 9.7, the execution of atomic operations at the same memory location is such that there can be only one in progress. The duration of each atomic operation is approximately the latency of a memory load (left section of the atomic operation time) plus the latency of a memory store (right section of the atomic operation time). The length of these time sections of each read-modify-write operation, usually hundreds of clock cycles, defines the minimal amount of time that must be dedicated to servicing each atomic operation and limits the throughput, or the rate at which atomic operations can be performed.

For example, assume a memory system with 64-bit (8-byte) Double Data Rate DRAM interface per channel, 8 channels, 1 GHz clock frequency, and typical access latency of 200 cycles. The peak access throughput of the memory system is

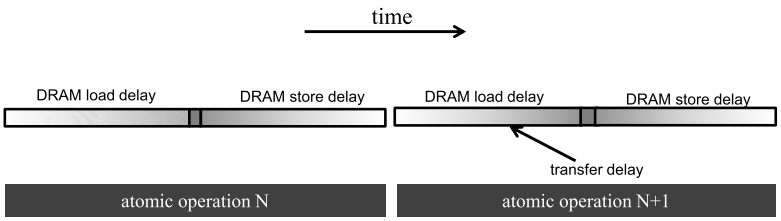


FIGURE 9.7

Throughput of atomic operation is determined by the memory access latency.

8 (bytes/transfer)*2 (transfers per clock per channel)* 1 G (clocks per second) * 8 (Channels) = 128 GB/sec. Assuming each data accessed is 4 bytes, the system has a peak access throughput of 32 G data elements per second.

However, when performing atomic operations on a particular memory location, the highest throughput one can achieve is one atomic operation every 400 cycles (200 cycles for the read and 200 cycles for the write). This translates into a throughput of 1/400 atomics/clock * 1 G (clocks/second) = 2.5 M atomics/second. This is dramatically lower than most users expect from a GPU memory system. Furthermore, the long latency of the sequence of atomic operations will likely dominate the kernel execution time and can dramatically lower the execution speed of the kernel.

In practice, not all atomic operations will be performed on a single memory location. In our image histogram example, the histogram has 256 bins. If the input pixels are uniformly distributed in brightness, the atomic operations are evenly distributed among the `bins` elements. This would boost the throughput to 256*2.5 M = 640 M atomic operations per second. In reality, the boost factor tends to be much lower than the number of intervals in the histogram because the pixels tend to have biased distribution in brightness. For example, in Fig. 9.1, we see that pixels in the example image are heavily biased towards the brighter levels. The heavy contention traffic to update these high-brightness bins will likely cause significant reduction to the achievable throughput.

One approach to improving the throughput of atomic operations is to reduce the access latency to the heavily contended locations. Cache memories are the primary tool for reducing memory access latency. For this reason, modern GPUs allow atomic operations with device scope to be performed in the last-level cache, which is shared among all SMs. During an atomic operation, if the updated variable is found in the last-level cache, it is updated in the cache. If it cannot be found in the last level cache, it triggers a cache miss and is brought into the cache where it is updated. Since the variables updated by atomic operations tend to be heavily accessed by many threads, these variables tend to remain in the cache while the threads perform atomic operations on them. Because the access time to the last-level cache is in tens of cycles rather than hundreds of cycles, the throughput of atomic operations is improved by at least an order of magnitude with this approach. This is an important reason why most modern GPUs support atomic operations in the last-level cache.

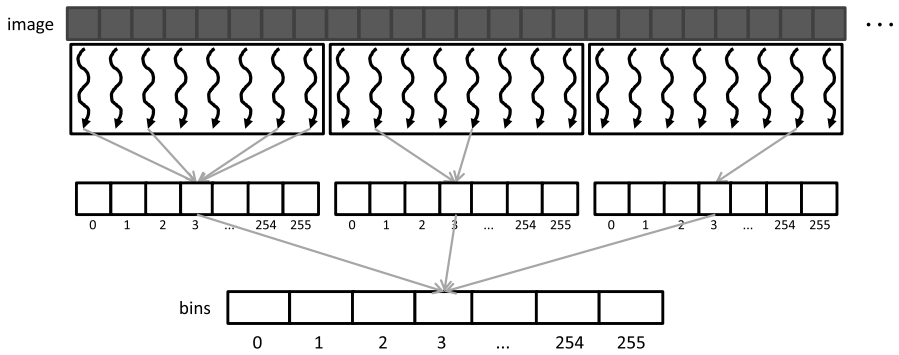


FIGURE 9.8

Private copies of the histogram reduce contention of atomic operations.

9.4 Privatization

Another approach to improving the throughput of atomic operations is to alleviate the contention by directing the traffic away from the heavily contended locations. This can be achieved with a technique referred to as *privatization* that is commonly used to address problems in parallel computing with heavy output interference. The idea is to replicate highly contended output data structures into private copies so that each subset of threads can update its private copy. The benefit is that the private copies can be accessed with much less contention and often at much lower latency. These private copies can dramatically increase the throughput for updating the data structures. The downside is that the private copies need to be merged into the final output data structure after the computation completes. One must carefully balance between the level of contention and the merging cost. Therefore, in massively parallel systems, privatization is typically done for subsets of threads rather than individual threads.

In our text histogram example, we can create multiple private histograms and designate a subset of threads to update each of them. For example, we can create two private copies and have even-index blocks to update one of them and odd-index blocks to update the other. As another example, we can create four private copies and have blocks with index i update the private copy numbered i mod 4. A common approach is to create a private copy for each thread block. This approach has multiple advantages that we will see later. Regardless of the approach, once the private histograms have been computed, they are merged into the public copy by being added together. This merger is facilitated by the fact that addition is commutative and associative, so the order in which the private copies are added into the public copy does not matter.

Fig. 9.8 shows an example of how privatization is applied to the image histogram example from Fig. 9.3. In this example, the threads are organized into thread blocks, where each thread block consists of 8 threads (in practice, thread blocks are much

```
01  __global__ void histogram_kernel(unsigned char* image, unsigned int* bins,
02      unsigned int* bins_pool, unsigned int width, unsigned int height) {
03
04      unsigned int* bins_priv = &bins_pool[blockIdx.x*NUM_BINS];
05
06      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
07      if(i < width*height) {
08          unsigned char b = image[i];
09          cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
10              bins_priv_ref(bins_priv[b]);
11          bins_priv_ref.fetch_add(1, cuda::memory_order_relaxed);
12      }
13      __syncthreads();
14
15      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
16          if(bins_priv[b] > 0) {
17              cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
18                  bins_ref(bins[b]);
19              bins_ref.fetch_add(bins_priv[b], cuda::memory_order_relaxed);
20          }
21      }
22
23  }
```

FIGURE 9.9

Histogram kernel with private bins per thread block in global memory.

larger). Each thread block receives a private copy of the histogram that it updates. As shown in the figure, rather than having contention across all the threads that update the same histogram bin, contention will only be experienced between threads in the same block, and across blocks when merging the private copies at the end.

Fig. 9.9 presents a simple kernel that creates and associates a private copy of the histogram to every block. Under this scheme, up to 1024 threads (the maximum allowed per block on current hardware) would work on the same private copy of the histogram. In this kernel, the private histograms are in the global memory. These private copies will likely be cached for reduced latency and improved throughput.

The kernel in Fig. 9.9 takes an additional parameter `bins_pool` (line 02) that points to a pool of global memory allocated by the host to hold all the private copies of the histogram, which amounts to `gridDim.x*NUM_BINS` integers. Each thread adds an offset of `blockIdx.x*NUM_BINS` to the `bins_pool` array to find the location of its block's private bins `bins_priv`. This offset shifts the position to the private copy for the block to which the thread belongs.

The next part of the kernel in Fig. 9.9 (lines 06-12) is similar to the kernel in Fig. 9.6 with two key distinctions. The first distinction is that `bins_priv` is used instead of `bins` such that each thread block updates its own private set of bins. In this case, the level of contention is reduced by a factor that is approximately the number of active blocks across all SMs. The effect of reduced contention can result in a significant improvement in the update throughput for the kernel.

The second distinction is that the atomic reference has a scope of `cuda::thread_scope_block` (line 09), i.e., a block scope instead of a device scope. In other words, the atomic operations invoked with this reference only need to be synchronized with other threads in the same thread block, not the entire device. Depending on the

hardware implementation of atomic operations, such narrower scope may be able to achieve shorter latencies and thus higher throughput. This narrower scope is possible because only threads in the same block contend on the bins in the block's private histogram.

At the end of the execution, each thread block will commit the values in its private copy to the public copy pointed to by `bins` (lines 15-21). The threads in the block first wait for each other to finish updating the private copy (line 13). Next, the threads iterate over the private histogram bins (line 15), each thread being responsible for committing one or more of the private bins. This iterative approach is used to accommodate an arbitrary number of bins that may be larger than the number of threads in the block. During each iteration, each thread reads the value of the private bin it is responsible for and checks if the bin is non-zero (line 16). If so, the thread commits the value by atomically adding it to the public copy (lines 17-19). Note that the addition needs to be performed atomically because threads from multiple blocks can be simultaneously performing the addition on the same location. Since only one thread from each block will be updating any given `bins` array element during this phase of the kernel execution, the level of contention for each location is modest.

One benefit of creating a private copy of the histogram on a per-thread-block basis is that the threads can use `__syncthreads()` to wait for each other before committing the private copy to the public copy. Another benefit is that, if the number of bins in the histogram is small enough, the private copy of the histogram can be declared in the block's shared memory. Recall that any reduction in latency directly translates into improved throughput of atomic operations on the same memory location. Hence, the latency for accessing memory can be dramatically reduced by placing bins in the shared memory. Shared memory is private to each SM and has very short access latency (a few cycles). This reduced latency directly translates into dramatic increase in the throughput of atomic operations.

In GPUs that support thread block clusters and distributed shared memory, it is also possible to create a private copy of the histogram on a per-cluster basis. In this case, thread blocks in the same cluster can use cluster-level barrier synchronization to wait for each other before committing the private copy to the public copy. They can also place the private copy of the histogram in their distributed shared memory, which allows them to support a larger number of bins.

Fig. 9.10 shows a privatized histogram kernel that stores the private copies of the histogram in the shared memory instead of global memory. The key difference from the kernel code in Fig. 9.9 is that the private copy of the histogram is allocated in shared memory in the `bins_s` array (line 04) and initialized to 0 in parallel by the threads of the block (line 05-07). The barrier synchronization (line 08) ensures that all bins of the private histogram have been properly initialized before any thread starts to update them. The remaining code is identical to Fig. 9.9, except for one distinction. The distinction is that the first atomic operation is performed on the elements of the shared memory array `bins_s` (line 14) and the private bin value is later read from there (line 20).

```
01  __global__ void histogram_kernel(unsigned char* image, unsigned int* bins,
02      unsigned int width, unsigned int height) {
03
04      __shared__ unsigned int bins_s[NUM_BINS];
05      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
06          bins_s[b] = 0;
07      }
08      __syncthreads();
09
10      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
11      if(i < width*height) {
12          unsigned char b = image[i];
13          cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
14             bins_s_ref(bins_s[b]);
15          bins_s_ref.fetch_add(1, cuda::memory_order_relaxed);
16      }
17      __syncthreads();
18
19      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
20          if(bins_s[b] > 0) {
21              cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
22                  bins_ref(bins[b]);
23              bins_ref.fetch_add(bins_s[b], cuda::memory_order_relaxed);
24          }
25      }
26
27  }
```

FIGURE 9.10

Histogram kernel with private bins per thread block in shared memory.

9.5 Thread coarsening

We have seen that privatization is effective at reducing the contention of atomic operations, and that storing the privatized histogram in the shared memory reduces the latency of each atomic operation. However, the overhead of privatization is the need to initialize private copies of the histogram and commit these private copies to the public copy. These initialize and commit operations are done once per thread block. Hence, the more thread blocks we use, the larger this overhead is. The overhead is usually worth paying when the thread blocks are executed in parallel. However, if the number of thread blocks launched exceeds the number that can be executed simultaneously by the hardware, the hardware scheduler will serialize these thread blocks. In this case, the privatization overhead is incurred unnecessarily.

We can reduce the overhead of privatization via thread coarsening. In other words, we can reduce the number of private copies that need to be initialized and committed to the public copy by reducing the number of blocks and having each thread process multiple input elements. In this section, we look at two strategies for assigning multiple input elements to a thread: contiguous partitioning and interleaved partitioning.

Fig. 9.11 shows an example of the contiguous partitioning strategy. The input is partitioned into contiguous segments and each segment is assigned to a thread. Fig. 9.12 shows the histogram kernel with coarsening applied using the contiguous partitioning strategy. The difference from Fig. 9.10 is on lines 10-12. In Fig. 9.10, the

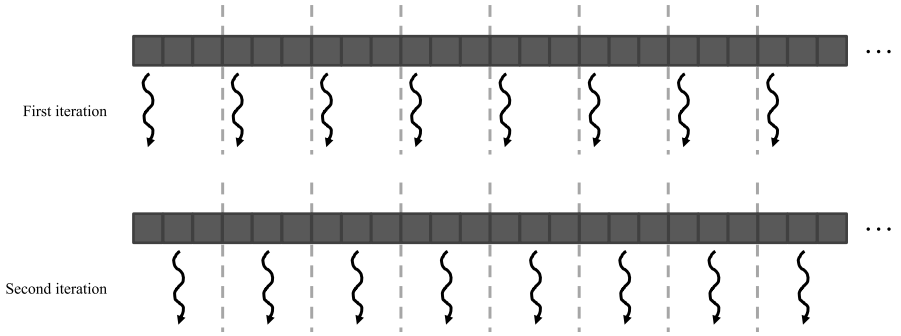


FIGURE 9.11

Contiguous partitioning of input elements.

```
01  __global__ void histogram_kernel(unsigned char* image, unsigned int* bins,
02   unsigned int width, unsigned int height) {
03
04      __shared__ unsigned int bins_s[NUM_BINS];
05      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
06          bins_s[b] = 0;
07      }
08      __syncthreads();
09
10      unsigned int segment = COARSE_FACTOR*blockIdx.x*blockDim.x;
11      for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
12          unsigned int i = segment + threadIdx.x*COARSE_FACTOR + c;
13          if(i < width*height) {
14              unsigned char b = image[i];
15              cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
16                  bins_s_ref(bins_s[b]);
17              bins_s_ref.fetch_add(1, cuda::memory_order_relaxed);
18          }
19      }
20      __syncthreads();
21
22      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
23          if(bins_s[b] > 0) {
24              cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
25                  bins_ref(bins[b]);
26              bins_ref.fetch_add(bins_s[b], cuda::memory_order_relaxed);
27          }
28      }
29
30  }
```

FIGURE 9.12

Histogram kernel with coarsening using contiguous partitioning.

input element index `i` corresponds to the global thread index, so that each thread received one input element. In Fig. 9.11, each thread block is assigned to a contiguous block segment whose size is the coarsening factor `COARSE_FACTOR` times the number of threads in the block `blockDim.x`. The offset of the segment, `segment`, is found by multiplying the block index by the segment size (line 10). Within each block segment, each thread is assigned to a contiguous thread segment whose size is `COARSE_FACTOR`. A coarsening loop iterates through the thread segment (line 11). In each loop itera-

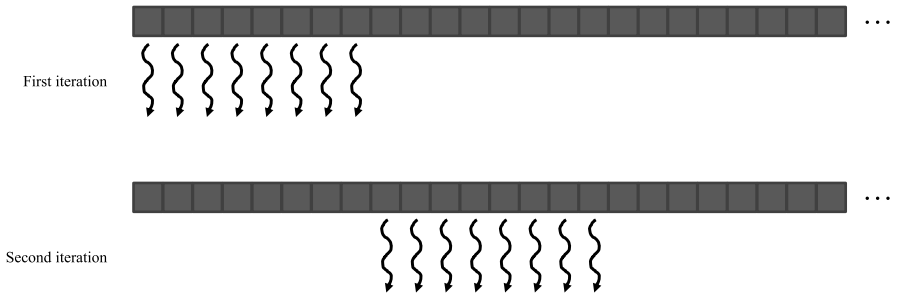


FIGURE 9.13

Interleaved partitioning of input elements.

tion, the input element index `i` for a thread is found by adding the offset of the block segment, `segment`, the offset of the thread's segment within the block, i.e., `threadIdx.x*COARSE_FACTOR`, and the coarsening counter `c` (line 12). Therefore, each thread accesses a contiguous segment of input elements.

Partitioning data into contiguous segments is conceptually simple and intuitive. On a CPU, where parallel execution typically involves a small number of threads, contiguous partitioning is often the best performing strategy since the sequential access pattern by each thread makes good use of cache lines. Since each CPU cache typically supports only a small number of threads, there is little interference in cache usage by different threads. The data in cache lines, once brought in for a thread, can be expected to remain for the subsequent accesses.

In contrast, contiguous partitioning on GPUs results in a sub-optimal memory access pattern. As we learned in Chapter 6, the large number of simultaneously active threads in an SM typically causes so much interference in the cache that one cannot expect data to remain in the cache for all the sequential accesses by a single thread. Instead, we need to make sure that threads in a warp access consecutive locations to enable memory coalescing. This observation motivates interleaved partitioning.

Fig. 9.13 shows an example of the interleaved partitioning strategy. During the first iteration, the eight threads access pixels 0 through 7. With memory coalescing, the elements will be fetched with the smallest number of memory transactions. During the second iteration, the four threads access characters 8 through 15 in a coalesced manner as well. It should be clear why this is called interleaved partitioning: the partitions to be processed by different threads are interleaved with each other. Obviously, this is a toy example, and in reality, there will be many more threads. There are also more subtle performance considerations. For example, each thread should process four pixels (a 32-bit word) in each iteration to fully utilize the interconnect bandwidth between the caches and the SMs.

Fig. 9.14 shows the histogram kernel with coarsening applied using the interleaved partitioning strategy. The difference from Fig. 9.12 is only on line 12. In the first iteration of the loop, when `c` is 0, each thread access data within its block segment using its local thread index (`threadIdx.x`): thread 0 accesses element `segment` + 0, thread 1 accesses element `segment` + 1, thread 2 accesses element `segment` + 2, etc.

```
01  __global__ void histogram_kernel(unsigned char* image, unsigned int* bins,
02   unsigned int width, unsigned int height) {
03
04      __shared__ unsigned int bins_s[NUM_BINS];
05      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
06          bins_s[b] = 0;
07      }
08      __syncthreads();
09
10      unsigned int segment = COARSE_FACTOR*blockIdx.x*blockDim.x;
11      for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
12          unsigned int i = segment + c*blockDim.x + threadIdx.x;
13          if(i < width*height) {
14              unsigned char b = image[i];
15              cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
16                  bins_s_ref(bins_s[b]);
17              bins_s_ref.fetch_add(1, cuda::memory_order_relaxed);
18          }
19      }
20      __syncthreads();
21
22      for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
23          if(bins_s[b] > 0) {
24              cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
25                  bins_ref(bins[b]);
26              bins_ref.fetch_add(bins_s[b], cuda::memory_order_relaxed);
27          }
28      }
29
30  }
```

FIGURE 9.14

Histogram kernel with coarsening using interleaved partitioning.

Thus, all threads jointly process the first `blockDim.x` elements of the block segment. In the second iteration, when `c` is 1, all threads add `blockDim.x` to their indices and jointly process the next section of `blockDim.x` elements. In general, in iteration `c`, each thread adds `c*blockDim.x` to the index of its first element (`segment + threadIdx.x`) to find the element it should process in that iteration.

9.6 Thread-level privatization

Applying thread coarsening to the histogram kernel creates the opportunity for another level of privatization at the level of the thread. Since each thread processes multiple input elements after coarsening, we can create a copy of the histogram that is private to the thread which the thread can update without using atomic operations. When the thread finishes processing all its input elements, it can then commit its thread-level private histogram to the block-level private histogram using atomic operations.

One limiting factor for this optimization is the number of bins in the histogram. If the number of bins is very small, then it would be feasible to allocate a copy of the histogram for each thread. The thread may even place this copy in the shared memory or its own registers. However, if the number of bins in the histogram is large, allocating a private copy of the histogram for each thread would be prohibitively expensive.

```
__global__ void histogram_kernel(unsigned char* image, unsigned int* bins,
    unsigned int width, unsigned int height) {

    __shared__ unsigned int bins_s[NUM_BINS];
    for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
        bins_s[b] = 0;
    }
    __syncthreads();

    unsigned int segment = COARSE_FACTOR*blockIdx.x*blockDim.x;
    if(segment + threadIdx.x < width*height) {
        unsigned int b = image[segment + threadIdx.x];
        unsigned int bin_r = 1;
        for(unsigned int c = 1; c < COARSE_FACTOR; ++c) {
            unsigned int i = segment + c*blockDim.x + threadIdx.x;
            if(i < width*height) {
                unsigned char bNext = image[i];
                if(bNext == b) {
                    ++bin_r;
                } else {
                    cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
                        bins_s_ref(bins_s[b]);
                    bins_s_ref.fetch_add(bin_r, cuda::memory_order_relaxed);
                    b = bNext;
                    bin_r = 1;
                }
            }
        }
        cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
            bins_s_ref(bins_s[b]);
        bins_s_ref.fetch_add(bin_r, cuda::memory_order_relaxed);
    }
    __syncthreads();

    for(unsigned int b = threadIdx.x; b < NUM_BINS; b += blockDim.x) {
        if(bins_s[b] > 0) {
            cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
                bins_ref(bins[b]);
            bins_ref.fetch_add(bins_s[b], cuda::memory_order_relaxed);
        }
    }

}
```

FIGURE 9.15

Histogram kernel with thread-level privatization.

One workaround is to not store a private copy of every bin, but rather, only the frequently accessed bins. This workaround is possible because some data sets have a large concentration of identical data values in localized areas. For example, in pictures of the sky, there can be large patches of pixels of identical value. Such high concentration of identical values causes heavy contention and reduced throughput of parallel histogram computation. However, it presents a great opportunity for thread-level privatization. For such data sets, a simple and yet effective optimization is for each thread to aggregate consecutive updates into a single update if they are updating the same element of the histogram.

Thread-level privatization reduces the number of atomic operations to the highly contended histogram bins by having each thread to accumulate the count in a thread-private bin, thus improving the effective throughput of the computation.

Fig. 9.15 shows an image histogram kernel where each thread privatizes a single bin of the histogram: the most recently accessed one. The key changes compared to

the kernel in Fig. 9.14 are as follows. The first pixel that a thread is responsible for is loaded (line 12) after a boundary check (line 11) and a thread-level private bin `bin_r` is initialized to 1 (line 13). Next, the coarsening loop starts at `c = 1` (line 14) since the first input pixel has already been loaded.

Whenever a new pixel is loaded to `bNext` (line 17), it is compared to the pixel value corresponding to the thread-level private bin (line 18). If they are the same, the thread simply increments the thread-level private bin (line 19). If they are different, the thread uses an atomic operation to add the thread-level private bin's old value to the block-level private bin (lines 21-23). It then updates the thread-level private bin's pixel value to the one it just loaded (line 24) and resets the bin's counter to 1 (line 25). With this scheme, the update is always at least one element behind. In the extreme case where there are no similar pixels found by the thread, all the updates will simply be always one loop iteration behind. For this reason, after a thread has completed processing all input pixels and exited the loop, the thread needs to perform one more atomic operation for the last pixel (lines 29-31).

One important observation is that the kernel with thread-level privatization requires more statements and variables. Thus, if the similarity between consecutive pixel values is low, the kernel may execute at slower speed than the previous kernel. However, if the data distribution exhibits high repetitiveness, thread-level privatization may result in significantly higher speed. The added if-statement can potentially exhibit control divergence. However, if there is either no similarity or heavy similarity, there will be little control divergence since the threads would be all either incrementing the private bin or performing the atomic operation. In the case where control divergence does occur, the control divergence is likely compensated by the reduced contention.

9.7 Summary

Computing histograms is important for analyzing large data sets. It also represents an important class of parallel computation patterns where the output location of each thread is data-dependent, which makes it infeasible to apply owner-computes rule. It is therefore a natural vehicle for introducing the concept of read-modify-write race conditions, and the practical use of atomic operations that ensure the integrity of concurrent read-modify-write operations to the same memory location.

Unfortunately, as we explained in this chapter, atomic operations have much lower throughput than simpler memory read or write operations because their throughput is approximately the inverse of two times the memory latency. Thus, in the presence of heavy contention, histogram computation can have surprisingly low computation throughput. Privatization is introduced as an important optimization technique that systematically reduces contention and can further enable the use of shared memory which supports low latency and thus high throughput. In fact, supporting fast atomic operations among threads in a block is an important use case of the shared memory. Thread coarsening is also applied to reduce the number of private copies that need

to be merged, and different coarsening strategies that use contiguous partitioning and interleaved partitioning were compared. Finally, thread-level privatization can also lead to further reduction in contention of atomic operations, especially for data sets with high similarity between different data elements.

9.8 Exercises

1. Assume that each atomic operation in a DRAM system has a total latency of 100 ns. What is the maximal throughput we can get for atomic operations on the same global memory variable?
2. For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 4 ns to complete in L2 cache and 100 ns to complete in DRAM. Assume that 90% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?
3. In Question 1, assume that a kernel performs 5 floating-point operations per atomic operation. What is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
4. In Question 1, assume that we privatize the global memory variable into shared memory variables in the kernel and the shared memory access latency is 1 ns. All original global memory atomic operations are converted into shared memory atomic operation. For simplicity, assume that the additional global memory atomic operations for accumulating privatized variables into the global variable add 10% to the total execution time. Assume that a kernel performs 5 floating-point operations per atomic operation. What is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
5. Consider a histogram kernel that processes an input with 524,288 elements to produce a histogram with 128 bins. The kernel is configured with 1024 threads per block.
   a. What is the total number of atomic operations performed on global memory by the kernel in Fig. 9.6 where no privatization, shared memory, and thread coarsening are used?
   b. What is the maximum number of atomic operations that may be performed on global memory by the kernel in Fig. 9.10 where privatization and shared memory are used, but not thread coarsening?
   c. What is the maximum number of atomic operations that may be performed on global memory by the kernel in Fig. 9.14 where privatization, shared memory, and thread coarsening are used with a coarsening factor of 4?

References

[1] C++ reference, https://cppreference.com, 2025.

CHAPTER

Reduction
And mitigating control divergence

10

CONTENTS

10.1 Background 221
10.2 Reduction trees 223
10.3 A simple reduction kernel 227
10.4 Reducing control divergence 230
10.5 Reducing memory access divergence 234
10.6 Reducing global memory accesses 236
10.7 Reducing synchronization overhead with warp-level primitives 237
10.8 Further reducing synchronization overhead with two-stage warp-wide reduction 241
10.9 Reduction for arbitrary length inputs 244
10.10 Thread coarsening to reduce overhead 246
10.11 Summary 249
10.12 Exercises 249
References 250

A reduction operation derives a single value from a list of values. The single value could be the sum, the maximum, the minimum, etc. of all the values in the list. The value can also be of various types: integer, single-precision floating point, double-precision floating point, half-precision floating point, characters, etc. All these types of reductions share the same computation structure. Like histogram, reduction is an important computation pattern as it generates a summary from a large amount of data. Parallel reduction is an important parallel pattern that requires parallel threads to coordinate with each other to produce correct results. Such coordination must be done carefully to avoid performance bottlenecks commonly found in parallel computing systems. Parallel reduction is therefore a good vehicle for illustrating these performance bottlenecks and introducing techniques for mitigating them.

10.1 Background

A reduction transforms a set of items into a single item based on a binary operator. For example, a sum reduction can be defined for a set of floating-point numbers based on the addition operator that produces the sum of all the floating-point numbers in

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00020-2
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

```
01    sum = 0.0f;
02    for(i = 0; i < N; ++i) {
03          sum += input[i];
04    }
```

FIGURE 10.1

A simple sum reduction sequential code.

```
01    acc = IDENTITY;
02    for(i = 0; i < N; ++i) {
03          acc = Operator(acc, input[i]);
04    }
```

FIGURE 10.2

The general form of a reduction sequential code.

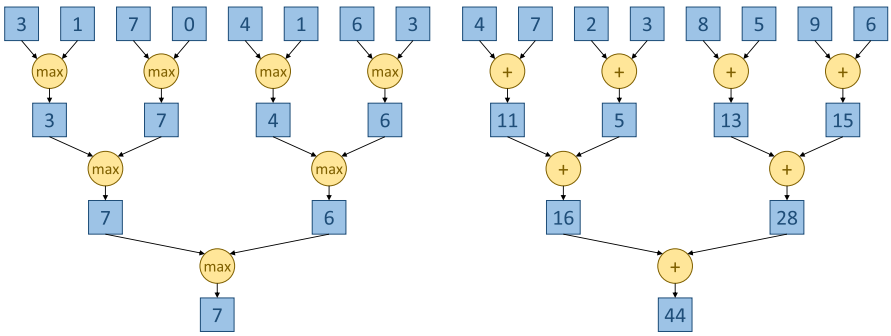
the set. For the set $\{7.0, 2.1, 5.3, 9.0, 11.2\}$, the sum reduction would produce $7.0 + 2.1 + 5.3 + 9.0 + 11.2 = 34.6$.

Reduction can be performed by sequentially going through every element of the array. Fig. 10.1 shows a sequential sum reduction code in C. The code initializes the result variable `sum` to the addition's identity value 0.0.[1] It then uses a `for`-loop to iterate through the input array that holds the list of values. During the ith iteration, the code performs an addition operation on the current value of sum and the ith element of the input array. In our example, after iteration 0, the sum variable contains the $0.0 + 7.0 = 7.0$. After iteration 1, the sum variable contains $7.0 + 2.1 = 9.1$. Thus, after each iteration, another value of the input array would be added (accumulated) to the sum variable. After iteration 5, the sum variable contains 34.6 which is the reduction result.

Reduction can be defined for many other operators. A product reduction can be defined for a floating-point multiplication operator whose identity value is 1.0. A product reduction of a set of floating-point numbers is the product of all these numbers. A minimum (min) reduction can be defined for a minimum comparison operator that returns the smaller value of the two inputs. A maximum (max) reduction can be defined for a maximum (max) comparison operator that returns the larger value of the two inputs. For real numbers, the identity value for the minimum (minimum) operator is $+\infty$ ($-\infty$). A min (max) reduction of a set of floating-point numbers produces the smallest (largest) number in the list.

Fig. 10.2 shows a general form of sequential reduction for an operator, which is defined as a function that takes two inputs and returns a value. When an element is visited during an iteration of the for-loop, the action to take depends on the type of

[1] When an identity value for a binary operator is an input to the operator, the output is always the value of the other input. For example, a floating-point addition operator has an identity value of 0.0 because an addition of any floating-point value v and value 0.0 always results in value v itself. Note that using an identity value in the code example is a matter of convenience. If the operator does not have an identity value, we can simply initialize the sum as element 0 and start the loop iterations from `i=1`.



(a) A parallel max reduction tree (b) A parallel sum reduction tree

FIGURE 10.3

Examples of parallel reduction trees for max and sum reductions.

reduction being performed. For example, for a max reduction, the `Operator` function performs a comparison between the two inputs and returns the larger value of the two. For a min reduction, the values of the two inputs are compared by the `Operator` function and the smaller value is returned. The sequential algorithm ends when all the elements are visited by the for-loop. For a set of N elements, the for-loop iterates N iterations and produces the reduction result at the exit of the loop.

Reduction is such a common operation that some programming languages have reduction functions built into their standard libraries. For example, the C++ standard library has `std::accumulate` and `std::reduce` that perform different variants of the reduction operation. Similarly, CUDA libraries such as Thrust [1,2] and CUB [3] provide built-in implementations of parallel reduction that are highly optimized for GPUs. Still, this chapter covers how to parallelize and optimize reduction on GPUs from scratch because reduction is an excellent example for demonstrating many parallelization and optimization techniques.

10.2 Reduction trees

Parallel reduction algorithms have been studied extensively in the literature. The basic concept of parallel reduction is illustrated in Fig. 10.3, where time progresses downwards in the vertical direction and the activities that threads perform in parallel in each time step are shown in the horizontal direction.

Let us focus on the max reduction tree in Fig. 10.3(a). During the first round (time step), four max operations are performed in parallel on the four pairs of the original elements. These four operations produce partial reduction results – the four larger values from the four pairs of original elements. During the second time step, two max operations are performed in parallel on the two pairs of partial reduction results and produce two partial results that are even closer to the final reduction result. These

two partial results are the largest value of the first four elements and the largest value of the second four elements in the original input. During the third and final time step, one max operation is performed to generate the final result, the largest value 7, from the original input.

Note that the order of performing the operations is changed from a sequential reduction algorithm to a parallel reduction algorithm. For example, for the input at the top of Fig. 10.3(a), a sequential max reduction like the one in Fig. 10.2 would start first by comparing the identity value ($-\infty$) in acc with the input value 3 and update acc with the winner which is 3. It will then compare the value of acc (3) with the input value 1 and update acc with the winner which is 3. This will be followed by comparing the value of acc (3) with the input value 7 and updating acc with the winner which is 7. However, in the parallel reduction of Fig. 10.3(a), the input value 7 is first compared with the input value 0 before it is compared against the maximum of 3 and 1. Being able to change the order of performing the operations in this way is essential for parallelizing the reduction operation.

To be able to reorder the operations, parallel reduction assumes that the order of applying the operator to the input values does not matter. In a max reduction, no matter what the order is for applying the operator to the input values, the outcome will be the same. This property is guaranteed mathematically if the operator is associative. An operator Θ is associative if $(a\Theta b)\Theta c = a\Theta(b\Theta c)$. In other words, if an operator is associative, one can insert parentheses at arbitrary positions of an expression involving the operator and the results are all the same. With this equivalence relation, one can convert any order of operator application to any other order while preserving the equivalence of results.

For example, integer addition is associative, i.e., $(1 + 2) + 3 = 1 + (2 + 3)$.[2] On the other hand, integer subtraction is not associative, i.e., $(1 - 2) - 3 \neq 1 - (2 - 3)$. One can convert the subtraction operator into addition of the negative form of the second operand to induce associativity. For example, $(1 - 2) - 3$ can be converted into $(1 + (-2)) + (-3) = 1 + ((-2) + (-3))$.

As for real numbers, real number addition is associative in the mathematical sense, but strictly speaking, floating-point additions are not associative due to potentially different rounding results for different ways of introducing parentheses. Some applications accept floating-point operation results to be the same if they are within a tolerable difference from each other. Such tolerance allows developers and compiler optimizations to treat floating-point addition as an associative operator for practical purposes. Interested readers are referred to Appendix A for a detailed treatment of this topic.

The conversion from the sequential reduction in Fig. 10.2 to the reduction tree in Fig. 10.3(a) requires that the operator be associative. We can think of a reduction as

[2] Although integer addition is associative in the mathematical sense, it is not exactly associative in C++ because the order of operations may impact whether or not an overflow or underflow occurs, which has undefined behavior. However, if the programmer is certain that overflow or underflow will not occur, they can treat integer addition as associative for practical purposes.

a list of operations. The difference of ordering between Fig. 10.2 and Fig. 10.3(a) is caused by inserting parentheses at different positions of the same list. For Fig. 10.2, the parentheses are:

((((((3 max 1) max 7) max 0) max 4) max 1) max 6) max 3

Whereas the parentheses for Fig. 10.3(a) are:

((3 max 1) max (7 max 0)) max ((4 max 1) max (6 max 3))

We will apply an optimization in Section 10.4 that not only rearranges the order of applying the operator, but also rearranges the order of the operands. To rearrange the order of the operands, this optimization further requires the operator to be commutative. An operator is commutative if $a \Theta b = b \Theta a$. That is, the position of the operands can be rearranged in an expression and the results are the same. Note that the max operator is also commutative, as are many other operators such as min, addition, and multiplication.[3] Obviously, not all operators are commutative. For example, addition is commutative, i.e., $1 + 2 = 2 + 1$, whereas integer subtraction is not, i.e., $1 - 2 \neq 2 - 1$.

The parallel reduction pattern in Fig. 10.3(a) is called a reduction tree because it looks like a tree whose leaves are the original input elements and whose root is the final result. The term reduction tree is not to be confused with tree data structures where the nodes are linked either explicitly with pointers or implicitly with assigned positions. In reduction trees, the edges are conceptual, reflecting information flow from operations performed in one time step to those in the next time step.

To compare the speed of sequential and parallel reduction implementations, we introduce the notions of *work* and *span*. The *work* performed by an implementation refers to the number of operations it performs, whereas the *span* of an algorithm refers to the time it needs to perform those operations assuming it has infinite resources. For example, the vector addition kernel in Chapter 2, which adds two vectors of size N, launches N threads. During execution, each thread performs one operation that involves reading two input vector elements, performing an addition, and writing one output element. Hence, this kernel performs $O(N)$ work, i.e., a total of N operations are performed across all threads.

We refer to the time period for a thread to perform an operation as a *step*. When multiple threads are involved, a step is defined as a time period in which all or a subset of the threads each execute an operation in parallel. The span of a computation is measured as the number of steps taken by an implementation assuming an unlimited amount of execution resources. The sequential vector addition implementation has a

[3] Although max and min with real numbers are commutative in the mathematical sense, they are not exactly commutative for floating point numbers because of the behavior when one operand is a `NaN`. Interested readers are referred to the IEEE Floating-Point Standard document for the detailed behavior of the comparison operators when one or more of the operands are NaNs. However, if the programmer is certain that their input does not contain `NaN` values, they can treat max and min as commutative for practical purposes.

span of $O(N)$ because the N operations are performed one after the other. On the other hand, the parallel vector addition kernel has a span of $O(1)$ because the N operations would all be performed in parallel.

There is no distinction between work and span when analyzing sequential algorithms because operations are performed sequentially so the number of steps taken by the algorithm is directly proportional to the amount of work it performs. For example, a sequential vector addition performs $O(N)$ work and takes $O(N)$ steps since the addition operations are performed one after the other.

Like sequential addition, a sequential reduction operation also performs $O(N)$ work and takes $O(N)$ steps. In comparison, let us analyze the work and the span of the parallel reduction tree algorithm. In Fig. 10.3(a), the parallel reduction tree performs seven operations to reduce eight values: four max operations during the first time step, two during the second, and one during the third. In general, for N input values, a reduction tree performs $\frac{1}{2}N$ operations during the first round, $\frac{1}{4}N$ operations in the second round, and so on. Therefore, the total number of operations performed is defined by the geometric series $\frac{1}{2}N + \frac{1}{4}N + \frac{1}{8}N + \cdots \frac{1}{N}N = N - 1$ operations. In other words, the parallel reduction tree algorithm performs $O(N)$ work, which is the same as the sequential reduction algorithm. We will see in Chapter 11 that parallel algorithms do not always perform the same amount of work as sequential algorithms, giving rise to the notion of *work efficiency*.

As for span, a parallel reduction tree takes $log_2 N$ steps to complete the reduction process for N input value. Since each step performs one operation, the parallel reduction tree has a span of $O(log\ N)$. For example, it can reduce $N = 1024$ input values in just $log_2\ N = 10$ steps assuming that there are enough execution resources. During the first step, we need $\frac{1}{2}N = 512$ execution resources. Note that the number of resources needed diminishes quickly as we progress in time steps. During the final step, we only need to have one execution resource. The level of parallelism at each step is the same as the number of execution units required. It is interesting to calculate the average level of parallelism across all steps. The average parallelism is the total number of operations performed divided by the number of steps, which is $(N-1)/log_2 N$. For $N = 1024$, the average parallelism in the 10 steps is 102.3, while the peak parallelism is 512 (during the first step). Such variation in the level of parallelism and resource consumption across steps makes reduction trees a challenging parallel pattern for parallel computing systems.

Parallel Reduction in Sports and Competitions

Parallel reduction has been used in sports and competitions long before the dawn of computing. Fig. 10.4 shows the schedule of the 2010 World Cup quarter finals, semi finals, and the final game in South Africa. It should be clear that this is just a rearranged reduction tree. The elimination process of the World Cup is a maximum reduction where the maximum operator returns the team that "beats" the other team. The tournament "reduction" is done in multiple rounds. The teams are divided into pairs. During the first round, all pairs play in parallel.

Winners of the first round advance to the second round, whose winners advance to the third round, etc. With 8 teams entering a tournament, four winners will emerge from the first round (quarter finals in Fig. 10.4), two from the second round (semi finals in Fig. 10.4), and one final winner (champion) from the third round (final in Fig. 10.4). Each round is a step of the reduction process. The tournament in Fig. 10.4 has a span of four steps.

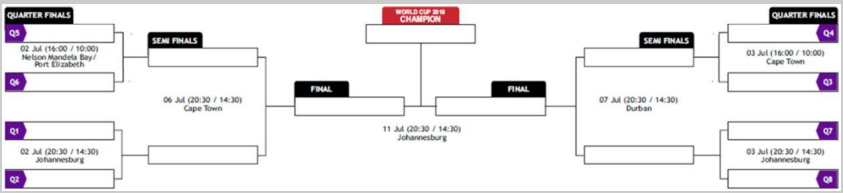


FIGURE 10.4

The 2010 World Cup Finals as a reduction tree.

It should be easy to see that even with 1024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough resources to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on. With enough resources, even with sixty thousand teams, we can determine the final winner in just 16 rounds. It is interesting to note that while reduction trees can greatly speedup the reduction process, they also consume quite a bit of resources. In the World Cup example, a game requires a large soccer stadium, officials, and staff as well as hotels and restaurants to accommodate the massive number of fans in the audience. The four quarter finals in Fig. 10.4 were played in three cities (Nelson Mandela Bay/Port Elizabeth, Cape Town, and Johannesburg) that all together provided enough resources to host the four games. Note that the two games in Johannesburg were played on two different days. Thus, sharing resources between two games made the reduction process take more time. We will see similar tradeoffs in computation reduction trees.

Fig. 10.3(b) shows a sum reduction tree example for 8 input values. Everything we presented so far about work and span for max reduction trees is also applicable to sum reduction trees. It takes $log_2 8 = 3$ time steps to complete the reduction using a maximum of four adders and their associated resources for reading and writing data. We will use this example to illustrate the design of sum reduction kernels.

10.3 A simple reduction kernel

We are now ready to develop a simple kernel to perform the parallel sum reduction tree shown in Fig. 10.3(b). Since the reduction tree requires collaboration across all

```
01    __global__ void SimpleSumReductionKernel(float* input, float* output) {
02        unsigned int i = 2*threadIdx.x;
03        for (unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
04            if (threadIdx.x % stride == 0) {
05                input[i] += input[i + stride];
06            }
07            __syncthreads();
08        }
09        if(threadIdx.x == 0) {
10            *output = input[0];
11        }
12    }
```

FIGURE 10.5

A simple sum reduction kernel.

threads which is not possible across an entire grid, we start by implementing a kernel that performs a sum reduction tree within a single block. That is, for an input array of N elements, we call this simple kernel and launch a grid with one block of $\frac{1}{2}N$ threads. Since a block can have up to 1024 threads, we can process up to 2048 input elements. We will eliminate this limitation in Section 10.10. During the first step, all $\frac{1}{2}N$ threads participate and each thread adds two elements to produce $\frac{1}{2}N$ partial sums. During the next step, half of the threads drop off and only $\frac{1}{4}N$ threads continue to participate to produce $\frac{1}{4}N$ partial sums. This process continues until the last step where only one thread remains and produces the total sum.

Fig. 10.5 shows the code of the simple sum kernel function, and Fig. 10.6 illustrates the execution of the reduction tree implemented by this code. Note that in Fig. 10.6, the time progresses from top to bottom. We assume that the input array is in the global memory and a pointer to the array is passed as an argument when the kernel function is called. Each thread is assigned to a data location that is `2*threadIdx.x` (line 02). That is, the threads are assigned to the even locations in the input array: thread 0 to input[0], thread 1 to input[2], thread 2 to input[4], etc., as shown in the top row of Fig. 10.6. Each thread will be the "owner" of the location it is assigned to and will be the only thread that writes into that location. The design of the kernel follows the "owner computes" approach where every data location is owned by a unique thread and can be updated by only that owner thread.

The top row of Fig. 10.6 shows the assignment of threads to the locations of the input array whereas each of the subsequent rows shows the writes to the input array locations at each step, i.e., iteration of the for-loop in Fig. 10.5 (line 03). The locations that the kernel overwrites in each iteration of the for-loop are marked as filled positions of the input array in Fig. 10.6. For example, at the end of the first iteration, locations with even indices are overwritten with the partial sums of pairs of the original elements (0-1, 2-3, 4-5, etc.) in the input array. At the end of the second iteration, the locations whose indices are multiples of 4 are overwritten with the partial sum of four adjacent original elements (0-3, 4-7, etc.) in the input array.

In Fig. 10.5, a stride variable is used for the threads to reach for the appropriate partial sums for accumulation into their owner locations. The stride variable is initialized to 1 (line 03). The value of the stride variable is doubled in each iteration so the

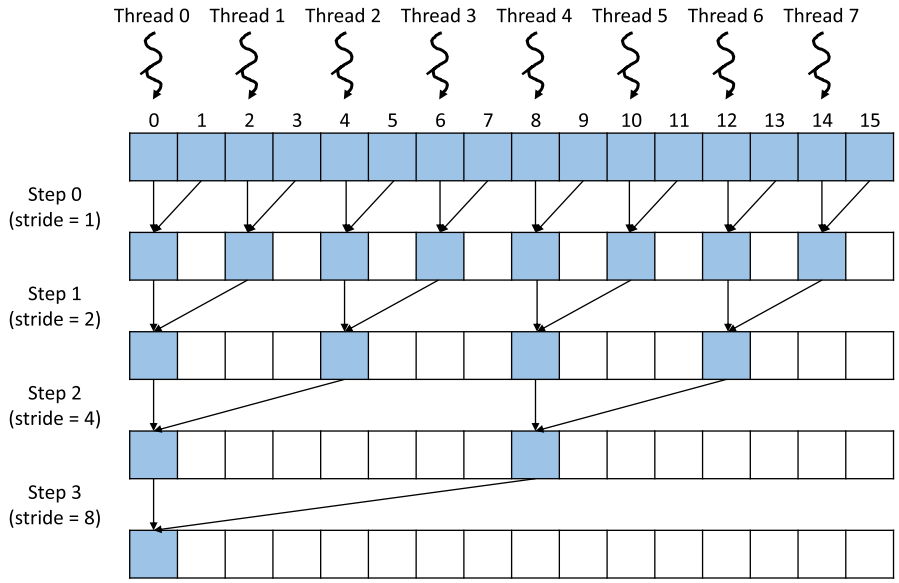


FIGURE 10.6

The assignment of threads ("owners") to the input array locations and progress of execution over time for the SimpleSumReductionKernel in Fig. 10.5. The time progresses from top to bottom and each level corresponds to one iteration of the for-loop.

stride variable value will be 1, 2, 4, 8, etc. until it becomes greater than `blockIdx.x`, the total number of threads in the block. As shown in Fig. 10.6, each active thread in an iteration uses the stride variable to add into its owned location the input array element that is of distance stride away. For example, in iteration 0, Thread 0 uses stride value 1 to add `input[1]` into its owned position `input[0]`. This updates `input[0]` to the partial sum of the first pair of original values of `input[0]` and `input[1]`. In iteration 1, Thread 0 uses stride value 2 to add `input[2]` to `input[0]`. At this time, `input[2]` contains the sum of the original values of `input[2]` and `input[3]` and `input[0]` contains the sum of the original values of `input[0]` and `input[1]`. So, after iteration 1, `input[0]` contains the sum of the original values of the first four elements of the input array. After the last iteration, `input[0]` contains the sum of all original elements of the input array and thus the result of the sum reduction. This value is written by Thread 0 as the final output (line 10).

We now turn to the if-statement in Fig. 10.5 (line 04). The condition of the if-statement is set up to select the active threads in each iteration. As shown in Fig. 10.6, during iteration n, the threads whose thread index (`threadIdx.x`) values are multiples of 2^n are to perform addition. The condition is `threadIdx.x % stride == 0`, which tests if the thread index value is a multiple of the value of the stride variable. Recall that the value of stride is 1, 2, 4, 8, $\cdots$ through the iterations, or 2^n for iteration n. Thus, the condition indeed tests if the thread index values are multiples of 2^n.

Recall that all threads execute the same kernel code. Those threads whose thread index value satisfies the if-condition are the active threads that perform the addition statement (line 05). Those whose thread index values fail to satisfy the condition are the inactive threads that skip the addition statement. As the iterations progress, fewer and fewer threads remain active. At the last iteration, only Thread 0 remains active and produces the sum reduction result.

The `__syncthreads()` statement (line 07 of Fig. 10.5) in the for-loop ensures that all partial sums calculated by the iteration have been written into their destination locations in the input array before any one of threads is allowed to begin the next iteration. In this way, all threads that enter an iteration will be able to correctly use the partial sums produced in the previous iteration. For example, after the first iteration, the even elements will be replaced by the pairwise partial sums. The `__syncthreads()` statement ensures that all these partial sums from the first iteration are indeed written to the even locations of the input array and ready to be used by the active threads in the second iteration.

It is also noteworthy that data written into the global memory by a thread should not be read by other threads without executing a memory fence operation. In this case, since the threads writing and reading the same location across iterations are from the same thread block, the `__syncthreads()` function at the end of each iteration also serves as a memory fence operation for threads within the same block. Thus, data written before a call to `__syncthreads()` will be correctly read by threads in the same block after the execution of `__syncthreads()`. The rules that dictate the expected behavior of data exchange between threads through memory accesses are called the memory consistency model. Interested readers are referred to the CUDA programming guide for more details on the CUDA memory consistency model.

10.4 Reducing control divergence

The kernel code in Fig. 10.5 implements the parallel reduction tree in Fig. 10.6 and produces the expected sum reduction result. Unfortunately, its management of active and inactive threads in each iteration results in a high degree of control divergence. For example, as shown in Fig. 10.6, only those threads whose `threadIdx.x` values are even will execute the addition statement during the second iteration. As we explained in Chapter 4, control divergence can significantly reduce the execution resource utilization efficiency, or the percentage of resources used in generating useful results. In this example, all 32 threads in a warp consume execution resources but only half of them are active, wasting half of the execution resources. The waste of execution resources due to divergence increases over time. During the third iteration, only one fourth of the threads in a warp are active, wasting three quarters of the execution resources. During iteration 6, only one out of the 32 threads in a warp are active, wasting $\frac{31}{32}$ of the execution resources.

If the size of the input array is greater than 32, then entire warps will become inactive after the sixth iteration. For example, for input size of 256, 128 threads or

4 warps would be launched. All four warps would have the same divergence pattern as we explained in the previous paragraph for iterations 1 through 6. During the 7th iteration, Warp 1 and Warp 3 would become completely inactive and thus exhibit no control divergence. On the other hand, Warp 0 and Warp 2 would have only one active thread, exhibiting control divergence and wasting $\frac{31}{32}$ of the execution resource. During the 8th iteration, only Warp 0 would be active, exhibiting control divergence and wasting $\frac{31}{32}$ of the execution resource.

In general, the execution resource utilization efficiency for an input array of size N can be calculated as the ratio of the total number of active threads to the total number of execution resources consumed. The total number of execution resources consumed is proportional to the total number of active warps across all iterations since every active warp, no matter how few of its threads are active, consumes full execution resources. This number can be calculated as follows:

$$(N/64 * 6 + N/64 * \tfrac{1}{2} + N/64 * \tfrac{1}{4} + \cdots + 1) * 32$$

Here, $N/64$ is the total number of warps launched since $N/2$ threads will be launched and every 32 threads form a warp. The $N/64$ term is multiplied by 6 because all launched warps are active for 6 iterations. After the sixth iteration, the number of warps is reduced by half in each successive iteration. The expression in the parenthesis gives the total number of active warps across all the iterations. The second term reflects that each active warp consumes full execution resources for all 32 threads regardless of the number of active threads in these warps. For input array size of 256, the consumed execution resource is (4*6 + 2 + 1)*32 = 864.

The number of execution results committed by the active threads is the total number of active threads across all iterations:

$$N/64 * (32 + 16 + 8 + 4 + 2 + 1) + N/64 * \tfrac{1}{2} * 1 + N/64 * \tfrac{1}{4} * 1 + \cdots + 1$$

The terms in the parenthesis give the active threads in the first six iterations for all $N/64$ warps. Starting at the 7th iteration, the number of active warps is reduced by half in each iteration and there is only one active thread in each active warp. For input array size of 256, the total number of committed results is $4 * (32 + 16 + 8 + 4 + 2 + 1) + 2 + 1 = 255$. This result should be intuitive because the total number of operations needed to reduce 256 values is 255 operations.

Putting the previous two results together, we find that the execution resource utilization efficiency for an input array size of 256 is 255/864 = 0.30. This ratio states that the parallel execution resources did not achieve their full potential in speeding up this computation. On average, only about 30% of the resources consumed contributed to the sum reduction result. That is, we are only using 30% of the hardware's potential to speed up the computation.

Based on the analysis, we see that there is wide-spread control divergence across warps and over time. As the reader might have wondered, there is a better way to assign threads to the input array locations to reduce control divergence and improve resource utilization efficiency. The problem with the assignment illustrated in Fig. 10.6 is that the partial sum locations become increasingly distant from each other

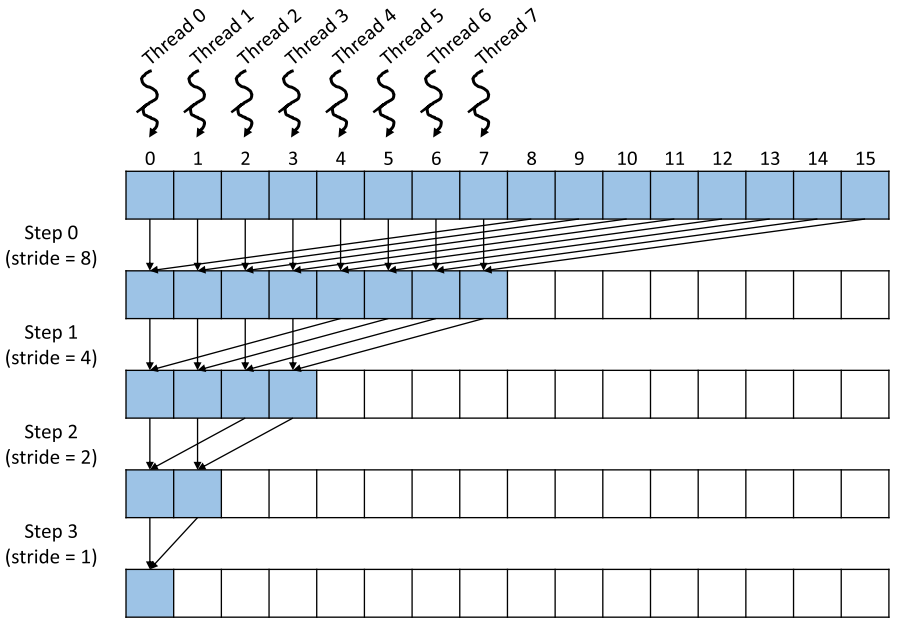


FIGURE 10.7

A better assignment of threads to input array locations for reduced control divergence.

and thus the active threads that own these locations are also increasingly distant from each other as time progresses. This increasing distance between active threads contributes to the increasing level of inactivity and resource underutilization.

There is, indeed, a better assignment strategy that significantly reduces control divergence. The idea is that we should arrange the threads and their owned positions so that they can remain close to each other as time progresses. That is, we would like to have the stride value to decrease, rather than increase, over time. The revised assignment strategy is shown in Fig. 10.7 for an input array of 16 elements. Here, we assign the threads to the first half of the locations. During the first iteration, each thread reaches halfway across the input array and adds an input element to its owned location. In our example, Thread 0 adds input[8] to its owned position input[0], Thread 1 adds input[9] to position input[1], etc.

During each subsequent iteration, half of the active threads drop off and all remaining active threads add an input element whose position is the number of active threads away from its owner position. In our example, during iteration 3, there are two remaining active threads: Thread 0 adds `input[2]` into its owned position `input[0]` and Thread 1 adds `input[3]` into its owned position `input[1]`. Note that if we compare the operation and operand orders of Fig. 10.7 to Fig. 10.6, there is effectively a reordering of the list of input values, rather than simply inserting parentheses in different ways. That is, the list of input values has been effectively reordered as follows:

```
01    __global__ void ConvergentSumReductionKernel(float* input, float* output) {
02        unsigned int i = threadIdx.x;
03        for (unsigned int stride = blockDim.x; stride >= 1; stride /= 2) {
04            if (threadIdx.x < stride) {
05                input[i] += input[i + stride];
06            }
07            __syncthreads();
08        }
09        if(threadIdx.x == 0) {
10            *output = input[0];
11        }
12    }
```

FIGURE 10.8

A kernel with less control divergence and improved execution resource utilization efficiency.

{input[0], input[8], input[1], input[9], ..., input[7], input[15]}

For the result to always remain the same after such reordering, the operation must be commutative in addition to being associative.

Fig. 10.8 shows a kernel with the work assignment illustrated in Fig. 10.7. There are several small but critical differences between this kernel and the one shown in Fig. 10.5. First, the owner position variable i is set to `threadIdx.x` rather than `2*threadIdx.x` (line 02). Thus, the owner positions of all adjacent threads are now adjacent to each other as illustrated in Fig. 10.7. Second, the stride value is initialized as `blockDim.x` and is reduced by half until it reaches 1 (line 03). Third, in each iteration, only the threads whose indices are smaller than the stride value remain active (line 04). This way, all active threads are of consecutive thread indices, as shown in Fig. 10.7.

With these changes, instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other and the section size is always twice the number of remaining active threads. All pairs added during the first round are `blockDim.x` away from each other. After the first iteration, all the pair-wise sums are stored in the first half of the input array, as shown in Fig. 10.7. Finally, the loop divides the stride by 2 before entering the next iteration. Thus, for the second iteration, the stride variable value is half of the `blockDim.x` value. That is, the remaining active threads add elements that are quarter a section away from each other during the second iteration.

The kernel in Fig. 10.8 still has an if-statement (line 04) in the loop. The number of threads that execute an addition operation (line 06) in each iteration is the same as in Fig. 10.5. So, why should there be a difference in control divergence between the two kernels? The answer lies in the positions of threads that perform the addition operation relative to those that do not. Let us consider the example of an input array of 256 elements. During the first iteration, all threads are active so there is no control divergence. During the second iteration, Threads 0 through 63 execute the add statement (active) while threads 64 through 127 do not (inactive). The pair-wise sums are stored in elements 0 through 63 during the second iteration. Since the warps consist of 32 threads with consecutive `threadIdx.x` values, all threads in warps 0 through

warp 1 execute the add statement whereas all threads in warp 2 through warp 3 become inactive. Since all threads in each warp take the same path of execution, there is no control divergence!

However, the kernel in Fig. 10.8 does not completely eliminate the divergence caused by the if-statement. The reader should verify that for the 256-element example, starting with the 4th iteration, the number of threads that execute the addition operation will fall below 32. That is, the final five iterations will have only 16, 8, 4, 2, and 1 thread(s) performing the addition. This means that the kernel execution will still have divergence in these iterations. However, the number of iterations of the loop that has divergence is reduced from ten to five. We can calculate the total number of execution resources consumed as:

$$(N/64 * 1 + N/64 * \tfrac{1}{2} + \cdots + 1 + 5 * 1) * 32$$

The part in parentheses reflects the fact that in each subsequent iteration, half of the warps become entirely inactive and no longer consume execution resources. This series continues until there is only one full warp of active threads. The last term (5*1) reflects the fact that for the final five iterations, there is only one active warp and all its 32 threads consume execution resources even though only a fraction of the threads are active. Thus, the sum in the parentheses gives the total number of warp executions through all iterations, which when multiplied by 32 gives the total amount of execution resources consumed.

For our 256-element example, the execution resources consumed are $(4 + 2 + 1 + 5 * 1) * 32 = 384$, which is almost half of 736, the resources consumed by the kernel in Fig. 10.5. Since the number of active threads in each iteration did not change from Fig. 10.6 to Fig. 10.7, the efficiency of the new kernel in Fig. 10.8 is $255/384 = 66\%$, which is almost double the efficiency of the kernel in Fig. 10.5. Note also that since the warps are scheduled to take turns to execute in an SM of limited execution resources, the total execution time will also improve with the reduced resource consumption.

The difference between the kernels in Fig. 10.5 and Fig. 10.8 is small but can have a significant performance impact. It requires someone with clear understanding of the execution of threads on the SIMD hardware of the device to be able to confidently make such adjustments.

10.5 Reducing memory access divergence

The simple kernel in Fig. 10.5 has another performance issue: memory access divergence. As we explained in Chapter 5, it is important to achieve memory coalescing within each warp. That is, adjacent threads in a warp should access adjacent locations when they access global memory. Unfortunately, as illustrated in Fig. 10.6, adjacent threads do not access adjacent locations. In each iteration, each thread performs two global memory reads and one global memory write. The first read is from its owned location, the second read is from the location that is of stride distance away from its

owned location, and the write is to its owned location. Since the locations owned by adjacent threads are not adjacent locations, the accesses made by adjacent threads will not be fully coalesced. During each iteration, the memory locations collectively accessed by a warp are of one stride distance away from each other.

For example, as shown in Fig. 10.6, when all threads in a warp perform their first read during the first iteration, the locations are two elements away from each other. As a result, twice the number of global memory transactions as that for a coalesced access are triggered, and half the data returned will not be used by the threads. The same behavior also occurs for the second read and the write. During the second iteration, every other thread drops out, and the locations collectively accessed by the warp are four elements away from each other. Four times the number of global memory transactions as that for a coalesced access are performed and only one fourth of the data returned will be used by the threads. This pattern continues and the inefficiency of the accesses increases as the stride increases.

This will continue until there is only one active thread for each warp that remains to be active. Only in the case when there is one active thread in the warp, the warp will perform one global memory request. Thus, the total number of global memory requests is:

$$(N/64 * 5 * 2 + N/64 * 1 + N/64 * \tfrac{1}{2} + N/64 * \tfrac{1}{4} + \cdots + 1) * 3$$

The first term $(N/64 * 5 * 2)$ corresponds to the first five iterations where all N/64 warps have two or more active threads whose divergent memory accesses trigger two global memory requests, and therefore, each warp triggers two global memory requests. The remaining terms account for the final iterations where each warp has only one active thread and triggers only one global memory request, and half of the warps drop out in each subsequent iteration. The multiplication by 3 accounts for the two reads and one write by each active thread during each iteration. In the 256-element example, the total number of global memory requests performed by the kernel is $(4 * 5 * 2 + 4 + 2 + 1) * 3 = 141$.

For the kernel in Fig. 10.8, the adjacent threads in each warp always access adjacent locations in the global memory so the accesses are always coalesced. As a result, each warp only triggers a minimal number of global memory transactions on any read or write and fully utilizes the returned data. As the iterations progress, entire warps drop out so no global memory access would be performed by any thread in these inactive warps. The active warps still perform coalesced accesses, triggering the minimal number of global memory transactions and fully utilizing the returned data. Half of the warps drop out in each iteration until there is only one warp for the final five iterations. The only period of resource under-utilization occurs when the amount of data requested drops below the size of a single global memory transaction. Since DRAM bandwidth is a limited resource, the execution time of the kernel in Fig. 10.7 is likely to be significantly better than that for the simple kernel in Fig. 10.5.

In conclusion, the kernel in Fig. 10.7 offers more efficiency in using both execution resources and DRAM bandwidth compared to the kernel in Fig. 10.5. The advantage comes from both reduced control divergence and improved memory coalescing.

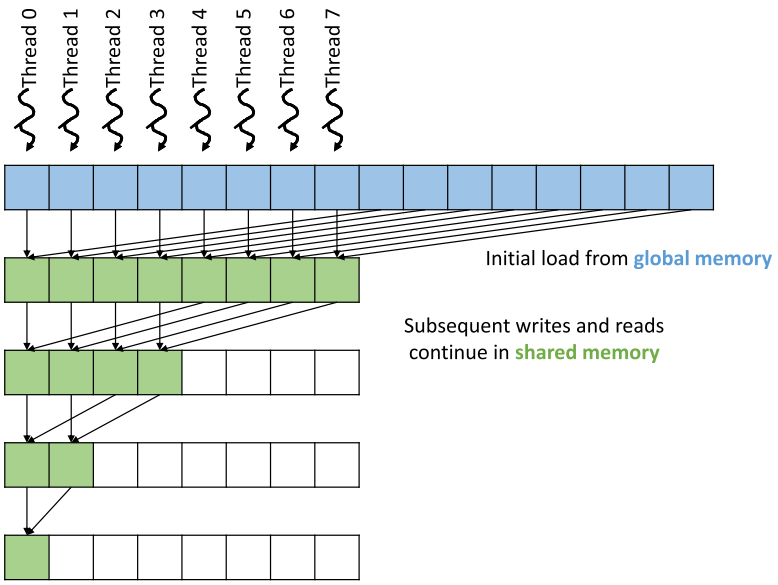


FIGURE 10.9

Using shared memory to reduce accesses to the global memory.

10.6 Reducing global memory accesses

The non-divergent kernel in Fig. 10.8 can be further improved by using shared memory. Note that in each iteration, threads write their partial sum result values out to the global memory and these values are reread by the same threads and other threads in the next iteration. Although some of the reads and writes may be served by the last-level cache, there are still significant latency and bandwidth limitations when accessing the last-level cache. Since the shared memory has much lower latency and higher bandwidth than the last-level cache and the global memory, we can further improve the execution speed by keeping the partial sum results in the shared memory. This idea is illustrated in Fig. 10.9.

The strategy for using the shared memory is implemented in the kernel shown in Fig. 10.10. The idea is to use each thread to load and add two of the original elements before writing the partial sum into the shared memory (line 04). Since the first iteration is already done when accessing the global memory locations outside the loop, the `for`-loop starts with `blockDim.x/2` (line 04) instead of `blockDim.x`. The `__syncthreads()` is moved to the beginning of the loop to ensure that we synchronize between the shared memory accesses and the first iteration of the loop. The threads proceed with the remaining iterations by reading and writing the shared memory (line 08). Finally, at the end of the kernel, Thread 0 writes the sum into output, maintaining the same behavior as the previous kernels (lines 11-13).

Using the kernel in Fig. 10.10, the number of global memory accesses are reduced to the initial loading of the original contents of the input array and the final write to

```
01    __global__ void SharedMemorySumReductionKernel(float* input) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int t = threadIdx.x;
04        input_s[t] = input[t] + input[t + BLOCK_DIM];
05        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2) {
06            __syncthreads();
07            if (threadIdx.x < stride) {
08                input_s[t] += input_s[t + stride];
09            }
10        }
11        if (threadIdx.x == 0) {
12            *output = input_s[0];
13        }
14    }
```

FIGURE 10.10

A kernel that uses shared memory to reduce global memory accesses.

input[0]. Thus for an N-element reduction, the number of global memory accesses is just N + 1. Note also that both global memory reads in Fig. 10.10 (line 04) are coalesced. So with coalescing, there will only be (N/32) + 1 global memory requests. For the 256-element example, the total number of global memory requests triggered will be reduced from 36 for the kernel in Fig. 10.8 to 8 + 1 = 9 for the shared memory kernel in Fig. 10.9, a 4× improvement. Another added benefit of using shared memory, besides reducing the number of global memory accesses, is that the input array is not modified. This property is useful if the original values of the array are needed for some other computation in another part of the program.

10.7 Reducing synchronization overhead with warp-level primitives

In the kernel in Fig. 10.10, when the stride reaches 32 and onwards, only one warp in the thread block is active while executing the reduction tree. CUDA provides multiple *warp-level primitives* that allow threads in the same warp to share data and synchronize with each other in ways that threads across different warps cannot. Accordingly, we can further optimize the reduction kernel when the number of active threads reaches 32 and onwards by using warp-level primitives to reduce the overhead of accessing shared memory and calling `__syncthreads()`. In this section, we introduce some of the warp-level primitives available in CUDA and show how they can be used to further optimize the reduction kernel.

The simplest warp-level primitive is `__syncwarp()` which behaves like `__syncthreads()`, but at the warp level. `__syncwarp()` ensures that all threads in the warp reach the point in the code where it is called before any of them proceed with their execution. In particular, if the threads in the warp have diverged, `__syncwarp()` ensures that they reconverge.

Another category of warp-level primitives, which are relevant for optimizing the reduction kernel in this section, is warp shuffle functions. *Warp shuffle functions*

```
T __shfl_down_sync(unsigned mask, T var, unsigned int delta);
```

- `mask` indicates which threads in the warp are to be active when executing the shuffle instruction
- `var` is the value passed by the sending thread to be shuffled to the receiving thread
- `delta` is the stride by which the sending thread's index is higher than the receiving thread's index
- The return value for the receiving thread is the value shuffled from the sending thread. For thread *i*, the return value is the value of `var` for thread *i* + `delta` assuming that both thread *i* and thread *i* + `delta` have their bits set in `mask`.

FIGURE 10.11

The `__shfl_down_sync` warp-level primitive.

allow threads in the same warp to exchange data with each other (or *shuffle* data) directly from their registers without having to access shared memory or call `__syncthreads()`. Of particular relevance to the reduction kernel is the `__shfl_down_sync` primitive which allows a thread to shuffle data from another thread whose index is higher by a given stride. The specific parameters to `__shfl_down_sync` are shown and explained in Fig. 10.11.

The `__shfl_down_sync` primitive can be used to implement a warp-wide reduction without the need to access shared memory or call `__syncthreads()`. Fig. 10.12 illustrates how such a warp-wide reduction can be integrated into the block-wide reduction to reduce the barrier synchronization overhead and access to shared memory. For simplicity, the illustration shows a block consisting of four warps with four threads each. In practice, a block with 1024 threads would consist of 32 warps with 32 threads each.

In Fig. 10.12, the first few iterations are executed like the previous implementation in Fig. 10.9. Each thread loads two elements from shared memory, adds them, and stores the result in shared memory. A block-wide barrier synchronization using `__syncthreads()` separates each iteration. After each iteration completes, half the threads drop out of the computation. However, unlike the previous implementation in Fig. 10.9, once the threads remaining comprise a single warp, there is no longer a need to use `__syncthreads()` and shared memory. Instead, the threads in the warp load the values from shared memory to their registers, and execute the remaining reduction tree iterations by shuffling values across their registers.

We now turn our attention to the code for implementing the approach in Fig. 10.12. To assist with warp-level programming, we define two preliminary device functions for identifying the index of a warp in a block and the index of a thread in a warp. To identify the index of a warp in a block we define the following device function:

```
__device__ unsigned int warpIdx() { return threadIdx.x/WARP_SIZE; }
```

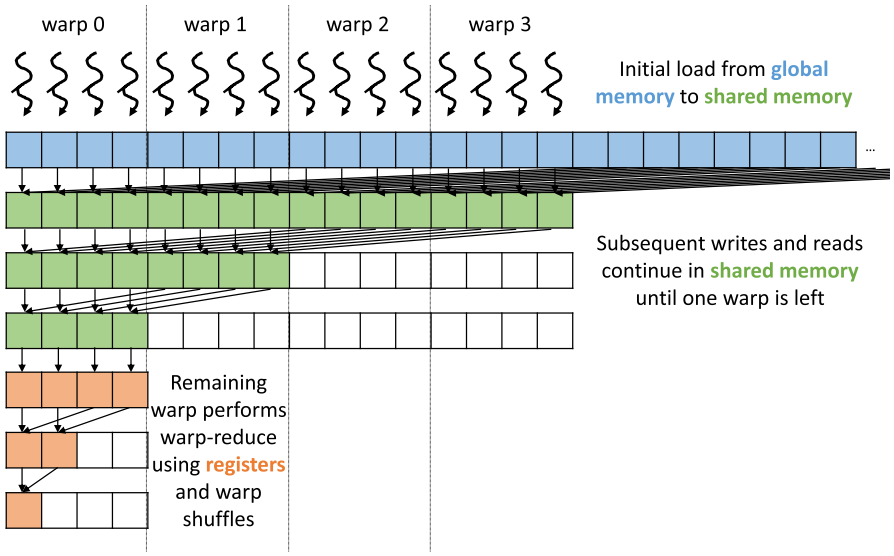


FIGURE 10.12

Using warp-wide reduction in the final stage to reduce synchronization overhead and accesses to shared memory.

This function divides the thread index by the warp size of 32 such that every group of 32 consecutive threads have the same warp index. For example, threads 0 to 31 all have a warp index of 0, threads 32 to 63 all have a warp index of 1, etc. To identify the index of a thread in a warp, also known as the *lane index*, we define the following device function:

```
__device__ unsigned int laneIdx() { return threadIdx.x%WARP_SIZE; }
```

This function takes the modulo of the thread index with respect to the warp size of 32 such that 32 consecutive threads in the same warp have lane indices ranging from 0 to 31. For example, thread 32 in the block has a lane index of $32\%32 = 0$ because it is the first thread in warp 1, thread 33 in the block has a lane index of $33\%32 = 1$ because it is the second thread in warp 1, etc.

With these helper device functions defined, we now turn our attention to implementing a warp-wide reduction. Fig. 10.13 shows a device function `warp_reduce` for performing a warp-wide reduction operation by shuffling values across registers. The function is intended to be called by all threads in the warp, with each thread providing a value `val` as a parameter that it would like to contribute to the reduction. Each thread starts by initializing its partial sum variable `partialSum` to `val` (line 02). A reduction loop is then executed starting from stride 16 down to stride one (line 03). The first stride is 16 because a warp has 32 threads, hence 32 values to reduce, so in the first iteration, the first 16 threads will accumulate values coming from the next 16 threads.

```
01    __device__ __inline__ float warp_reduce(float val) {
02        float partialSum = val;
03        for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
04            partialSum += __shfl_down_sync(0xffffffff, partialSum, stride);
05        }
06        return partialSum;
07    }
```

FIGURE 10.13

A device function that uses warp-level primitives to perform a warp-wide reduction.

```
01    __global__ void WarpLevelSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int t = threadIdx.x;
04        input_s[t] = input[t] + input[t + BLOCK_DIM];
05        for (unsigned int stride = blockDim.x/2; stride >= WARP_SIZE; stride /= 2) {
06            __syncthreads();
07            if (threadIdx.x < stride) {
08                input_s[t] += input_s[t + stride];
09            }
10        }
11        __syncthreads();
12        if(warpIdx() == 0) {
13            float partialSum = input_s[t];
14            partialSum = warp_reduce(partialSum);
15            if (threadIdx.x == 0) {
16                *output = partialSum;
17            }
18        }
19    }
```

FIGURE 10.14

A reduction kernel that uses warp-wide reduction in the final stage to reduce synchronization overhead and accesses to shared memory.

In each iteration of the reduction loop, we use `__shfl_down_sync` to exchange data and synchronize within the warp in a lightweight manner (line 04). The first argument, `0xffffffff`, is a 32-bit mask where all the bits are set, indicating that all threads in the warp participate in this reduction operation. The second argument indicates that each sending thread will pass the value in its register containing `partialSum` to the receiving thread. The third argument indicates that the receiving thread's index is `stride` below the sending thread's index. The value returned by `__shfl_down_sync` is the value received by the receiving thread from the sending thread. In other words, for thread i in the warp, `__shfl_down_sync` will return the value of `partialSum` for thread $i+$`stride`. This value is then added to the receiving thread's partial sum. After the loop finishes executing, the final result is found in the `partialSum` variable of the first thread in the warp. Hence, this value is returned to the caller (line 06) with the understanding that only the first thread's return value represents the true result.

Having implemented a warp-wide reduction using warp-level primitives, we can now use this device function to optimize the reduction kernel. Fig. 10.14 shows the updated reduction kernel that uses warp-wide reduction. The first part of the code (lines 01-10) is similar to that in Fig. 10.10, with the exception that the loop on line

05 ends when the stride reaches the warp size, which is 32. Starting from stride 32 down to stride one, only the first warp in the thread block participates in executing the reduction tree. For this reason, only the threads whose warp index evaluates to 0 continue to execute (line 12). Each thread in the remaining warp begins by loading the element corresponding to its own index from shared memory to a register called `partialSum` (line 13). Next, the threads of the warp jointly call the `warp_reduce` function from Fig. 10.13 (line 14), which returns the final sum to the first thread in the warp, i.e., thread 0 of the block. This thread then writes the result to the output in global memory (lines 17-19). In this way, we have eliminated the shared memory accesses and barrier synchronization from the iterations with stride 16 onwards.

The `__shfl_down_sync` function, as well as other warp-level primitives, takes an additional optional parameter that specifies the width of the group of threads involved in executing the primitive. By default, this width is 32, which means that all 32 threads in the warp are treated as a single group. However, it is possible to specify a smaller width that breaks the warp into smaller collaborating groups. For example, if a width of eight is specified, then the warp is treated as four groups of eight threads, and each eight threads of a group execute the primitive in collaboration with each other and independent of the other threads in the warp.

Other types of warp shuffle functions are also available in CUDA, and are useful in other contexts where exchanging data between different threads in the same warp is needed. These other functions include `__shfl_sync`, `__shfl_up_sync`, and `__shfl_xor_sync`. We refer the reader to the CUDA C++ Programming Guide for details about how the other warp shuffle functions operate.

In addition to warp shuffle functions, other types of warp-level primitives include *warp voting functions* that check which threads in a warp meet some condition, *warp match functions* which check which threads in a warp have the same value for some parameter, and *warp reduce functions* which perform warp-wide reductions for integer values. Chapter 12 presents an example of how warp voting functions can be used.

10.8 Further reducing synchronization overhead with two-stage warp-wide reduction

The kernel in Fig. 10.14 eliminates the overhead of shared memory accesses and barrier synchronization in the last few iterations of the reduction tree. However, the initial iterations still incur this overhead. One way to eliminate this overhead is to use warp-wide reductions in the initial stages as well. In this case, we can have each warp in the thread block perform a warp-wide reduction on a segment of the input and place its partial sum in shared memory, then have a single warp perform a warp-wide reduction to reduce the partial sums.

Fig. 10.15 illustrates how two stages of warp-wide reduction can be used to further reduce the barrier synchronization overhead and access to shared memory. Initially, each warp loads the elements its threads are responsible for reducing and performs a

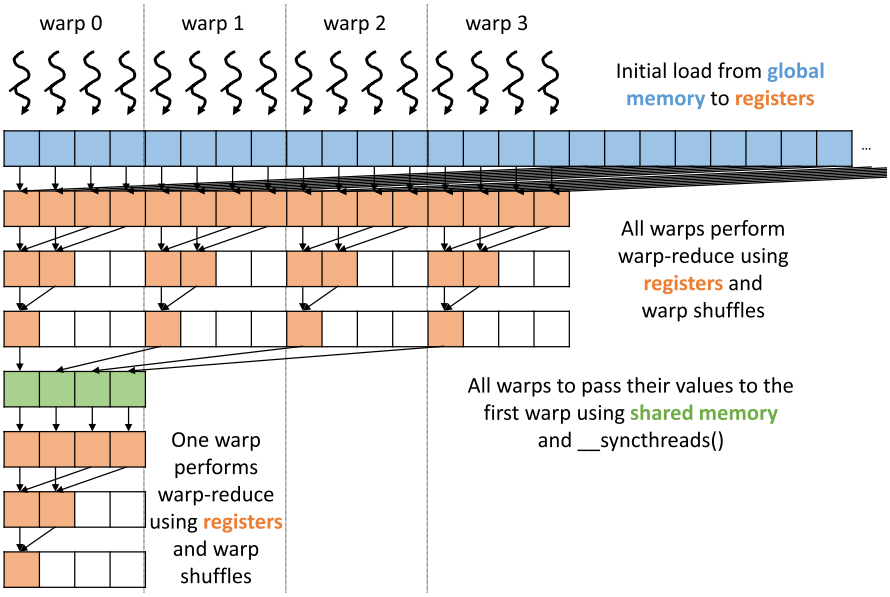


FIGURE 10.15

Using two-stage warp-wide reduction to further reduce synchronization overhead and accesses to shared memory.

warp-wide reduction. At the end of the first stage of warp-wide reductions, the first thread in each warp has the total sum of all elements assigned to that warp. These threads write the sums to shared memory and a block-wide barrier synchronization is performed using `__syncthreads()`.[4] After that, the first warp picks up these sums and reduces them into a total sum using another warp-wide reduction.

Fig. 10.16 shows a reduction kernel that performs such a block-wide reduction using two stages of warp-wide reductions. First, each thread loads the two input elements assigned to it, adds them, and places the result in a register (line 04). Next, the threads in the same warp jointly call the `warp_reduce` function to perform a warp-wide reduction (line 05). All warps in the thread block are active throughout this process, each performing an independent reduction. Next, the first thread in each warp, i.e., the thread with lane index 0 which has the warp's partial sum, places the partial sum in shared memory at a location associated with that warp (lines 07-10). A barrier synchronization is then performed to ensure that all warps have written their partial sums (line 11).

[4] Keep in mind that we still assume that only one thread block is used to reduce up to 2048 elements. This is why all warps can exchange data through shared memory. This assumption will be removed in the next section.

```
01    __global__ void TwoStageWarpLevelSumReductionKernel(float* input, float* output) {
02
03        unsigned int t = threadIdx.x;
04        float partialSum = input[t] + input[t + BLOCK_DIM];
05        partialSum = warp_reduce(partialSum);
06
07        __shared__ float partialSums_s[BLOCK_DIM/WARP_SIZE];
08        if(laneIdx() == 0) {
09            partialSums_s[warpIdx()] = partialSum;
10        }
11        __syncthreads();
12
13        if(warpIdx() == 0) {
14            float partialSum = partialSums_s[t];
15            partialSum = warp_reduce(partialSum);
16            if (threadIdx.x == 0) {
17                *output = partialSum;
18            }
19        }
20    }
```

FIGURE 10.16

A reduction kernel that uses two-stage warp-wide reduction to further reduce synchronization overhead and accesses to shared memory.

Once all warps have written their partial sums, the code proceeds similar to the kernel in Fig. 10.14. Only the threads in the first warp continue execution (line 13).[5] Each thread in the warp loads one of the partial sums from shared memory (line 14), then the threads perform a warp-wide reduction (line 15). Finally, the first thread in the warp writes out the final sum to global memory (lines 16-18).

The main advantage of the implementation in Fig. 10.16 that uses two stages of warp-wide reductions is that it eliminates the overhead of shared memory accesses and barrier synchronization from the first stage as well. The only time we need to access shared memory and perform barrier synchronization is when transitioning between the two stages. However, this implementation also has a disadvantage compared to the one in Fig. 10.14. Recall that in the first stage of the implementation in Fig. 10.14, half the warps become inactive after each iteration and there is no control divergence. However, in Fig. 10.16 that uses two stages, all the warps are active throughout the entire first stage. However, within each warp, not all threads perform useful work in every shuffle operation, which means that this implementation suffers from more control divergence. Hence, this implementation trades off more control divergence to reduce shared memory accesses and barrier synchronization.

In practice, this trade-off is worthwhile because once the data is loaded from global memory, the performance of the rest of the reduction kernel is primarily limited by latency. Eliminating the shared memory accesses and barrier synchronizations helps reduce the long-latency instructions in the kernel. On the other hand, eliminating divergent warps, while effective, does not substantially benefit a latency bound

[5] Since there are up to 1024 threads in a block and 32 threads in a warp, there will be only up to 32 warps in the block and thus we up to 32 partial sums from the first stage. Therefore, we will need only one warp to complete the second stage.

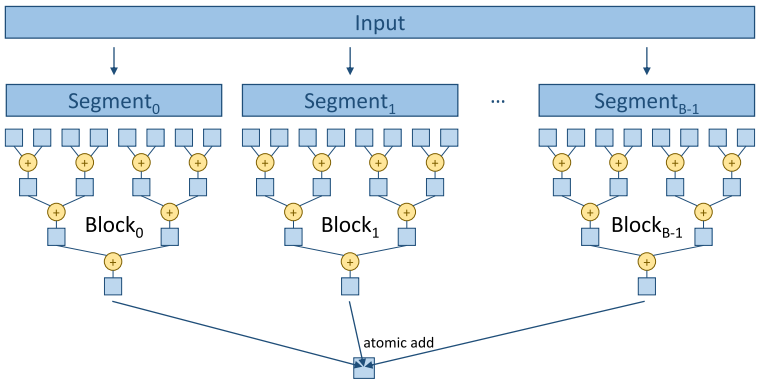


FIGURE 10.17

Multi-block reduction using atomic operations.

kernel because the execution cycles saved by eliminating divergence will likely be replaced by stall cycles.

10.9 Reduction for arbitrary length inputs

All the kernels that we have studied so far assume that they will be launched with one thread block. The main reason for this assumption is that `__syncthreads()` is used as a barrier synchronization among all the active threads. Recall that `__syncthreads()` can be used only among threads in the same block. This limits the level of parallelism to 1024 threads on current hardware. For large input arrays that contain millions or even billions of elements, we can benefit from launching more threads to further accelerate the reduction process. Since we do not have a good way to perform barrier synchronization among threads in different blocks, we will need to allow threads in different blocks to execute independently.

Fig. 10.17 illustrates the concept of multi-block reduction using atomic operations. The idea is to partition the input array into segments so that each segment is of appropriate size for a block. All blocks then independently execute a reduction tree and accumulate their results to the final output using an atomic add operation.

Fig. 10.18 shows the kernel implementation for the multi-block reduction, which has only a few changes compared to the kernel in Fig. 10.16. The partitioning is done by assigning a different value to the segment variable according to the thread's block index (line 03). The size of each segment is `2*blockDim.x`. That is, each block processes `2*blockDim.x` elements. Thus, when we multiply the size of each segment by `blockIdx.x` of a block, we have the starting location of the segment to be processed by the block. For example, if we have 1024 threads in a block, the segment size would be 2*1024 = 2048. The starting locations of segments would be 0 for block 0, 2048 (2048 * 1) for block 1, 4096 (2048 * 2) for block 2, etc.

```
01    __global__ void reduce_kernel(float* input, float* output, unsigned int N) {
02
03        unsigned int segment = 2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        float partialSum = input[i] + input[i + BLOCK_DIM];
06        partialSum = warp_reduce(partialSum);
07
08        __shared__ float partialSums_s[BLOCK_DIM/WARP_SIZE];
09        if(threadIdx.x%WARP_SIZE == 0) {
10            partialSums_s[threadIdx.x/WARP_SIZE] = partialSum;
11        }
12        __syncthreads();
13
14        if(threadIdx.x < WARP_SIZE) {
15            float partialSum = partialSums_s[threadIdx.x];
16            partialSum = warp_reduce(partialSum);
17            if(threadIdx.x == 0) {
18                cuda::atomic_ref<float, cuda::thread_scope_device>
19                    output_ref(*output);
20                output_ref.fetch_add(partialSum, cuda::memory_order_relaxed);
21            }
22        }
23
24    }
```

FIGURE 10.18

A multi-block sum reduction kernel using atomic operations.

Once we know the starting location for each block, all threads in a block can simply work on the assigned segment as if it is the entire input data. Within a block, we assign the owned location to each thread by adding the `threadIdx.x` to the segment starting location for the block that the thread belongs to (line 04). The local variable `i` holds the owned location of the thread in the global input array, in place of `t` in the previous kernels. Line 05 is adapted to use `i` instead of `t` when accessing the global input array.

The rest of the code that performs the main reduction computation remains as is. One can adapt any of the single-block reduction kernel code into this part of the kernel. The final difference is that instead of having Thread 0 (leader thread) of each thread block writing the partial sum to the global output value, it accumulates the partial sum to the output using an atomic addition operation (line 18-20). The atomic operation ensures that the leader threads of all thread blocks accumulate their partial sums to the global value without any race conditions.

A subtle but important aspect of using the atomic operation to accumulate partial sums into the final sum is that the thread blocks can make their contributions in any arbitrary order with respect to each other. This is because atomic operations only enforce mutual exclusion but not ordering. Therefore, the operator for the reduction must be both commutative and associative in order to benefit from the use of atomic operations to generalize to inputs of arbitrary length.

As an alternative to using atomic operations to produce a global reduction from the reduction results of segments, one can have the leader threads of thread blocks to write into a partial sum array using their block index. A new kernel can then be launched with one thread block to produce the final output value. Another alternative

is to copy the partial sum array to the host memory and use the CPU to complete the reduction. In systems where atomic operations are not available or are too expensive, using a single-block grid or the host to complete the reduction would make sense. These approaches also give the programmer more control over the order in which the partial sums are accumulated.

10.10 Thread coarsening to reduce overhead

The reduction kernels that we have worked with so far all try to maximize parallelism by using as many threads as possible. That is, for a reduction of N elements, N/2 threads are launched. With a thread block size of 1024 threads, the resulting number of thread blocks is N/2048. However, in processors with limited execution resources, the hardware may have only enough resources to execute a portion of the thread blocks in parallel. In this case, the hardware will serialize the surplus thread blocks, executing a new thread block whenever an old one has completed.

To parallelize reduction, we have actually paid a heavy price to distribute the work across multiple threads and thread blocks. As we have seen in earlier sections, hardware underutilization increases with each successive stage of the reduction tree because of more threads become idle. The phase where the hardware is underutilized occurs for every thread block we launch. It is an inevitable price to pay if the thread blocks were to actually run in parallel. However, if the hardware were to serialize these thread blocks, we are better off serializing them ourselves in a more efficient manner. As we discussed in Chapter 6, thread granularity coarsening, or thread coarsening for brevity, is a category of optimizations that serializes some of the work into fewer threads to reduce parallelization overhead. We start by showing an implementation of parallel reduction with thread coarsening applied by assigning more elements to each thread block. We then further elaborate on how this implementation reduces hardware underutilization.

Fig. 10.19 illustrates how thread coarsening can be applied to the example in Fig. 10.9. In Fig. 10.9, each thread block received 16 elements which is two elements per thread. Each thread independently adds the two elements it is responsible for while loading them from global memory, then the threads collaborate to execute a reduction tree. In Fig. 10.19, we coarsen the thread block by a factor of two. Hence, each thread block receives twice the number of elements, i.e., 32 elements which is 4 elements per thread. In this case, each thread independently adds four elements while loading them from global memory before the threads collaborate to execute a reduction tree. The three steps to add the four elements are illustrated by the first three rows of arrows in Fig. 10.19. Note that all threads are active and performing useful work during these three steps. Moreover, since the threads add the four elements that they are responsible for independently, they do not need to synchronize. The remaining steps performing the reduction tree are the same as before.

Fig. 10.20 shows the kernel code for implementing reduction with thread coarsening for the multi-block kernel. Compared to Fig. 10.18, the kernel has two main

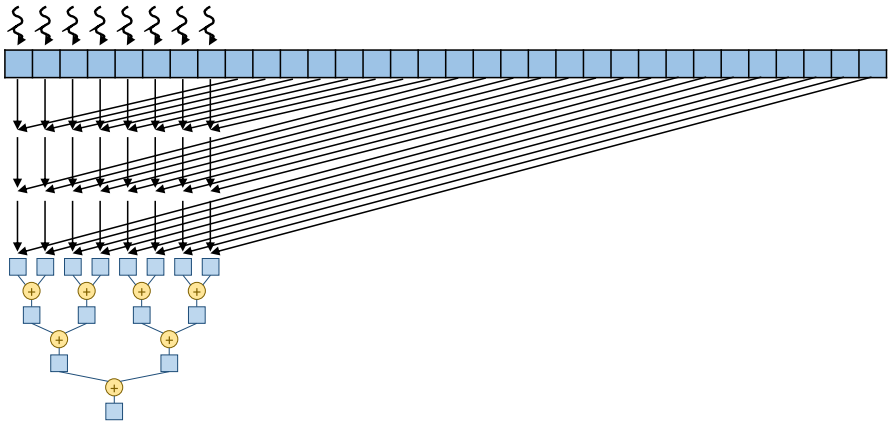


FIGURE 10.19

Thread coarsening in reduction.

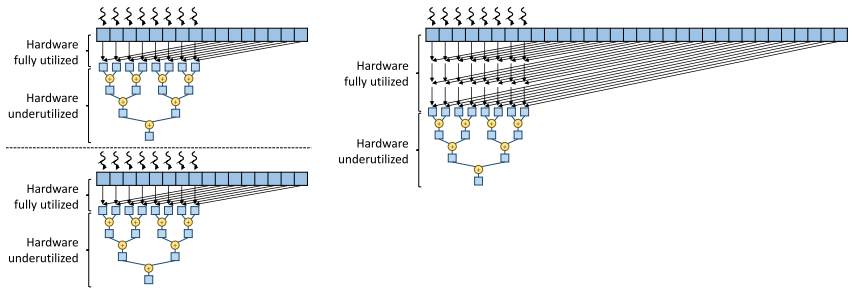
```
01    __global__ CoarsenedSumReductionKernel(float* input, float* output) {
02
03        unsigned int segment = COARSE_FACTOR*2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        float partialSum = input[i];
06        for(unsigned int c = 1; c < COARSE_FACTOR*2; ++c) {
07            partialSum += input[i + c*BLOCK_DIM];
08        }
09        ...
10
11    }
```

FIGURE 10.20

Sum reduction kernel with thread coarsening.

differences. The first difference is that, when the beginning of the block's segment is identified, we multiply by `COARSE_FACTOR` to reflect the fact that the size of the block's segment is `COARSE_FACTOR` times larger (line 03). The second difference is that, when adding the elements that the thread is responsible for, rather than just adding two elements (line 05 in Fig. 10.18), we use a coarsening loop to iterate over the elements and add them based on `COARSE_FACTOR` (lines 06-08 in Fig. 10.20). Note that all threads are active throughout this coarsening loop, the partial sum is accumulated to the local variable `partialSum`, and no calls to `__syncthreads()` are made in the loop because the threads act independently.

Fig. 10.21 compares the execution of two original thread blocks without coarsening serialized by the hardware, shown in Fig. 10.21(a), with one coarsened thread block performing the work of two thread blocks, shown in Fig. 10.21(b). In Fig. 10.21(a), the first thread block performs one step where each thread adds the two elements that it is responsible for. All threads are active during this step, so the hardware is fully utilized. The remaining three steps execute the reduction tree where



(a) Execution of two original thread blocks serialized by the hardware

(b) Execution of one coarsened thread block doing the work of two original thread blocks

FIGURE 10.21

Comparing parallel reduction with and without thread coarsening.

half the threads drop out each step, underutilizing the hardware. Moreover, each step requires a barrier synchronization as well as accesses to shared memory. When the first thread block is done, the hardware then schedules the second thread block which follows the same steps but on a different segment of the data. Overall, the two blocks collectively take a total of eight steps, out of which two steps fully utilize the hardware whereas six steps underutilize the hardware and require barrier synchronization and shared memory access.

In contrast, in Fig. 10.21(b), the same amount of data is processed by only a single thread block that is coarsened by a factor of two. This thread block initially takes three steps where each thread adds the four elements it is responsible for. All threads are active during all three steps so the hardware is fully utilized, and there are no barrier synchronizations or accesses to shared memory performed. The remaining three steps execute the reduction tree where half the threads drop out each step, underutilizing the hardware, and where barrier synchronization and accesses to shared memory are needed. Overall, only six steps are needed (instead of eight), out of which three steps (instead of two) fully utilize the hardware whereas three steps (instead of six) underutilize the hardware and require barrier synchronization and shared memory access. Therefore, thread coarsening effectively reduces the overhead from hardware underutilization, synchronization, and access to shared memory.

Theoretically, we can increase the coarsening factor well beyond two. However, one must keep in mind that as we coarsen threads, less work will be done in parallel. Therefore, increasing the coarsening factor will reduce the amount of data parallelism being exploited by the hardware. If we increase the coarsening factor too much such that we launch fewer thread blocks than the hardware is capable of executing, we will no longer be able to take full advantage of the available parallel hardware execution resources. The best coarsening factor ensures that there are enough thread blocks to fully utilize the hardware, which usually depends on the total size of the input as well as the characteristics of the specific device.

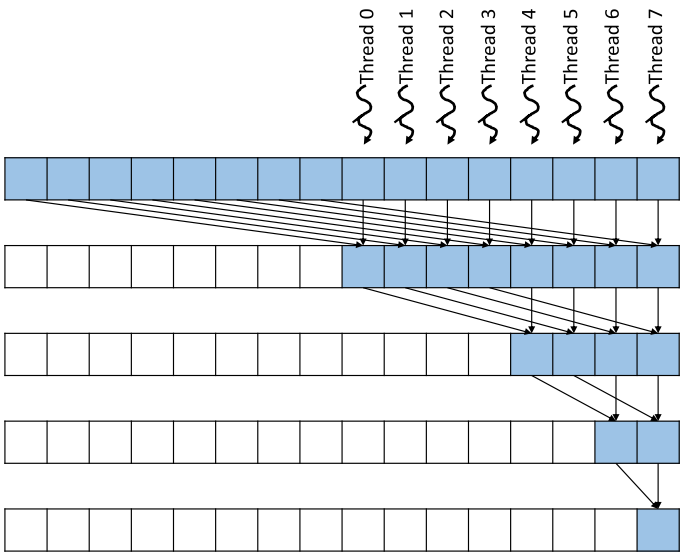
10.11 Summary

The parallel reduction pattern is important as it plays a key role in many data processing applications. Although the sequential code is simple, it should be clear to the reader that several techniques are needed to achieve high performance parallel execution. These techniques include thread index assignment for reduced divergence, using shared memory for reduced global memory accesses, using warp-level primitives for reduced barrier synchronizations and shared memory access, multi-block reduction with atomic operations, and thread coarsening. Reduction is also an important foundation for the prefix-sum pattern that is an important algorithm component for parallelizing many applications and will be the topic for the next chapter.

We reiterate that in practice, programmers do not need to implement parallel reduction from scratch because CUDA libraries such as Thrust [1,2] and CUB [3] provide built-in implementations that are highly optimized for GPUs. However, parallel reduction serves as an excellent vehicle for introducing parallelization and optimization techniques that are applicable in many application contexts.

10.12 Exercises

1. For the simple reduction kernel in Fig. 10.5, if the number of elements is 1024 and the warp size is 32, how many warps in the block will have divergence during the 5th iteration?
2. For the improved reduction kernel in Fig. 10.8, if the number of elements is 1024 and the warp size is 32, how many warps will have divergence during the 5th iteration?
3. Modify the kernel in Fig. 10.8 to use the access pattern illustrated below.



4. Modify the kernel in Fig. 10.20 to perform a max reduction instead of a sum reduction.
5. Modify the kernel in Fig. 10.20 to work for an arbitrary length input that is not necessarily a multiple of `COARSE_FACTOR*2*blockDim.x`. Add an extra parameter `N` to the kernel which represents the length of the input.
6. Assume parallel reduction is to be applied on the following input array:

   {6, 2, 7, 4, 5, 8, 3, 1}

   Show how the contents of the array change after each iteration if:
   a. The unoptimized kernel in Fig. 10.5 is used.
   b. The kernel optimized for coalescing and divergence in Fig. 10.8 is used.

References

[1] N. Bell, J. Hoberock, Thrust: a productivity-oriented library for CUDA, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.
[2] NVIDIA, Thrust: the C++ parallel algorithms library, https://nvidia.github.io/cccl/thrust/index.html, 2025.
[3] NVIDIA, CUB, https://nvidia.github.io/cccl/cub/index.html, 2025.

CHAPTER

11 Scan

And work efficiency in parallel algorithms

With special contributions from Li-Wen Chang, Juan Gómez-Luna, and John Owens

CONTENTS

11.1 Background 252
11.2 Parallel scan with the Kogge-Stone algorithm 254
11.3 Double-buffering to reduce synchronization 258
11.4 Warp-level primitives to reduce synchronization 261
11.5 Work efficiency considerations 265
11.6 Coarsening to improve work-efficiency 267
11.7 Register tiling to avoid shared memory access latency 270
11.8 Memory bandwidth considerations 272
11.9 Consolidating block segments for a global scan 273
11.10 Parallel scan with the Brent-Kung algorithm 282
11.11 Summary 285
11.12 Exercises 286
References 286

Parallel scan is frequently used to parallelize seemingly sequential operations, such as resource allocation, work assignment, and polynomial evaluation. In general, if a computation is naturally described as a mathematical recursion where each item in a series is defined in terms of the previous item, it is likely parallelizable as a parallel scan operation. Parallel scan plays a key role in massively parallel computing for a simple reason: any sequential section of an application can drastically limit the overall performance of the application. Many such sequential sections can be converted into parallel computation with parallel scan. For this reason, parallel scan is often used as a primitive operation in parallel algorithms that perform filtering, radix sort, quick sort, string comparison, polynomial evaluation, solving recurrences, and tree operations. The filtering example is presented in Chapter 12 and the radix sort example is presented in Chapter 14.

Multiple algorithms exist for parallel scan with different tradeoffs between work complexity and span. Some parallel algorithms can have higher work complexity than a sequential algorithm. As we shall discuss in this chapter, an increase in work complexity can make a parallel scan run slower than a sequential scan when hardware resources are constrained, necessitating optimizations that mitigate the higher work

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00021-4
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

complexity. We also use parallel scan as an example to introduce new techniques such as double-buffering and unidirectional synchronization, as well as showcase other optimizations that we have already seen such as using warp-level primitives, thread coarsening, and register tiling.

11.1 Background

Mathematically, an *inclusive scan* operation takes a binary operator $\oplus$, and an input array of N elements $[x_0, x_1, \cdots, x_{N-1}]$, and returns the output array:

$$[x_0, (x_0 \oplus x_1), \cdots, (x_0 \oplus x_1 \oplus \cdots \oplus x_{N-1})].$$

For example, if $\oplus$ is addition, an inclusive scan operation on the input array [3, 1, 7, 0, 4, 1, 6, 3], would return $[3, 3 + 1, 3 + 1 + 7, 3 + 1 + 7 + 0, \cdots, 3 + 1 + 7 + 0 + 4 + 1 + 6 + 3] = [3, 4, 11, 11, 15, 16, 22, 25]$. The name *inclusive* scan comes from the fact that each output element *includes* the effect of the corresponding input element. Note that when the scan operator is addition, the computation is sometimes also referred to as *prefix sum*.

We can illustrate the applications for inclusive scan operations using an example of cutting sausage for a group of people. Assume that we have a 40-inch sausage to be served to 8 people. Each person has ordered a different amount in terms of inches: 3, 1, 7, 0, 4, 1, 6, 3. That is, Person 0 wants 3 inches of sausage, Person 1 wants 1 inch, and so on. We can cut the sausage either sequentially or in parallel. The sequential way is very straightforward. We first cut a 3-inch section for Person 0. The sausage is now 37 inches long. We then cut a 1-inch section for Person 1. The sausage becomes 36 inches long. We can continue to cut more sections until we serve the 3-inch section to Person 7. At that point, we have served a total of 25 inches of sausage, with 15 inches remaining.

With an inclusive scan operation, we can calculate the locations of all the cutting points based on the amount ordered by each person. That is, given an addition operation and an order input array [3, 1, 7, 0, 4, 1, 6, 3], the inclusive scan operation returns [3, 4, 11, 11, 15, 16, 22, 25]. The numbers in the return array are the cutting locations. With this information, one can simultaneously make all the eight cuts that will generate the sections that each person ordered. The first cut point is at the 3-inch location so the first section will be 3 inches, as ordered by Person 0. The second cut point is at the 4-inch location, therefore the second section will be 1-inch long, as ordered by Person 1. The final cut point will be at the 25-inch location, which will produce a 3-inch-long section since the previous cut point is at 22-inch point. This gives Person 7 what she ordered. Note that since all the cut points are known from the scan operation, all cuts can be done in parallel or in any arbitrary sequence.

In summary, an intuitive way of thinking about inclusive scan is that the operation takes a request from a group of people and identifies all the cut points that allow the orders to be served all at once. The order could be for sausage, bread, campground

space, or a contiguous chunk of memory in a computer. As long as we can quickly calculate all the cut points, all orders can be served in parallel.

An *exclusive scan* operation is similar to an inclusive scan operation with a slightly different arrangement of the output array:

$$[\text{ID}_{\oplus}, x_0, (x_0 \oplus x_1), \cdots, (x_0 \oplus x_1 \oplus \cdots \oplus x_{N-2})].$$

That is, each output element *excludes* the effect of the corresponding input element. The first output element is $\text{ID}_{\oplus}$, the identity value for operator $\oplus$, while the last output element only reflects the contribution of up to x_{N-2}. An identity value for a binary operator is defined as a value that when used as an input operand, it causes the operation to generate an output value that is the same as the other input operand's value. In the case of the addition operator, the identity value is 0 as any number added with zero will result in itself.

The applications of an exclusive scan operation are pretty much the same as those for inclusive scan. The inclusive scan provides slightly different information. In the sausage example, exclusive scan would return [0, 3, 4, 11, 11, 15, 16, 22], which are the beginning points of the cut sections. For example, the section for Person 0 starts at the 0-inch point, and the section for Person 7 starts at the 22-inch point. The beginning point information is important in applications such as memory allocation, where the allocated memory is returned to the requester via a pointer to its beginning point.

Note that it is easy to convert between the inclusive scan output and the exclusive scan output. One simply needs to shift all the elements and fill in an element. When converting from inclusive scan to exclusive scan, one can simply shift all the elements to the right and fill in the identity value for the 0th element. When converting from exclusive scan to inclusive scan, one can shift all elements to the left and fill in the last element with the previous last element $\oplus$ the last input element. It is just a matter of convenience that we can directly generate an inclusive or exclusive scan depending on whether we care about the cut points or the beginning points for the sections. Therefore, we will only present parallel algorithms and implementations for inclusive scan throughout this chapter.

Before we present parallel scan algorithms and their implementations, we would like to first show a sequential inclusive scan algorithm and its implementation. We will assume that the operator involved is addition. The code in Fig. 11.1 assumes that the input elements are in the `input` array and the output elements are to be written into the `output` array. The code initializes the output element `output[0]` with the value of input element `input[0]` (line 02). After that, in each iteration of the loop (lines 03-05), the loop body adds one more input element to the previous output element (which stores the accumulation of all the previous input elements) to generate one more output element.

It should be clear that the work done by the sequential implementation of inclusive scan in Fig. 11.1 is linearly proportional to the number of input elements. That is, the computational complexity of the sequential scan algorithm is $O(N)$.

```
01    void sequential_scan(float *input, float *output, unsigned int N) {
02        output[0] = input[0];
03        for(unsigned int i = 1; i < N; ++i) {
04            output[i] = output[i - 1] + input[i];
05        }
06    }
```

FIGURE 11.1

A simple sequential implementation of inclusive scan based on addition.

Similar to reduction, scan is a common operation that is available in many libraries. For example, the C++ standard library includes the functions `std::inclusive_scan` and `std::exclusive_scan`, and CUDA libraries such as Thrust [1,2] and CUB [3] also provide implementations of parallel scan that are highly optimized for GPUs. While programmers will use these libraries in practice, parallel scan is a great vehicle for demonstrating many parallelization and optimization techniques.

11.2 Parallel scan with the Kogge-Stone algorithm

To parallelize inclusive scan, one might be tempted to assign a thread to each output element that performs a sequential reduction to compute that element. After all, this approach allows the calculation of all output elements to be performed in parallel. However, this approach will unlikely improve the execution time over the sequential scan code in Fig. 11.1. The reason is that the calculation of y_{N-1} by the last thread would take N steps, and since the completion time of a parallel program is limited by the thread that takes the longest time, the span of such a parallelization would be $O(N)$. This span is no different from that of a sequential scan which also has a complexity of $O(N)$. In fact, with limited computing resources, the execution time of this naïve parallel scan algorithm can be much longer than the sequential algorithm.

To understand why this approach may have longer execution time, let us analyze the amount of work it performs. The work complexity, or the total number of operations performed, is much higher for the approach described. Since the number of computations for output element i would be i, the total number of computations performed by all threads would be:

$$\sum_{i=0}^{N-1} i = \frac{N \cdot (N-1)}{2}$$

That is, the proposed approach has a work complexity of $O(N^2)$, which is higher than the complexity of the sequential scan which is $O(N)$ while offering no speedup. The higher work complexity means that much more execution resources need to be provisioned, which is obviously a bad idea.

A better approach to parallel scan is to adapt the parallel reduction tree described in Chapter 10 to calculate each output element with a reduction tree on the relevant input elements. From what we learned in Chapter 10, the reader should realize that

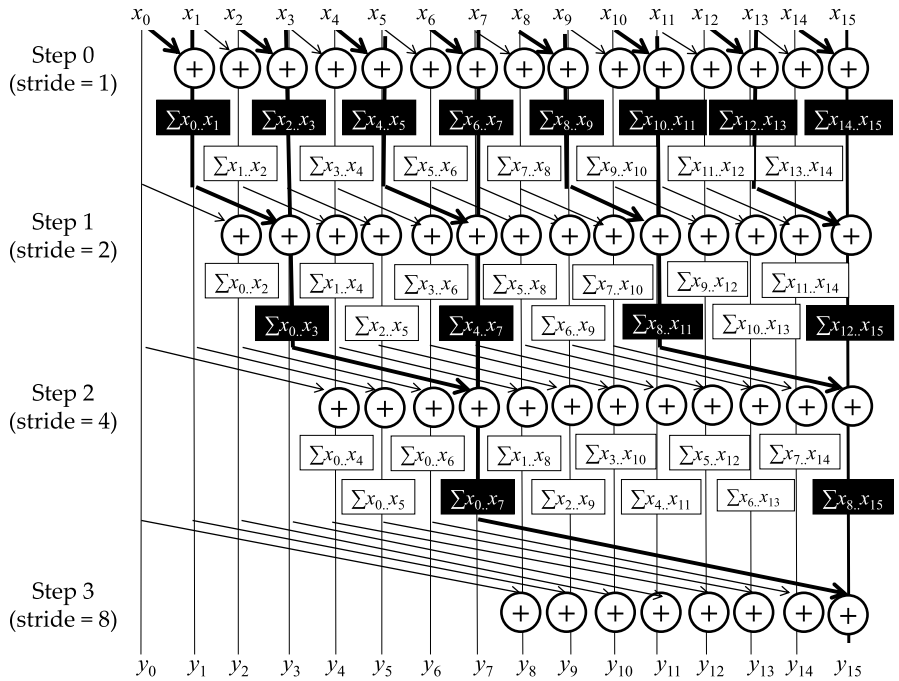


FIGURE 11.2

A parallel inclusive scan algorithm based on Kogge-Stone adder design. The heavy lines highlight the reduction tree for generating output element y_{15}.

using reduction trees requires that the scan operator to be associative, and even commutative depending on the optimizations used. There are multiple ways to design the reduction tree for each output element. Since the reduction tree for element i involves i add operations, this approach would still increase the work complexity to $O(N^2)$ unless we find a way to share the partial sums across the reduction trees of different output elements. We present such a sharing approach that is based on the Kogge-Stone algorithm which was originally invented for designing fast adder circuits in the 1970s [4]. This algorithm is still being used in the design of high-speed computer arithmetic hardware today.

The algorithm, illustrated in Fig. 11.2, is an in-place scan algorithm that operates on an array that originally contains input elements. It iteratively evolves the contents of the array into output elements. Before the algorithm begins, we assume that position i of the array contains input element x_i. After k iterations, position i will contain the sum of up to 2^k input elements at and before the location. For example, after one iteration, position i will contain $x_{i-1} + x_i$ and at the end of iteration 2, position i will contain $x_{i-3} + x_{i-2} + x_{i-1} + x_i$, and so on.

Fig. 11.2 shows the operations of the algorithm with a 16-element input example. Each vertical line represents an element of the array, with the 0th element in the

```
01    __global__ void scan_kernel(float *input, float *output, unsigned int N) {
02        __shared__ float buffer_s[SEG_SIZE];
03        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04        if(i < N) {
05            buffer_s[threadIdx.x] = input[i];
06        } else {
07            buffer_s[threadIdx.x] = 0.0f;
08        }
09        for(unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
10            __syncthreads();
11            float temp;
12            if(threadIdx.x >= stride)
13                temp = buffer_s[threadIdx.x] + buffer_s[threadIdx.x-stride];
14            __ syncthreads();
15            if(threadIdx.x >= stride)
16                buffer_s[threadIdx.x] = temp;
17        }
18        if(i < N) {
19            output[i] = buffer_s[threadIdx.x];
20        }
21    }
```

FIGURE 11.3

A Kogge-Stone kernel for inclusive (per-block) scan.

leftmost position. The vertical direction shows the progress of iterations, starting from the top of the figure. For inclusive scan, by definition, y_0 is x_0 so position 0 already contains its final answer. In the first iteration, each position other than position 0 receives the sum of its current content and that of its left neighbor. This is illustrated by the first row of addition operators in Fig. 11.2. As a result, position i contains $x_{i-1} + x_i$. This is reflected in the labeling boxes under the first row of addition operators in Fig. 11.2. For example, after the first iteration, position 3 contains $x_2 + x_3$, shown as $\sum x_2..x_3$. Note that after the first iteration, position 1 contains $x_0 + x_1$, which is the final answer for this position. So, there should be no further changes to position 1 in subsequent iterations.

In the second iteration, each position other than positions 0 and 1 receives the sum of its current content and that of the position that is two elements to its left. This is illustrated in the labeled boxes below the second row of addition operators. As a result, position i now contains $x_{i-3} + x_{i-2} + x_{i-1} + x_i$. For example, after the second iteration, position 3 contains $x_0 + x_1 + x_2 + x_3$, shown as $\sum x_0..x_3$. Note that after the second iteration, positions 2 and 3 have reached their final answers and will not need to be changed in subsequent iterations. The reader is encouraged to work through the rest of the iterations.

Fig. 11.3 shows a kernel that implements the algorithm illustrated in Fig. 11.2. Since the parallel scan operation requires synchronization across participating threads, we will for now implement a kernel where each thread block performs a local parallel scan on a segment of the input. Later, in Section 11.9, we will discuss how to consolidate these scanned segments to produce a globally scanned output array for the entire input array. The size of the block's segment is defined as a compile-time constant `SEG_SIZE`. We assume that the kernel function is called using `SEG_SIZE` as

the block size so there will be the same number of threads as segment elements. We assign each thread to evolve the contents of one of the elements in the segment.

The implementation shown in Fig. 11.3 assumes that input values were originally in a global memory array `input`, whose address is passed into the kernel as an argument (line 01). First, the threads in the block collaboratively load the segment of the input array that the block is responsible for into a shared memory array `buffer_s` (line 02). To do so, each thread calculates its global data index `i = blockIdx.x*blockDim.x + threadIdx.x` to find the position of the element it is responsible for evolving (line 03). Each thread then loads the input element at that position into the shared memory array (lines 04-08). At the end of the kernel, each thread writes its final result from the shared memory array to the output array `output` (lines 18-20).

We now focus on the for-loop (lines 09-17) that executes the concurrent reduction trees for generating `buffer_s` elements. Similar to the implementation of the first version of the reduction tree in Chapter 10, the loop index is the `stride` which doubles every iteration. In every iteration, each thread adds the element that it is responsible for evolving to the element that is `stride` positions away to its left (line 13) and writes the result back at the thread's position (line 16). When the stride value becomes greater than a thread's `threadIdx.x` value, it means that the thread's assigned `buffer_s` position has already accumulated all the required input values, and the thread no longer needs to be active (lines 12 and 15). Two barrier synchronizations are needed in the loop to ensure that there are no race conditions. The first barrier synchronization (line 10) ensures that all threads have finished writing the values they calculated to `buffer_s` in the previous iteration before other threads read them in the current iteration. The second barrier synchronization (line 14) ensures that all threads have finished reading their input values from `buffer_s` before other threads overwrite them.

The control divergence introduced by the conditions on lines 12 and 15 is quite modest. When the stride values are smaller than the size of a warp, only threads in the first warp drop out causing only the first warp to exhibit control divergence. For large thread blocks with many warps, the impact of this divergence is hardly felt. Once the stride reaches and exceeds the size of a warp, entire warps drop out from the first half of the thread block resulting in no control divergence.

The first barrier synchronization on line 10 is the same use of `__syncthreads()` as in the reduction kernel implemented in Chapter 10. However, the second barrier synchronization is not needed in the reduction kernel and is only needed in the scan kernel. Let us further examine why we would have a race condition if we did not have the second barrier. In each loop iteration, each active thread adds the `buffer_s` value at its own position (`buffer_s[threadIdx.x]`) and that at a position of another thread (`buffer_s`[threadIdx.x-stride]). If a thread i writes to its output position before the thread i + `stride` reads the old value at that position, the new value would corrupt the addition performed by thread i + `stride`.

For example, in Fig. 11.2, let us examine the activities of thread 4 and thread 6 during iteration 2, which is represented as the addition operations in the second row from the top. Thread 6 needs to add the old value of `buffer_s[4]` ($x_3 + x_4$) to the old value of `buffer_s[6]` ($x_5 + x_6$) to generate the new value of `buffer_s[6]` ($x_3 + x_4$

$+ x_5 + x_6$). However, if thread 4 stores its addition result for the iteration ($x_1 + x_2 + x_3 + x_4$) into `buffer_s[4]` too early, thread 6 could end up using the new value as its input and store ($x_1 + x_2 + x_3 + x_4 + x_5 + x_6$) into `buffer_s[6]`. Since $x_1 + x_2$ will be added again to `buffer_s[6]` by thread 6 in the third iteration, the final answer in `buffer_s[6]` will become ($2x_1 + 2x_2 + x_3 + x_4 + x_5 + x_6$), which is obviously incorrect. On the other hand, if thread 6 happens to read the old value in `buffer_s[4]` before thread 4 overwrites it during iteration 2, the results would be correct. That is, the execution result of the code may or may not be correct depending on the timing of thread execution and the execution results can vary from run to run. Such lack of reproducibility can make debugging a nightmare.

The race condition is overcome in Fig. 11.3 with the temporary variable `temp` used in line 13 and the `__syncthreads()` barrier in line 14. In line 13, all active threads first perform addition and write into their private `temp` variables. Therefore, none of the old values in `buffer_s` locations will be overwritten. The barrier __syncthread() in line 14 ensures that all active threads have completed their read of the old `buffer_s` values before any of them can move forward and perform a write. Thus, it is safe for the statement in line 16 to overwrite the `buffer_s` locations.

The reader may wonder why the reduction tree kernels in Chapter 10 did not need to use a second `__syncthreads()`. The answer is that the elements written to by the active threads in an iteration are not read by any of the other active threads during the same iteration. This should be apparent by inspecting Figs. 10.6 and 10.7. For example, in Fig. 10.7, each active threads takes its inputs from its own position (`input[threadIdx.x]`) and a position that is of stride distance to the right (`input[threadIdx.x+stride]`). None of the stride-distance positions are updated by any active threads during any given iteration. Therefore, all active threads will always be able to read the old value of their respective `input[threadIdx.x]` safely. The reader should verify that the same property holds in Fig. 10.6. In contrast, in the parallel scan kernel, the reason why a `buffer_s` position being written to may be read by another active thread is because of the interaction between the concurrent reduction trees being executed. In particular, the Kogge-Stone approach reuses the partial sums across reduction trees to reduce the computational complexity. We will further study this point in Section 11.5.

11.3 Double-buffering to reduce synchronization

One of the main bottlenecks in the parallel scan kernel that we implemented in Fig. 11.3 is the overhead of barrier synchronization. Every loop iteration must perform two calls to `__syncthreads()` to order concurrent memory accesses to avoid race conditions. However, we observe that these two barriers do not enforce the same kind of memory access ordering. The first barrier (line 10) ensures that all threads finish writing their computed values before other threads read them. In other words, it enforces a true dependence. On the other hand, the second barrier (line 14) ensures that

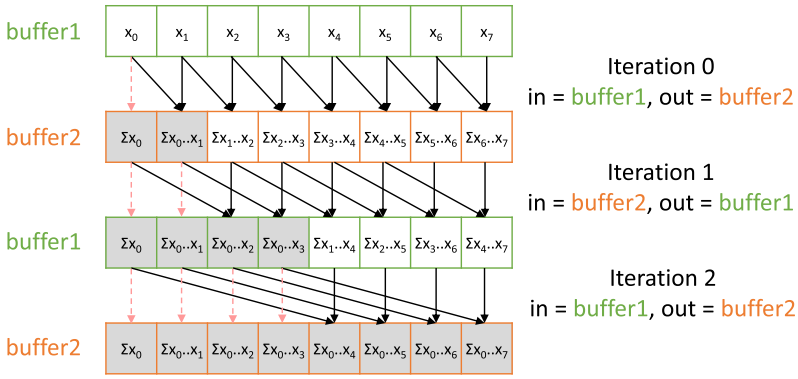


FIGURE 11.4

Double buffering applied to Kogge-Stone parallel scan.

all threads finish reading the values they need before other threads overwrite them. In other words, it enforces a false dependence.

Recall from Chapter 6 that we can eliminate barrier synchronizations that enforce false dependences via double-buffering. In general, we can avoid overwriting values that are being read in the same iteration by using different memory buffers to store the loop iteration's input values and its output values. One way to do so is to allocate a different output buffer for each iteration, which is also the input buffer to the next iteration. However, this approach would require a large amount of shared memory resources. Alternatively, we observe that once the output values of an iteration have been computed, the input values to that iteration are no longer needed. Hence, the input buffer of an iteration can be recycled and used as the output buffer of the next iteration. Overall, it is sufficient to allocate two buffers in total that keep trading places as the input and output buffers.

Fig. 11.4 illustrates how double-buffering can be applied to parallel scan. We use two shared memory buffers called `buffer1` and `buffer2`. Initially, we load the input elements from the global memory to `buffer1`. Next, in iteration 0, we read the values from `buffer1`, perform the addition, and write the sums to `buffer2`. Note that if we did not use two different buffers, then the value x_1 would be read by thread 2 from the same location where thread 1 writes its result $\sum x_1..x_2$, so a barrier would be needed to make sure that thread 1 does not write $\sum x_1..x_2$ before thread 2 reads x_1. However, since we are using two different buffers, the synchronization is not needed because thread 1 is writing $\sum x_1..x_2$ to `buffer2`, whereas thread 1 is reading x_1 from `buffer1`. After iteration 0 is over, `buffer2` has the most up-to-date results and the values in `buffer1` are no longer needed. Hence, in iteration 1, we use `buffer2` as our input buffer and write our results to `buffer1`. Following the same reasoning, in the third iteration, we read from `buffer1` and write to `buffer2`. We alternate the input and output buffers until the iterations complete.

An important observation in Fig. 11.4 is that since we are using different buffers for input and output, the values that are complete and are no longer evolving (e.g.,

```
01    __global__ void scan_kernel(float* input, float* output, unsigned int N) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        __shared__ float buffer1_s[SEG_SIZE];
04        __shared__ float buffer2_s[SEG_SIZE];
05        float* inBuffer_s = buffer1_s;
06        float* outBuffer_s = buffer2_s;
07        inBuffer_s[threadIdx.x] = (i < N)?input[i]:0.0f;
08        for(unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
09           __syncthreads();
10            if(threadIdx.x >= stride) {
11                outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x]
12                                         + inBuffer_s[threadIdx.x - stride];
13            } else {
14                outBuffer_s[threadIdx.x] = inBuffer_s[threadIdx.x];
15            }
16            float* temp = inBuffer_s;
17            inBuffer_s = outBuffer_s;
18            outBuffer_s = temp;
19        }
20      if(i < N) {
21          output[i] = inBuffer_s[threadIdx.x];
22      }
23    }
```

FIGURE 11.5

A Kogge-Stone kernel for inclusive (per-block) scan with double-buffering applied.

x_0 in iteration 0) are no longer retained automatically. They need be explicitly copied from the input buffer to the output buffer. Hence, the threads responsible for evolving these values (e.g., thread 0 in iteration 0), which became inactive in the original kernel, are now responsible for copying the values they are evolving from the input buffer to the output buffer. For example, in iteration 0, thread 0 is responsible for copying x_0 from position 0 in `buffer1` to position 0 in `buffer2`.

Fig. 11.5 shows the kernel code for parallel scan with the double-buffering optimization applied. Like the kernel in Fig. 11.3, each thread first identifies the position in the input array that it is responsible for evolving (line 02). Next, two shared memory buffers, `buffer1_s` and `buffer2_s`, are created instead of just one (lines 03-04). Moreover, two pointers, `inBuffer_s` and `outBuffer_s`, are created to assist with swapping the two buffers each iteration (lines 05-06). They are initialized to point to `buffer1_s` and `buffer2_s`, respectively, which are to be the input and output buffers, respectively, in the first iteration. After that, each thread loads the input value it is responsible for evolving from the global memory to the input buffer in shared memory (line 07).

Next, we begin the main loop for executing the Kogge-Stone algorithm (lines 08-19). The loop header (line 08) is the same as the kernel in Fig. 11.3, and the condition for the threads responsible for performing an addition (line 10) is also the same. However, an important difference is that the threads involved in the addition do not read and write the same buffer, but instead, they load their input values from `inBuffer_s` and write their computed sum to `outBuffer_s` (lines 11-12). Meanwhile, the threads not involved in the addition simply copy their value from the input buffer to the output buffer (line 14). At the end of the loop body, the input buffer and the output buffer are swapped (lines 16-18).

Notice that the loop in Fig. 11.5 only contains one call to `__syncthreads()` (line 09). This call enforces the true dependence whereby threads wait for their input values to be written by other threads in the previous iteration before they read them. The second `__syncthreads()` is no longer needed because all reads are happening from `inBuffer_s` and all writes are happening to `outBuffer_s`, so there is no write-after-read (false) dependence.

11.4 Warp-level primitives to reduce synchronization

While the overhead of barrier synchronization is reduced by double-buffering, it remains substantial. In the kernel in Fig. 11.5, every loop iteration calls `__syncthreads()` in conjunction with just a single floating point addition. Moreover, every loop iteration also performs three accesses to shared memory. It is clear that the performance of the loop is dominated by barrier synchronizations and shared memory accesses.

Recall from Chapter 10 that one way to reduce the overhead of barrier synchronizations and shared memory accesses is to leverage warp-level primitives, which allow threads within the warp to synchronize and share data much more efficiently than threads across an entire block. Hence, to reduce the overhead of barrier synchronizations and shared memory accesses for parallel scan, we can decompose our computation across multiple warps to take advantage of warp-level primitives. In particular, we can allow each warp to perform its own local scan on the segment belonging to the threads in the warp, then combine the results to yield a scanned segment for the entire block.

One way to decompose a scan operation into multiple smaller scan operations is using the *scan-scan-add* decomposition. The scan-scan-add decomposition divides the scan operation into three stages: a scan operation on smaller segments, a scan operation on the sums of the segments, and an add operation that updates each segment with the sum of previous segments. Fig. 11.6 shows a small example of the scan-scan-add decomposition. In this example, the input has 16 elements and is decomposed into four segments with four elements each. In the first stage, each segment is scanned independently. After this stage, each element contains the scan result within its segment. For example, the second segment has inputs 0, 4, 1, 2. The first scan stage produces the scan result for this segment, which is 0, 4, 5, 7. Note that these results do not contain the contributions from any of the elements in the previous segment. To produce the final result for this segment, the sum of all elements in the previous segment, $2 + 1 + 3 + 1 = 7$, should be added to every result element of the segment. As another example, the inputs of the third segment are 0, 3, 1, 2. After the first scan stage, the scanned segment values are 0, 3, 4, 6. In order to produce the final results for this segment, the sum of all elements in the two previous segments, $2 + 1 + 3 + 1 + 0 + 4 + 1 + 2 = 14$, should be added to every result.

After the first scan stage, the last element of each scanned segment gives the sum of all input elements of that segment. These values are 7, 7, 6, and 11 for the four

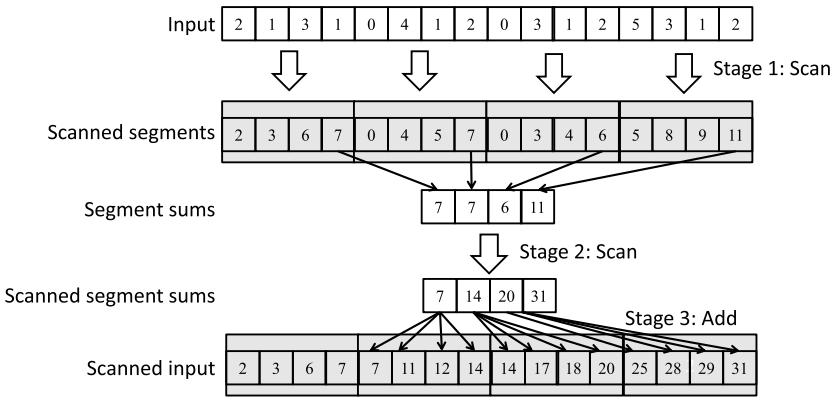


FIGURE 11.6

An example of the scan-scan-add decomposition.

segments in Fig. 11.6. This brings us to the second stage of the scan-scan-add decomposition, which gathers the last result from each segment into an array of sums and performs a scan operation on those sums. The result of that second scan operation is an array of scanned sums: 7, 14, 20, 31. Note that each of these values is the accumulated sum from the first element in the original input to the end of the corresponding segment. That is, the first value 7 is the accumulated sum from the first input element to the end of the first segment. The second value 14 is the accumulated sum from the first input element to the end of the second segment. Therefore, the values in the array of scanned sums give the scan results at strategic locations of the original scan problem. That is, scanned sums at indices 0, 1, 2, and 3 give the final scan results for the original input at positions 3, 7, 11, and 15, respectively. These results can be used to bring the partial results in each segment to their final values.

This brings us to the last step of the scan-scan-add decomposition in Fig. 11.6. The scanned sums are added to the values of their subsequent segments. For example the first scanned sum 7 is added to all the elements in the second segment, which completes them. The final results of the second segment are 7, 11, 12, 14. As another example, the second scanned sum 14 is added to the elements of the third segment, which completes them. These final results are 14, 17, 18, and 20. Similarly, the third scanned sum 20 is added to the elements of the fourth segment, which completes them. The fourth scanned sum 31 is the sum of all elements of the original input, which is also the final result in the last element.

Having seen how to decompose a scan operation into multiple smaller scan operations, we now apply this decomposition in the context of decomposing a block-level scan operation across multiple warps. Fig. 11.7 shows how a block-level scan can be decomposed into warp-level scans using the scan-scan-add decomposition. On the top of the figure, we start with the block's input segment that has already been loaded from global memory. Next, every warp performs a warp-level scan on the warp's

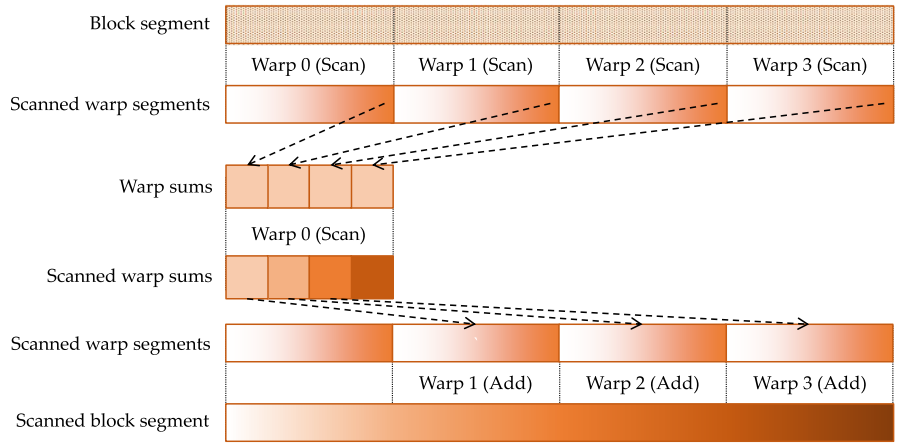


FIGURE 11.7

Decomposing a block-level scan into warp-level scans using the scan-scan-add decomposition.

contiguous subsegment of the block's segment. Once we have scanned the warp subsegments, we still need to add the sum of all the elements belonging to the preceding warps to complete the block's scanned segment. To do so, we observe that the last thread of each warp has the sum of all the elements belonging to the warp. Therefore, we can have this thread write that sum to a shared memory array of warp sums, i.e., the sums of the elements in each warp. Next, we have a single warp perform a warp-level scan on these warp sums. The result is an array that has, for each warp, the sum of the elements belonging to it and all preceding warps. These values are the values we would like to add to the warp subsegments. Accordingly, we have each warp load the value that represents the sum of all previous warps' values and add that value to all its own scanned elements, which yields the scanned block segment.

We are now ready to show how this approach can be implemented in code. We start by showing how a warp-level scan can be executed by a warp. Fig. 11.8 shows a device function that performs a parallel warp-level scan using the Kogge-Stone approach. The device function takes a parameter `val` that contains the value the thread in the warp is responsible for evolving (line 01). Similar to the codes in Figs. 11.3 and 11.5, the loop starts at stride one, and doubles it until it reaches the size of the segment being scanned, which in this case is the size of the warp (line 02). In each iteration, the thread obtains the value owned by the thread that is `stride` to its left. In this case, the value is obtained using a shuffle-up instruction (line 03), so no shared memory accesses or barrier synchronizations are needed. Once each thread has obtained the value it needs, it adds it to its own value (line 05). Similar to the codes in Figs. 11.3 and 11.5, only the threads whose indices are at least the stride value participate in the addition (line 04). However, we cannot use `threadIdx.x` as the index in the condition because we are doing a warp-level scan, so we need the thread's index

```
01    __device__ inline float warpScan(float val) {
02        for(unsigned int stride = 1; stride < WARP_SIZE; stride *= 2) {
03            float leftVal = __shfl_up_sync(0xffffffff, val, stride);
04            if(laneIdx() >= stride) {
05                val += leftVal;
06            }
07        }
08        return val;
09    }
```

FIGURE 11.8

A device function that performs a warp-level scan using warp-level primitives.

```
01  __device__ inline float blockScan(float val) {
02      // Warp-level scan
03      val = warpScan(val);
04      // Collect warp sums
05      __shared__ float warpSums_s[NUM_WARPS];
06      if(laneIdx() == WARP_SIZE - 1) {
07          warpSums_s[warpIdx()] = val;
08      }
09      __syncthreads();
10      // Scan warp sums
11      if(warpIdx() == 0) {
12          float warpSum = (threadIdx.x<NUM_WARPS)?warpSums_s[threadIdx.x]:0.0f;
13          warpSum = warpScan(warpSum);
14          if(threadIdx.x < NUM_WARPS) {
15              warpSums_s[threadIdx.x] = warpSum;
16          }
17      }
18      __syncthreads();
19      // Add previous warp's scanned sum
20      if(warpIdx() > 0) {
21          val += warpSums_s[warpIdx() - 1];
22      }
23      return val;
24  }
```

FIGURE 11.9

A device function that performs a block-level scan using the scan-scan-add decomposition and the warp-level scan.

within the warp, i.e., its lane index. To obtain the thread's index in the warp, we call a device function `laneIdx()` which is defined in Chapter 10. After the loop is finished, the device function returns `val` which contains each thread's evolved value after the wap-level scan is complete.

Having written a warp-level scan operation, our next step is to write the block-level scan operation that uses it. Fig. 11.9 shows a device function that performs a block-level scan using the scan-scan-add decomposition and the warp-level scan. The device function takes a parameter `val` that contains the value the thread in the block is responsible for evolving (line 01). The first step is for each warp to perform its own warp-level scan (line 03). Next, the last thread in each warp (line 06) writes its warp's sum to a shared memory array (line 07). Here, the array is indexed with the warp's index within the block, which is obtained by calling the device function `warpIdx()` which is defined in Chapter 10. A barrier synchronization ensures that all

```
01    __global__ void scan_kernel(float* input, float* output, unsigned int N) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        float val = (i < N)?input[i]:0.0f;
04        val = blockScan(val);
05        if(i < N) {
06            output[i] = val;
07        }
08    }
```

FIGURE 11.10

A block-level kernel that uses the block-level scan function which uses the warp-level scan.

warps have written their sums to the shared memory (line 09). After that, the first warp (line 11) has each thread load one of the warp sums (line 12) and performs a warp-level scan on the warp sums (line 13). The scanned warp sums are then written to shared memory (line 15) and a barrier synchronization ensures that all the other warps wait for this first warp to finish (line 18). Finally, all the threads except those in the first warp (line 20) read the sum of all the preceding warps' elements from shared memory and add it to their own value (line 21). The value is finally returned (line 23).

An important distinction between the code in Fig. 11.9 and the code in Fig. 11.5 is the number of barrier synchronizations and shared memory accesses performed. The code in Fig. 11.5 performs barrier synchronizations and shared memory accesses in every loop iteration. On the other hand, the code in Fig. 11.5 only performs barrier synchronizations and shared memory accesses on two occasions: once between the first warp-level scan stage and the second warp-level scan stage, and once between the second warp-level scan stage and the final addition stage.

Having written the block-level scan function, the block-level scan kernel is quite simple as shown in Fig. 11.10. Every thread identifies the element it is responsible for evolving (line 02), loads it from global memory (line 03), calls the block scan function (line 04), then writes the final value back to global memory.

11.5 Work efficiency considerations

One important consideration when analyzing parallel algorithms is *work efficiency*. The work efficiency of an algorithm refers to the extent to which the work performed by the algorithm is close to the minimum amount of work needed for the computation.[1] For example, the minimum number of additions required by the scan operation is $N - 1$ additions, or $O(N)$, which is the number of additions that the sequential algorithm performs. However, as we saw in the beginning of Section 11.2, the naïve parallel algorithm performs $\frac{N \cdot (N-1)}{2}$ additions, or $O(N^2)$, which is substantially larger than the sequential algorithm. For this reason, the naïve parallel algorithm is not work efficient.

[1] The minimum amount of work for the computation is empirically defined by the minimum amount of work performed among all known algorithms.

Let us analyze the work efficiency of the Kogge-Stone parallel scan algorithm illustrated in Fig. 11.2. Let N be the number of elements in the segment being scanned. We observe that the algorithm iterates for $log_2 N$ steps. In each step, out of a total of N threads, the number of inactive threads is equal to the $stride$, which means that the number of active threads is $N - stride$. Each active thread performs one addition operation. Therefore, we can calculate the amount of work done for the algorithm as

$$\sum_{stride}(N - stride), \text{ for } stride = 1, 2, 4, \cdots N/2 \ (log_2 N \text{ terms})$$

The first part of each term is independent of stride, its summation adds up to $N \cdot log_2 N$. The second part is a familiar geometric series and sums up to $N - 1$. Hence, the total amount of work done by the computation is:

$$N \cdot log_2 N - (N - 1)$$

When executed on a real CUDA GPU device, the amount of resources consumed by the Kogge-Stone parallel algorithm is even more than the theoretical $N \cdot log_2 N - (N - 1)$. This reason is that while many of the N threads stop participating in later iterations, some of them still consume execution resources until the entire warp that they are part of completes execution. Hence, the amount of execution resources consumed by the Kogge-Stone parallel algorithm is closer to $N \cdot log_2 N$. Either way, the work complexity of the Kogge-Stone approach is $O(N \cdot log_2 N)$.

The good news is that the work complexity of the Kogge-Stone approach is better than the $O(N^2)$ complexity of the naïve approach that performs sequential reductions for all the output elements. The bad news is that the Kogge-Stone algorithm is still not as work efficient as the sequential algorithm which performs $O(N)$ operations. Even for modest sized sections, the parallel algorithm does much more work than the sequential algorithm. For example, in the case of 512 elements, the algorithm does between eight and nine times more work than the sequential code.

Although the Kogge-Stone algorithm performs more computations than the sequential algorithm, it does so in fewer steps because of parallel execution. The loop of the sequential code executes N iterations, which means that the sequential algorithm has a time complexity of $O(N)$. As for the kernel code, the loop(s) execute up to $log_2 N$ iterations, which means that the parallel implementation has a span of $O(log\ N)$. With unlimited execution resources, the reduction in the number of steps of the kernel code over the sequential code would be approximately $N/log_2 N$. For $N = 512$, the reduction in the number of steps would be about $512/9 = 56.9\times$.

We will use the concept of steps as an approximate indicator for comparing between scan algorithms. The sequential scan should take approximately N steps to process N input elements. For example, the sequential scan should take approximately 1024 steps to process 1024 input elements. With P execution units in the CUDA device, we can expect the Kogge-Stone kernel to execute for $(N \cdot log_2 N)/P$ steps. If P was equal to N, i.e., if we had enough execution units to process all input element in parallel, then we need $log_2 N$ steps as we saw earlier. However, P could be smaller than N. For example, if we use 1024 threads and 32 execution units to process 1024 input elements, the kernel will likely take $(1024 * 10)/32 = 320$ steps.

In this case, we expect to achieve a $1024/320 = 3.2\times$ reduction in the number of steps.

The additional work done by the Kogge-Stone kernel over the sequential code is problematic in two ways. First, the use of hardware for executing the parallel kernel is much less efficient. If the hardware does not have enough resources (i.e., if P is small), the parallel algorithm could end up needing more steps than the sequential algorithm and the parallel algorithm will be slower. Second, even if the kernel is faster, the extra work done consumes additional energy. This makes the kernel less appropriate for power-constrained environments such as mobile applications.

11.6 Coarsening to improve work-efficiency

We have seen in Section 11.5 that the main overhead of parallelizing scan is the reduction in work efficiency. We also saw that with enough execution resources, this overhead is worth paying because of the reduction in the span of the execution. However, with constrained execution resources where the hardware begins to serialize the execution of threads, the overhead is incurred unnecessarily. Instead, we can serialize some of the work ourselves with thread coarsening, thereby reducing the parallelization overhead of the kernel and improving its work efficiency.

To apply thread coarsening to scan, we need to assign each thread to perform a work efficient sequential scan of multiple input elements before the threads collaborate to perform the parallel scan. We can do so by using the scan-scan-add decomposition to decompose the scan operation into independent thread-level scans. Fig. 11.11 illustrates how thread coarsening can be applied to parallel scan using the scan-scan-add decomposition. On the top of the figure, we start with the block's input

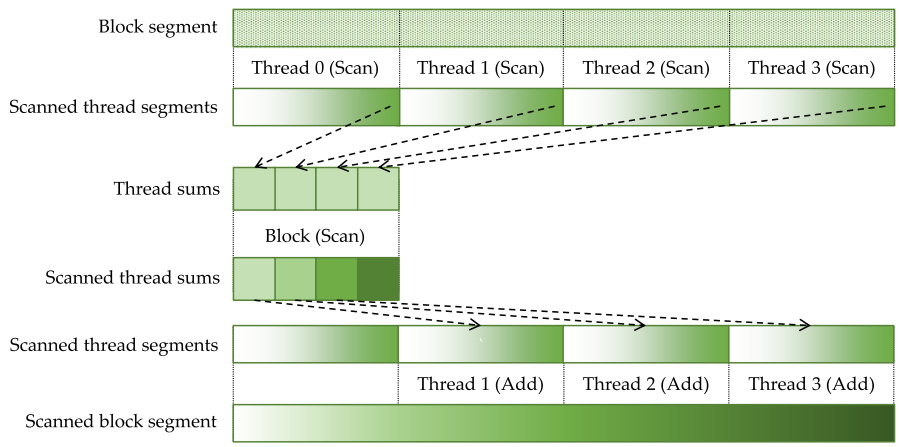


FIGURE 11.11

Applying thread coarsening to parallel scan using the scan-scan-add decomposition.

```
01    __global__ void scan_kernel(float* input, float* output, unsigned int N) {
02        unsigned int blockSegment = blockIdx.x*COARSE_FACTOR*BLOCK_DIM;
03        // Load data to shared memory
04        __shared__ float buffer_s[COARSE_FACTOR*BLOCK_DIM];
05        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
06            buffer_s[c*BLOCK_DIM + threadIdx.x] =
07                                input[blockSegment + c*BLOCK_DIM + threadIdx.x];
08        }
09        __syncthreads();
10        // Scan thread subsegment
11        unsigned int threadSegment = threadIdx.x*COARSE_FACTOR;
12        for(unsigned int c = 1; c < COARSE_FACTOR; ++c) {
13            buffer_s[threadSegment + c] += buffer_s[threadSegment + c - 1];
14        }
15        // Block-wide scan of thread sums
16        float threadSum = buffer_s[threadSegment + COARSE_FACTOR - 1];
17        threadSum = blockScan(threadSum);
18        // Collect thread partial sums
19        __shared__ float threadSums[BLOCK_DIM];
20        threadSums[threadIdx.x] = threadSum;
21        __syncthreads();
22        // Add previous thread's partial sums
23        if(threadIdx.x > 0) {
24            float prevPartialSum = threadSums[threadIdx.x - 1];
25            for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
26                buffer_s[threadSegment + c] += prevPartialSum;
27            }
28        }
29        __syncthreads();
30        // Write output
31        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
32            output[blockSegment + c*BLOCK_DIM + threadIdx.x] =
33                                buffer_s[c*BLOCK_DIM + threadIdx.x];
34        }
35    }
```

FIGURE 11.12

A parallel scan kernel with thread coarsening applied using the scan-scan-add decomposition.

segment that has already been loaded from global memory to shared memory. The number of elements in this block segment is greater than the number of threads in the block by some factor which is the coarsening factor. We partition the block's segment into multiple contiguous subsegments, one for each thread. Next, every thread performs a sequential scan of its subsegment, which is work efficient. Once we have scanned the thread subsegments, each thread uses its last value, which represents the sum of all the elements owned by the thread, and participates in a block-level scan. After the block-level scan, each thread obtains the value that represents the sum of all previous threads' values and adds that value to all its own scanned values, which yields the scanned block segment.

Fig. 11.12 shows the code for a parallel scan kernel with thread coarsening applied using the scan-scan-add decomposition. We define a constant `COARSE_FACTOR` that determines the coarsening factor, i.e., the number of elements assigned to each thread. We also define the block dimension as a constant `BLOCK_DIM` for convenience. The size of the segment owned by a single block is therefore `COARSE_FACTOR*BLOCK_DIM`.

The code in Fig. 11.12 starts by identifying the location of the segment the block is responsible for by multiplying the block index with the segment size (line 02).

Next, a shared memory buffer for storing the segment is declared (line 04) and the segment is loaded from global memory to shared memory (lines 05-09). Importantly, although each thread needs to load `COARSE_FACTOR` input elements from global memory to shared memory, we do not have each thread load the contiguous subsegment it owns because doing so would lead to uncoalesced accesses. For example, if our coarsening factor was four, then during the first iteration of block 0, thread 0 would access input element 0, thread 1 would access input element 4, thread 2 would access input element 8, etc, which are not contiguous. Instead, we transfer data from global memory to shared memory in a coalesced manner by dividing the block's segment into `COARSE_FACTOR` contiguous chunks that are each `BLOCK_DIM` in size and loading them one chunk at a time (lines 05-08). For example, if our coarsening factor was four, then during the first iteration of block 0, thread 0 loads element 0, thread 1 loads element 1, thread 2 loads element 2, etc. Finally, a barrier synchronization (line 09) ensures that the whole segment is loaded before threads proceed to access their subsegments.

Once the block segment is in shared memory, each thread identifies the location of its subsegment by multiplying its index with the size of the subsegment (line 11). Next, each thread performs a sequential scan operation (lines 12-14) similar to that in Fig. 11.1. Next, each thread loads its final scanned value, which represents the sum of all elements owned by the thread (line 16), and passes it to the `blockSum` device function from Fig. 11.9, which performs a block-level scan of the thread sums (line 17). Next, the threads place their scanned thread sums in shared memory (lines 19-20) and perform a barrier synchronization to ensure that the other threads can read them (line 21). With the scanned thread sums now available, all threads except the first one (line 23) load the scanned thread sum of the previous thread (line 24) which represents the sum of all the elements owned by the preceding threads. Each thread then iterates through its subsegment and adds the sum to all the values it owns (line 25-27). The scanned block segment is now complete in shared memory.

The final step is to store the scanned block segment from shared memory to global memory. Similar to the concerns when loading the input block segment, we cannot have each thread store the subsegment it owns because the stores would not be coalesced. Instead, the threads execute a barrier synchronization to ensure that all threads finish computing their subsegments (line 29). Finally, we divide the output segment into `COARSE_FACTOR` contiguous chunks that are each `BLOCK_DIM` in size and store them one chunk at a time (lines 31-34).

Let us now analyze the work efficiency of parallel scan after thread coarsening is applied. Assume that we are scanning N input elements using P workers (i.e., threads). With coarsening, each of the P threads performs a sequential scan of N/P elements, which is a total of $P \cdot (N/P - 1) = N - P$ operations. Next, the P thread sums are scanned in parallel which requires $P \cdot log_2 P$ operations. Finally, each of the P threads except the first adds the scanned thread sums to its N/P scanned values, which requires $(P - 1) \cdot N/P = N - N/P$ operations. Overall, the number of

operations performed with thread coarsening is $2N + P \cdot log_2 P - P - N/P$. Hence, when P is large and close to N, the number of operations is $O(N \cdot log_2 N)$ which is similar to the parallel implementation without coarsening. However, when P is much smaller than N, the number of operations is closer to N and is $O(N)$ like the sequential implementation. This analysis shows that thread coarsening substantially improves the parallel scan work efficiency when the number of execution resources is significantly smaller than the number of input elements.

The improvement in work efficiency also impacts the number of steps performed. With coarsening, each thread performs a sequential scan of N/P elements which takes $N/P - 1$ steps. Next, the P thread sums are scanned in parallel which takes $log_2 P$ steps. Finally, the threads add the scanned thread sums to their scanned values which takes N/P steps. Overall, the approach with thread coarsening requires $2N/P + log_2 P - 1$ steps. Recall that without thread coarsening, the number of steps is approximately $\frac{N \cdot log_2 N}{P}$. Hence, when P is large and close to N, both approaches take $O(log_2 N)$ steps. However, when P is much smaller than N, the approach without thread coarsening takes approximately $\frac{N \cdot log_2 N}{P}$ steps, while the approach with thread coarsening takes approximately $\frac{2N}{P}$ steps, which is a substantial improvement.

Thread coarsening has other benefits to parallel scan besides improving work efficiency. The parallel block-level scan needs to perform barrier synchronization and incurs modest control divergence, whereas the sequential thread-level scan does not. When we move more operations from the parallel block-level scan to the sequential thread-level scan, we reduce the impact of the barrier synchronization and control divergence overhead on overall performance. This benefit is similar to the benefit we see from applying thread coarsening to the reduction kernel in Chapter 10.

Yet another benefit of thread coarsening will become more apparent in Section 11.9. Recall that so far we have been implementing a scan kernel where each thread block scans its own segment, and we have deferred the discussion on how to consolidate the results of different blocks to Section 11.9. When we apply thread coarsening to scan and assign each thread to multiple input elements, one way we can do that is by keeping the size of the block segment the same and reducing the number of threads in the block by the coarsening factor. Another approach is to keep the number of threads in the block the same and increase the size of the block segment by the coarsening factor. The latter approach would in turn reduce the number of thread blocks by the coarsening factor. Reducing the number of thread blocks participating in the global scan operation is favorable when we need to consolidate the segments of the different thread blocks. We will revisit this point in Section 11.9.

11.7 Register tiling to avoid shared memory access latency

In the kernel in Fig. 11.12, the subsegments owned by each thread were placed in shared memory. Hence, whenever a thread needs to access its subsegment's elements, it must access shared memory. Since the elements of a thread's subsegment are private to the thread and are not needed by other threads, it is wasteful for the thread

```
01    __global__ void scan_kernel(float* input, float* output, unsigned int N) {
02        unsigned int blockSegment = blockIdx.x*COARSE_FACTOR*BLOCK_DIM;
03        // Load data to shared memory
04        __shared__ float buffer_s[COARSE_FACTOR*BLOCK_DIM];
05        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
06            buffer_s[c*BLOCK_DIM + threadIdx.x] =
07                              input[blockSegment + c*BLOCK_DIM + threadIdx.x];
08        }
09        __syncthreads();
10        // Scan thread subsegment
11        unsigned int threadSegment = threadIdx.x*COARSE_FACTOR;
12        float buffer_r[COARSE_FACTOR];
13        buffer_r[0] = buffer_s[threadSegment];
14        #pragma unroll
15        for(unsigned int c = 1; c < COARSE_FACTOR; ++c) {
16            buffer_r[c] = buffer_s[threadSegment + c] + buffer_r[c - 1];
17        }
18        // Block-wide scan of thread sums
19        float threadSum = buffer_r[COARSE_FACTOR - 1];
20        threadSum = blockScan(threadSum);
21        // Collect thread partial sums
22        __shared__ float threadSums[BLOCK_DIM];
23        threadSums[threadIdx.x] = threadSum;
24        __syncthreads();
25        // Add previous thread's partial sums
26        float prevPartialSum = (threadIdx.x>0)?threadSums[threadIdx.x - 1]:0.0f;
27        #pragma unroll
28        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
29            buffer_s[threadSegment + c] = buffer_r[c] + prevPartialSum;
30        }
31        __syncthreads();
32        // Write output
33        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
34            output[blockSegment + c*BLOCK_DIM + threadIdx.x] =
35                              buffer_s[c*BLOCK_DIM + threadIdx.x];
36        }
37    }
```

FIGURE 11.13

A parallel scan kernel with register tiling applied to thread subsegments.

to repeatedly load and store the elements of its subsegment from and to the shared memory. Instead, we can apply register tiling where the thread places the elements of its subsegments in registers and repeatedly accesses them from there.

Fig. 11.13 shows a parallel scan kernel with register tiling applied to the thread subsegments. Most of the kernel resembles the kernel that applies thread coarsening in Fig. 11.12. The key differences are as follows. We declare a local array `buffer_r` that is used to store the thread's subsegment (line 12). As we scan the thread's subsegment, the first time we load it from shared memory, we keep it in the local array (lines 13 and 16). Subsequent accesses to the thread's subsegment are made using the local array (lines 16, 19, and 29). Finally, as we add the preceding threads' sum, we return the thread's subsegment to shared memory (line 29).

Note that all the loops that iterate over `buffer_r` are annotated with `#pragma unroll` (lines 14 and 27) to ensure that the compiler unrolls the loops. Recall from Chapter 6 that loop unrolling ensures that the local array is accessed with a constant index and

can therefore be promoted to a register. In practice, the compiler most likely unrolls the loops in the kernel without needing this annotation. However, the annotation is a useful reminder to the reader that the loop is intended to be unrolled.

It is noteworthy that the kernel still loads the thread subsegments to shared memory (line 06) before loading them to the register tile (lines 13 and 16), and stores the thread subsegments to shared memory (line 29) before storing them to global memory (line 34). The reason shared memory is still retained as an intermediary is that if a thread were to transfer data between global memory and its register tile directly, the accesses would not be coalesced. Hence, this kernel uses shared memory not for the purpose of tiling since the data in shared memory is not reused. The kernel uses shared memory just to enable coalesced accesses to global memory. This optimization is similar to the corner turning optimization discussed in Chapter 6.

There are still additional optimizations from the checklist in Chapter 6 that we can apply to the scan kernel in Fig. 11.13. First, since each thread is responsible for loading multiple elements from global memory to shared memory (lines 05-08 in Fig. 11.13) and storing multiple elements from shared memory to global memory (lines 33-36 in Fig. 11.13), the threads can use vector loads and stores instead of scalar ones. Second, when threads access their thread segments in shared memory (lines 13 and 29 in Fig. 11.13), the strided access pattern can cause bank conflicts. These bank conflicts can be eliminated by adding padding to the shared memory array, as described in Chapter 6. We leave these optimizations as an exercise to the reader.

11.8 Memory bandwidth considerations

A scan operation on an array of N floating point elements loads N floating point values (or $4 \cdot N$ bytes) from memory, performs $N - 1$ floating point additions, and stores N floating point values (or $4 \cdot N$ bytes) to memory. Hence, the arithmetic intensity of the scan operation is $\frac{(N-1)}{4 \cdot N + 4 \cdot N} \approx \frac{1}{8}$ OP/B. Recall from Chapter 5 that the NVIDIA H100 GPU has a peak floating point throughput of 66.9 TFLOPS and a peak global memory bandwidth of 3.35 TB/s, thereby requiring a computation to perform at least $\frac{66.9}{3.35} = 20.0$ FLOP/B to be compute-bound. Clearly, the arithmetic intensity of the scan operation is well below this threshold, making scan a highly memory-bound computation.

For memory-bound computations, the peak memory bandwidth sets the limit for the peak performance achievable by a computation. On an NVIDIA H100 GPU with a peak global memory bandwidth of 3.35 TB/s, since scan needs to access at least 8 B of global memory for every element it scans, the peak performance of a scan operation is $\frac{3.35 \times 10^{12}}{8} = 419 \times 10^9$ elements/s. In other words, the fastest possible scan kernel cannot scan more than 419 billion elements per second on an NVIDIA H100 GPU. Achieving this 419×10^9 elements/s target would require the kernel to use the memory bandwidth perfectly, accessing each global memory location only once and hiding the latency of all the other compute, synchronization, and shared memory

access operations behind the global memory accesses. In practice, such perfect utilization of the memory bandwidth is hard to achieve. Even a kernel that simply copies data from one array to another without performing any computations or synchronization cannot achieve an absolute 100% utilization of the global memory bandwidth. Typically, a memory-bound kernel that achieves > 80% utilization of global memory bandwidth without having redundant global memory accesses is considered to be well-optimized.

The scan kernel that we have implemented so far in Fig. 11.13 does not perform any redundant global memory accesses, and is thoroughly optimized to limit the overhead of computation, synchronization, and shared memory accesses. Accordingly, this kernel can hide most of the work it performs behind the global memory accesses and achieve > 80% utilization of the global memory bandwidth. However, recall that this kernel only performs local scans, where each thread block scans a block segment that it owns. Consolidating the block segments into a global scanned segment requires more global memory accesses and synchronization overhead, which poses a challenge for retaining the kernel's global memory access efficiency. We look deeply at this issue in Section 11.9.

11.9 Consolidating block segments for a global scan

The kernels presented so far perform block-wide scans on segments of the input. For some applications, a block-wide scan may be all we need. However, for other applications, the number of elements to be processed by a scan operation can be in the millions or even billions. For such applications, we need to perform a grid-wide scan using multiple thread blocks, and we need a way to consolidate the different block segments.

One way to decompose a grid-wide scan across multiple thread blocks is to use the scan-scan-add decomposition, as illustrated in Fig. 11.14. On the top of the figure, we start with the global input to be scanned. We partition the input into multiple contiguous segments that can each be processed by a single thread block. Next, every thread block scans its segment using one of the kernels we have seen so far, such as the most optimized kernel in Fig. 11.13. Once we have scanned the block segments, each block contributes its last value, which represents the sum of all the elements owned by the block, to a global inter-block scan. After the inter-block scan, each thread obtains the value that represents the sum of all previous blocks' values and adds that value to all its own scanned values, which yields the final globally scanned output.

The main challenge with this strategy is how to perform the inter-block scan, which requires synchronization across the thread blocks after they finish their local scans. A simple approach to perform the inter-block scan is to use separate kernel calls for each of the three stages. In other words, we call a first kernel to perform the local scan of the block segments, a second kernel to scan the block sums using a single thread block, and a third kernel to add the scanned block sums.

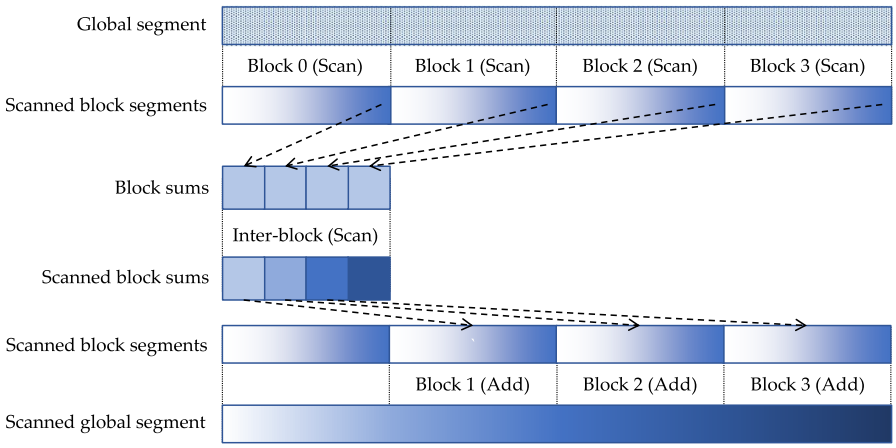


FIGURE 11.14

Decomposing a global scan into block-level scans using the scan-scan-add decomposition.

The main drawback of this three-kernel approach is that the intermediate scanned values need to be written to and read from global memory between kernel calls. In particular, when scanning N elements, the first kernel which performs the local scans loads N elements from global memory and stores N elements to global memory. The second kernel which scans the block sums loads $\frac{N}{S}$ elements from global memory and stores $\frac{N}{S}$ elements to global memory, where S is the number of elements per block segment. The third kernel which adds the scanned block sums loads N elements from global memory and stores N elements to global memory. Overall, the three-kernel approach accesses a total of $N + N + \frac{N}{S} + \frac{N}{S} + N + N = (4 + \frac{2}{S}) \cdot N$ elements, or $(16 + \frac{8}{S}) \cdot N$ bytes.

For a very large S where $\frac{8}{S}$ is negligible, the three-kernel approach accesses $\approx 16 \cdot N$ bytes to scan N elements, or 16 B per element. On an NVIDIA H100 GPU with a peak global memory bandwidth of 3.35 TB/s, the peak performance of this approach is $\frac{3.35 \times 10^{12}}{16} = 209 \times 10^9$ elements/s. This peak performance is half the peak performance of an ideal scan implementation which was found in Section 11.8 to be 419×10^9 elements/s. Intuitively, this approach can only achieve half the performance of the ideal implementation because it performs twice the number of global memory accesses due to the need to store all N intermediate values at the end of the first kernel and load them back at the beginning of the third kernel.

To avoid performing such a large number of global memory accesses, let us explore alternative strategies for decomposing a global scan across multiple thread blocks. One alternative decomposition is the reduce-scan-scan decomposition which has three stages: a reduce stage followed by two scan stages. Fig. 11.15 illustrates the reduce-scan-scan decomposition applied to decomposing a grid-wide scan across multiple thread blocks. On the top of the figure, we start with the global input to be

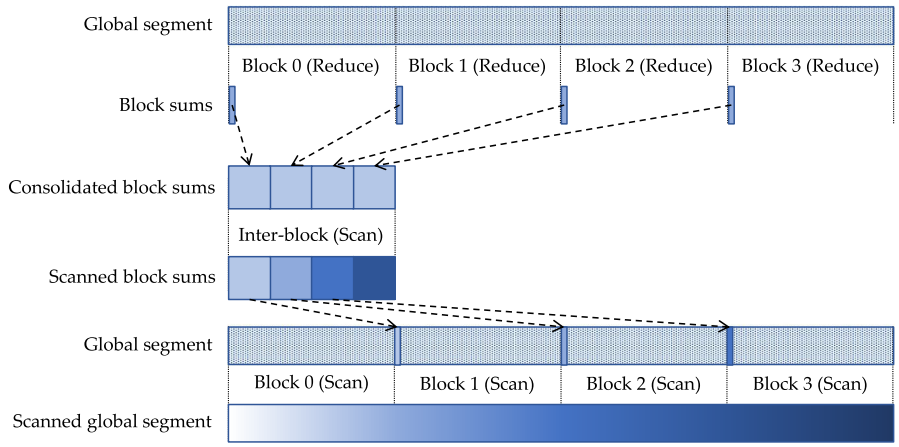


FIGURE 11.15

Reduce-scan-scan decomposition of a global scan into block-level scans.

scanned and partition it to multiple contiguous segments each processed by a different thread block. Every thread block starts by performing a reduction operation on its segment using a reduction kernel similar to those implemented in Chapter 10. Next, each block contributes the sum it computed to a global inter-block scan. After the inter-block scan, each thread block adds the sum of all previous blocks' values to the first element in its block segment, then performs a scan of the block segment yielding the final globally scanned array.

Similar to the *scan-scan-add* decomposition, the *reduce-scan-scan* decomposition also requires an inter-block scan step. Again, a simple approach for performing the inter-block scan is to use a three-kernel approach: the first kernel to reduce the block segments, the second kernel to scan the block sums using a single thread block, and the third kernel to perform the local scans. However, when applying the three-kernel approach to the *reduce-scan-scan* decomposition, the number of global memory accesses is not as high.

In particular, when scanning N elements, the first kernel which performs the reduction loads N elements from global memory and stores $\frac{N}{S}$ elements to global memory, where S is the number of elements per block segment. The second kernel which scans the block sums loads $\frac{N}{S}$ elements from global memory and stores $\frac{N}{S}$ elements to global memory. The third kernel which performs the local scans loads N elements from global memory and stores N elements to global memory. Overall, the three-kernel approach with the *reduce-scan-scan* decomposition accesses a total of $N + \frac{N}{S} + \frac{N}{S} + \frac{N}{S} + N + N = (3 + \frac{3}{S}) \cdot N$ elements, or $(12 + \frac{12}{S}) \cdot N$ bytes.

For a very large S where $\frac{12}{S}$ is negligible, the three-kernel approach with the *reduce-scan-scan* decomposition accesses $\approx 12 \cdot N$ bytes to scan N elements, or 12 B per element. On an NVIDIA H100 GPU with a peak global memory bandwidth

of 3.35 TB/s, the peak performance of this approach is $\frac{3.35\times10^{12}}{12} = 279 \times 10^9$ elements/s. This peak performance is two-thirds the 419×10^9 elements/s peak performance of the ideal scan implementation, which is better than the three-kernel approach with the *scan-scan-add*. Intuitively, the three-kernel approach with the *reduce-scan-scan* decomposition avoids having to store N elements at the end of the first kernel, but still needs to load N elements at the beginning of the third kernel.

An important consideration when using the *reduce-scan-scan* decomposition is operator commutativity. So far, all our parallelization and optimization strategies for scan have only assumed that the scan operator is associative. However, some optimizations for the reduction pattern in Chapter 10 also assumed that the operator is commutative. Therefore, when implementing the *reduce-scan-scan* decomposition, we must be careful about which implementation of reduction to use depending on whether or not the scan operator is commutative.

After seeing that the *reduce-scan-scan* decomposition outperforms the *scan-scan-add* decomposition in this context, the reader may wonder if the *reduce-scan-scan* decomposition would also perform better when decomposing the block-wide scan across warps in Section 11.4 and across threads in Section 11.6. To answer this question, let us compare the two decomposition strategies more closely. Both strategies have two scan operations. They key difference is that one has an add operation while the other has a reduce operation. The advantage of the reduce operation over the add operation is that it writes to fewer outputs. On the other hand, the advantage of the add operation over the reduce operation is that it does not need to perform multiple steps with intermediate synchronization to execute a reduction tree. When decomposing the global computation across blocks, our primary concern is to limit accesses to the array elements in global memory. Hence, the reduce operation which performs fewer accesses is more favorable. On the other hand, when decomposing the block-wide scan across warps or threads, the elements are already in the registers. Accessing the elements more frequently from the registers in the add operation is much cheaper than the synchronization overhead incurred by the reduce operation. Accordingly, the *scan-scan-add* decomposition is better when decomposing the block-wide scan across warps and threads.

Although the three-kernel approach with the *reduce-scan-scan* decomposition accesses fewer bytes in global memory than with the *scan-scan-add* decomposition, it still falls short from the ideal scan implementation because of the need to load the input from global memory twice. To come closer to ideal performance, we must load the entire input array only once and store the entire output array only once, which cannot be achieved with a multi-kernel implementation. For this reason, we need to use an alternative method for the inter-block scan that does not involve terminating a kernel and calling a new one [5–7].

Fig. 11.16 shows three alternative methods for implementing an inter-block scan without terminating the kernel and calling a new one. The first method, illustrated in Fig. 11.16(a), requires all thread blocks to share their sums with one thread block, such as block 0, which scans the block sums and redistributes the scanned sums. This method would require all the thread blocks in the grid to perform grid-wide barrier

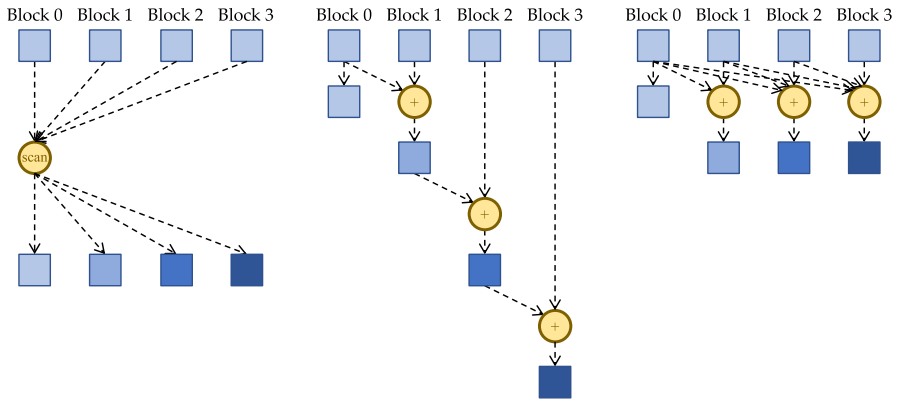


FIGURE 11.16

Alternative methods of inter-block scan within a single kernel.

synchronizations before and after block 0 performs the scan operation. Such a grid-wide barrier synchronization can be done using cooperative groups, as we will see in Chapter 18. However, it requires that all thread blocks in the grid run simultaneously, which limits the number of thread blocks that can be configured. Since each thread block has a limit on the size of the block segment it can scan due to the limited shared memory and register capacity in an SM, the limit on the number of thread blocks limits the total size of the array that can be scanned with this method. More importantly, grid-wide barrier synchronizations are very expensive operations, and this method requires performing it twice.

The second method, shown in Fig. 11.16(b), involves passing sums from one thread block to the next and scanning them along the way. In other words, each thread block waits for the previous thread block's scanned sum, adds the previous thread block's scanned sum to its own sum to find its own scanned sum, then passes its own scanned sum to the next thread block. This method is referred to as the *single lookback* method because each thread block looks back to a single preceding thread block.

The advantage of the single lookback method over the method in Fig. 11.16(a) is that it does not require a grid-wide barrier synchronization. Instead, each thread block only needs to wait for the preceding thread blocks to complete the previous operation. We refer to this kind of synchronization as *unidirectional synchronization*. Unlike barrier synchronization, unidirectional synchronization does not require all participating thread blocks to be active simultaneously. It only requires earlier thread blocks in the grid to be scheduled before later thread blocks. This requirement can be achieved by assigning a block index to a thread block only after it starts executing, instead of relying on its default `blockIdx.x` value. We will see how this dynamic assignment can be done later in this section.

```
01   __device__ inline float interBlockScan(float val, unsigned int bid,
02       float* partialSums, unsigned int* flags) {
03       __shared__ float prevBlockPartialSum_s;
04       if(threadIdx.x == blockDim.x - 1) {
05           if(bid > 0) {
06               // Wait for previous block to pass partial sum
07               cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
08                   prevBlockFlag_ref(flags[bid - 1]);
09               while(prevBlockFlag_ref.fetch_add(0, cuda::memory_order_acquire)==0){}
10               // Read the previous block's partial sum
11               prevBlockPartialSum_s = partialSums[bid - 1];
12           } else {
13               prevBlockPartialSum_s = 0.0f;
14           }
15           // Write this block's partial sum
16           partialSums[bid] = prevBlockPartialSum_s + val;
17           // Set this block's flag
18           cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
19               myBlockFlag_ref(flags[bid]);
20           myBlockFlag_ref.fetch_add(1, cuda::memory_order_release);
21       }
22       __syncthreads();
23       return prevBlockPartialSum_s;
24   }
```

FIGURE 11.17

Code for inter-block scan with single lookback.

Fig. 11.17 shows the code for performing an inter-block scan using the single lookback method. The device function takes four parameters (lines 01-02): the value `val` that the thread block contributes to the scan, the block index `bid` that is dynamically assigned, an array `partialSums` where the thread blocks should place their scanned sums, and another array `flags` that the thread blocks use to indicate when their scanned sums are ready. The last thread in the block (line 04), which has the block's sum, acts on behalf of the whole block as the leader thread. The leader thread first checks if the block is the first block or not (line 05). If it is the first block, the scanned sum of the previous block is trivially zero because there is no previous block (line 13). Otherwise, the leader thread must wait until the previous block's scanned sum is ready.

To signal that its scanned sum is ready, the previous block sets its entry in the `flags` array to one. Hence, the leader thread waits for the previous block by repeatedly checking if the previous block's flag is still set to zero. Checking the flag should be done atomically because the flag is set by a thread in another block. The atomic operations should be performed with device scope to ensure the visibility of the updated flag value across multiple blocks in the device. For this reason, the thread creates a device-scope (`cuda::thread_scope_device`) atomic reference to the flag of the previous block (lines 07-08) and repeatedly adds zero to it and checks its old value (line 09). Since adding zero does not change the flag's value, the flag is expected to remain zero until the previous block sets it. At that point, the leader thread breaks out of the loop and loads the previous block's scanned sum from the `partialSums` array (line 11).

An important distinction between this use of atomic operations and the previous use in Chapter 9 is that the second argument to the `fetch_add` function is

`cuda::memory_order_acquire`, not `cuda::memory_order_relaxed`. The reason we do not use `cuda::memory_order_relaxed` is that in this case, we care about the order between this atomic operation and other memory accesses in the code. The purpose of the flag is to indicate that the scanned sum is ready, so the subsequent load of the scanned sum should not be reordered to occur before the atomic operation on the flag. Using `cuda::memory_order_acquire` indicates that the atomic operation has *acquire semantics*, meaning that subsequent memory instructions should not be reordered to occur before the atomic operation.

Once the leader thread has obtained the previous block's partial sum, it adds the obtained sum value to its own block's value `val` and stores the resulting scanned partial sum for its own block in the `partialSums` array (line 16). After that, the thread needs to set its own block's flag to indicate to the next block that its own block's scanned sum is ready. This update must also be done atomically and with device scope since the flag is being accessed simultaneously by the leader thread of the next block. For this reason, the thread creates an atomic reference to its block's flag with device scope (lines 18-19) and atomically adds one to it using `fetch_add` (line 20).

When incrementing the flag to one, we use yet another option for the second argument to the `fetch_add` function which is `cuda::memory_order_release`, not `cuda::memory_order_relaxed` or `cuda::memory_order_acquire`. For this atomic operation too, the order between the atomic operation and other memory accesses in the code is important. The purpose of the flag is to indicate that the scanned sum is ready, so the preceding store of the scanned sum should not be reordered to occur after the atomic operation on the flag. Using `cuda::memory_order_release` indicates that the atomic operation has *release semantics*, meaning that preceding memory instructions should not be reordered to occur after the atomic operation.

The final part of the inter-block scan function in Fig. 11.17 is to perform a barrier synchronization across all threads in the block (line 22). The barrier ensures that none of the threads in the block proceed before the leader thread has written the previous block's scanned sum to shared memory where the other threads can see it. Finally, all threads return the previous block's scanned sum to the caller.

While the single lookback method enables performing the scan operation in a single kernel, its performance can be actually quite poor. The disadvantage of the single lookback method is that it creates a long chain of dependence between thread blocks, which becomes a long-latency critical path in the execution of the kernel. For an array of size N and a block segment of size S, the kernel executes $\frac{N}{S}$ thread blocks. In this case, the method Fig. 11.16(a) would require two grid-wide barrier synchronizations, whereas the method in Fig. 11.16(b) would require $\frac{N}{S}$ unidirectional synchronizations. Hence, even if unidirectional synchronizations are more lightweight, they are much larger in number making the single lookback method inefficient when we have a large number of thread blocks.

The critical path in the single lookback method can be reduced by allowing the thread block to look back to multiple preceding thread blocks, which is known as *multiple lookback*. For example, if a thread block b finds that its preceding thread block $b-1$ has its sum ready but not its scanned sum, it can look back to thread

block $b-2$. If thread block $b-2$ has its scanned sum ready, then thread block b does not need to wait for thread block $b-1$ to add thread block $b-2$'s scanned sum. Thread block b can itself add thread block $b-1$'s sum to thread block $b-2$'s scanned sum and proceed. In doing so, we would have performed more redundant work since both thread blocks b and $b-1$ will add the scanned sum of thread block $b-2$ to the sum of thread block $b-1$. However, this redundant work would reduce the latency of waiting that thread block b has to do.

The multiple lookback method could look back for a fixed number of blocks or a variable number of blocks. That is, a thread block could keep checking a fixed number of preceding blocks until one of them has its scanned sum ready and the rest have their sums ready. Alternatively, a thread block can keep looking back further and further until it finds a scanned sum that is ready, then add the sums of all the later thread blocks. The more a thread block looks back, the more redundant work it performs but the longer the chain of dependence that it bypasses. The multiple lookback method where a thread blocks keeps looking back until it finds a scanned sum ready is referred to as *decoupled lookback*. We leave the implementation of a revised `interBlockScan` device function based on decoupled lookback as an exercise.

Fig. 11.16(c) shows decoupled lookback in its most extreme case where each thread block goes back and reduces all the preceding thread blocks' sums. It is visually clear that this method can finish in much fewer steps and synchronizations than the other two methods at the expense of more computations performed. This method should remind the reader of the naïve parallel scan presented in Section 11.2 where each thread performs its own reduction operation. Although the naïve parallel scan has the worst work efficiency, it has the lowest synchronization overhead. This property makes it favorable in the inter-block scan scenario where the cost of synchronization is much higher than the cost of compute, and the number of block sums to be reduced is much smaller than the total number of input elements.

Having seen the alternative methods for performing an inter-block scan without terminating the kernel, we can now look at the implementation of a single-kernel scan that includes an inter-block scan to consolidate block segments. Fig. 11.18 shows the code for such a kernel. The kernel follows the *scan-scan-add* decomposition. Since we perform the scan in a single kernel, there is no need to write the output of the first scan to global memory and read it back for the add operation. Therefore, the benefit of the *reduce-scan-scan* decomposition over the *scan-scan-add* decomposition is no longer applicable.

The kernel in Fig. 11.18 starts by assigning an index to the thread block dynamically (lines 04-10). Recall that the inter-block scan requires unidirectional synchronization, which requires earlier thread blocks to be scheduled before later thread blocks. This can be achieved by assigning the indices of the thread blocks dynamically in increasing order after the thread blocks are scheduled. To ensure that, once a thread block starts executing, one thread in the block (line 05) atomically increments a thread block counter (line 06-08) to obtain an index, which the other threads in the block wait for (line 10) and read from shared memory (line 11). The thread block counter is allocated in memory, initialized to zero, and passed as a parameter to the kernel (line 01).

```
01    __global__ void scan_kernel(float* input, float* output, unsigned int* blockCounter,
02        float* partialSums, unsigned int* flags unsigned int N) {
03        // Assign block index dynamically
04        __shared__ unsigned int bid_s;
05        if(threadIdx.x == 0) {
06            cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
07                blockCounter_ref(*blockCounter);
08            bid_s = blockCounter_ref.fetch_add(1, cuda::memory_order_relaxed);
09        }
10        __syncthreads();
11        unsigned int bid = bid_s;
12        unsigned int blockSegment = COARSE_FACTOR*bid*blockDim.x;
13        // Load data to shared memory
14        __shared__ float buffer_s[COARSE_FACTOR*BLOCK_DIM];
15        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
16            buffer_s[c*BLOCK_DIM + threadIdx.x] =
17                              input[blockSegment + c*BLOCK_DIM + threadIdx.x];
18        }
19        __syncthreads();
20        // Scan thread subsegment
21        unsigned int threadSegment = threadIdx.x*COARSE_FACTOR;
22        float buffer_r[COARSE_FACTOR];
23        buffer_r[0] = buffer_s[threadSegment];
24        #pragma unroll
25        for(unsigned int c = 1; c < COARSE_FACTOR; ++c) {
26            buffer_r[c] = buffer_s[threadSegment + c] + buffer_r[c - 1];
27        }
28        // Block-wide scan of thread sums
29        float threadSum = buffer_r[COARSE_FACTOR - 1];
30        threadSum = blockScan(threadSum);
31        // Collect thread partial sums
32        __shared__ float threadSums[BLOCK_DIM];
33        threadSums[threadIdx.x] = threadSum;
34        __syncthreads();
35        // Add previous thread's partial sums
36        float prevPartialSum = (threadIdx.x>0)?threadSums[threadIdx.x - 1]:0.0f;
37        #pragma unroll
38        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
39            buffer_r[c] += prevPartialSum;
40        }
41        // Scan block partial sums
42        float prevBlockPartialSum =
43            interBlockScan(buffer_r[COARSE_FACTOR - 1], bid, partialSums, flags);
44        // Add previous block's partial sum
45        #pragma unroll
46        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
47            buffer_s[threadSegment + c] = buffer_r[c] + prevBlockPartialSum;
48        }
49        __syncthreads();
50        // Write output
51        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
52            output[blockSegment + c*BLOCK_DIM + threadIdx.x] =
53                        buffer_s[c*BLOCK_DIM + threadIdx.x];
54        }
55    }
```

FIGURE 11.18

A scan kernel that includes an inter-block scan to consolidate block segments.

The next part of the kernel in Fig. 11.18 (lines 12-40) is mostly the same as the code in Fig. 11.13. A block segment is loaded to shared memory, each thread scans a thread subsegment, the block scans the thread sums, and the threads add the scanned thread sums to their segment yielding the scanned block segment. This point marks the completion of the first scan operation in the *scan-scan-add* decomposition.

For the second scan operation, the device function from Fig. 11.17 that performs an inter-block scan operation is called (lines 42-43). After the second scan operation, the add operation remains (lines 45-48). Each thread iterates over its thread segment

(line 46) and adds the previous block's scanned sum to all its elements (line 47). The final part of the kernel writes the block's segment to global memory (lines 51-54) in the same way as is done in the code in Fig. 11.13.

11.10 Parallel scan with the Brent-Kung algorithm

Throughout this chapter, we used the Kogge-Stone algorithm as the basis for our parallel implementation of the scan operation. We then observed that the Kogge-Stone algorithm exhibits poor work efficiency. In this section, we cover an alternative algorithm for parallelizing scan that exhibits better work efficiency at the expense of more computation steps. We cover this alternative algorithm for the completeness of the reader's knowledge, while noting that the Kogge-Stone algorithm's work efficiency limitation was overcome with thread coarsening making work efficiency no longer a major concern.

By inspecting the progression of the Kogge-Stone algorithm in Fig. 11.2, we can see that there are potential opportunities for further sharing of some intermediate results. However, to allow more sharing across multiple threads, we need to strategically calculate the intermediate results and distribute them to different threads, which may require additional computation steps.

As we know, the fastest parallel way to produce sum values for a set of values using a binary operator is a reduction tree. With sufficient execution units, a reduction tree can generate the sum for N values in $log_2 N$ time units. Furthermore, the tree can also generate several sub-sums that can be used in the calculation of some of the scan output values. This observation was used as a basis for the Kogge-Stone adder design, and also forms the basis of the Brent-Kung adder design [8]. The Brent-Kung adder design can also be followed to implement a parallel scan algorithm with better work efficiency.

Fig. 11.19 illustrates the steps for a parallel inclusive scan algorithm based on the Brent-Kung adder design. In the top half of Fig. 11.19, we produce the sum of all 16 elements in 4 steps. We use the minimal number of operations needed to generate the sum. During the first step, only the positions whose indices are of the form $2 \cdot n - 1$ are updated. During the second step, only the positions whose indices are of the form of $4 \cdot n - 1$ are updated. During the third step, only the positions whose indices are of the form $8 \cdot n - 1$ are updated. Finally, during the fourth step, only the last position 15 whose index is of the form $16 \cdot n - 1$ is updated. The total number of operations performed is $8 + 4 + 2 + 1 = 15$. In general, for a scan segment of N elements, we would do $\frac{N}{2} + \frac{N}{4} + \cdots + 2 + 1 = N - 1$ operations for this reduction phase.

The second part of the algorithm is to use a reverse tree to distribute the partial sums to the positions that can use them to complete the result of those positions. The distribution of partial sums is illustrated in the bottom half of Fig. 11.19. To understand the design of the reverse tree, we should first analyze the needs for additional values to complete the scan output at each position of the array. It should be apparent

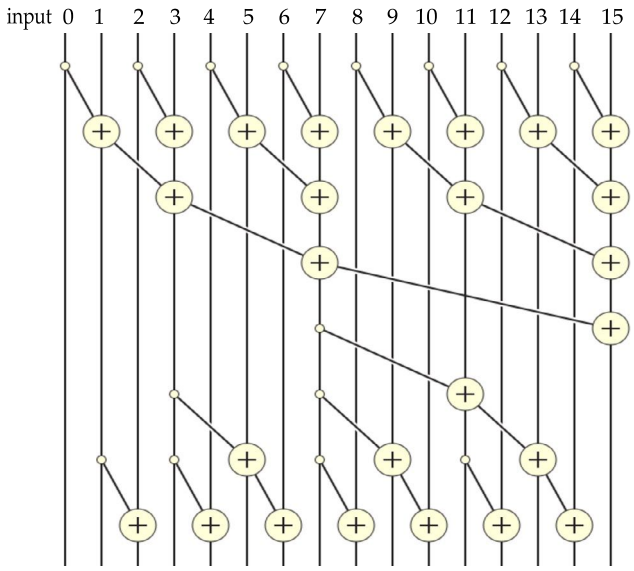


FIGURE 11.19

A parallel inclusive scan algorithm based on Brent-Kung adder design.

from an inspection of Fig. 11.19 that the additions in the reduction tree always accumulate input elements in a continuous range. Therefore, we know that the values that have been accumulated into each position of the array can always be expressed as a range of input elements $x_i..x_j$ where x_i is the starting position and x_j is the ending position (inclusive).

Fig. 11.20 shows the state of each position (column), both the values already accumulated into the position as well as the need for additional input element values at each level (row) of the reverse tree. The state of each position initially and after each level of additions in the reverse tree is expressed as the input elements, in the form of $x_i..x_j$, that have already been accounted for in the position. For example, $x_8..x_11$ in row Initial and column 11 indicates that the values of x_8, x_9, x_10, and x_11 have been accumulated into position 11 before the reverse tree begins (right after the reduction phase shown in the bottom portion of Fig. 11.19). At the end of the reduction tree phase, we have quite a few positions that are complete with the final scan values. In our example, positions 0, 1, 3, 7, and 15 are all complete with their final answers.

The need for additional input element values is indicated with the shade of each cell in Fig. 11.20. Black means that the position needs to accumulate partial sums from three other positions, dark gray means two, light gray means one, and white means zero. For example, initially, position 14 is marked black because it has only the value of x_14 at the end of the reduction tree phase and needs to accumulate the partial sums from positions 7 ($x_0..x_7$), 11 ($x_8..x_11$) and 13 ($x_12..x_13$) to complete

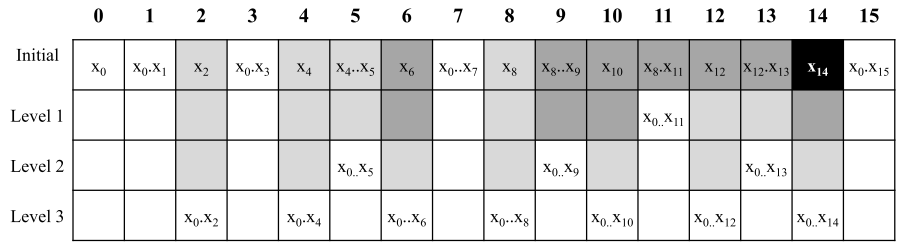


FIGURE 11.20

Progression of values in `input` after each level of additions in the reverse tree.

its final scan value ($x_0..x_14$). The reader should verify that because of the structure of the reduction tree, the array positions for an input of size N elements will never need to accumulate from more than $log_2N - 1$ partial sums from other positions. Furthermore, these partial sum positions will always be 1, 2, 4, $\cdots$ (powers of 2) away from each other. In our example, position 14 needs $log_2(16) - 1 = 3$ partial sums from positions that are 1 (between 14 and 13), 2 (between 13 and 11), and 4 (between 11 and 7).

To organize the second half of the addition operations, we first show all the operations that need partial sums from 4 positions away, then 2 positions away, and then 1 position way. During the first level of the reverse tree, we add the value at position 7 to that at position 11, which brings position 11 to the final answer. In Fig. 11.20, position 11 is the only one that advances to its final answer. During the second level, we complete positions 5, 9, and 13 that can be completed with the partial sums that are 2 positions away at 3, 7, and 11, respectively. Finally, during the third level, we complete the positions 2, 4, 6, 8, 10, 12, and 14 by accumulating the partial sums that are of 1 position away (immediate left neighbor of each position).

We now turn our attention to the analysis of the number of operations in the reverse tree stage. The number of operations is $\frac{16}{8} - 1 + \frac{16}{4} - 1 + \frac{16}{2} - 1$. In general, for N input elements, the total number of operations would be $(2-1)+(4-1)+\cdots+(\frac{N}{4}-1)+(\frac{N}{2}-1)$ which is $N-1-log_2N$. Therefore, the total number of operations in the parallel scan, including both the reduction tree phase ($N-1$ operations) and the reverse tree phase ($N-1-log_2N$ operations), is $2*N-2-log_2N$. Hence, the total number of operations is $O(N)$, as opposed to $O(N \cdot log_2N)$ for the Kogge-Stone algorithm.

The tradeoffs between the Brent-Kung algorithm and the Kogge-Stone algorithm are quite clear. The Brent-Kung algorithm has better work efficiency, performing fewer operations in total, at the expense of executing more steps. However, recall from Section 11.6 that after applying thread coarsening, most of the work in the parallel scan kernel is done by the work efficient thread-level sequential scans. The work inefficient parallel scan is only performed at the warp level and comprises a small part of the kernel's execution time.

In fact, when comparing the Kogge-Stone algorithm and the Brent-Kung algorithm at the warp-level, we find that the Kogge-Stone algorithm has better performance in practice. The work avoided by the Brent-Kung algorithm at the warp level

is replaced by inactive warp lanes that still consume execution resources because of the nature of SIMD execution. It is better to utilize those inactive warp lanes to perform more work but finish in fewer steps, as the Kogge-Stone algorithm does. Still, the Brent-Kung algorithm is an interesting case study for reasoning about the interactions between theoretical work efficiency and practical implementation considerations. We leave the implementation of the Brent-Kung algorithm for parallel scan as an exercise for the reader.

11.11 Summary

In this chapter, we studied parallel scan, also known as prefix-sum, as an important parallel computation pattern. Scan is used to enable parallel allocation of resources to parties whose needs are not uniform. It converts seemingly sequential computation based on mathematical recurrence into parallel computation, which helps to reduce sequential bottlenecks in many applications. We show that a simple sequential scan algorithm performs only $N - 1$, or $O(N)$, additions for an input of N elements.

We first introduced a parallel Kogge-Stone scan algorithm that is fast, conceptually simple, but not work-efficient. The algorithm performs $O(N \cdot log_2 N)$ operations which is more than its sequential counterpart. As the data set size increases, the number of execution units needed for a parallel algorithm to break even with the simple sequential algorithm also increases. Therefore, Kogge-Stone scan algorithms are typically used to process modest-sized scan blocks in processors with abundant execution resources.

We saw how to reduce the synchronization overhead of the scan kernel by using double-buffering and warp-level primitives. We also applied thread coarsening to improve its work efficiency. Thread coarsening was applied by having each thread in the block perform a work-efficient sequential scan on its own subsegment of input elements before the threads collaborate to perform the less work efficient block-wide parallel scan to produce the entire block's segment.

We presented two different decompositions of scan to consolidate scanned segments across multiple thread blocks: *scan-scan-add* and *reduce-scan-scan*. We saw that a three-kernel implementation of the *reduce-scan-scan* decomposition incurs fewer redundant global memory accesses than that of the *scan-scan-add* decomposition. We also saw that global memory accesses can be further reduced with a single-kernel scan that performs an inter-block scan using unidirectional synchronization between thread blocks. The unidirectional synchronization requires special care to prevent deadlocks by using dynamic block index assignment. Decoupled lookback was also presented to overcome the latency of the critical path created by unidirectional synchronization.

We then presented the Brent-Kung algorithm for parallel scan as an alternative to the Kogge-Stone algorithm. Using a reduction tree phase and a reverse tree phase, the algorithm performs only $O(N)$ additions. However, although the Brent-Kung algorithm has better work efficiency than the Kogge-Stone algorithm, it requires more

steps to complete. Moreover, its work efficiency advantage is obviated by the application of thread coarsening to improve work efficiency, and the SIMD nature of warp execution whereby the work avoided is replaced by inactive threads that still consume resources.

In general, implementing and optimizing parallel scan algorithms on GPUs is complex and programmers are encouraged to use a parallel scan library for GPUs such as Thrust [1,2] or CUB [3] than to implement their own scan kernels from scratch. Nevertheless, parallel scan is an important parallel pattern, and it offers an interesting and relevant case study of the trade-offs that go into optimizing parallel patterns.

11.12 Exercises

1. Consider the following array: [4 6 7 1 2 8 5 2]. Perform a parallel inclusive prefix scan on the array using the Kogge-Stone algorithm. Report the intermediate states of the array after each step.
2. Analyze the Kogge-Stone parallel scan kernel in Fig. 11.3. Show that control divergence only occurs in the first warp of each block for stride values up to half of the warp size. That is, for warp size 32, control divergence will occur to iterations for stride values 1, 2, 4, 8, and 16.
3. For the Kogge-Stone scan kernel in Fig. 11.3, assuming that we have 2048 elements, how many add operations are performed in total?
4. Modify the kernel in Fig. 11.13 to use vector loads and stores and to eliminated shared memory bank conflicts.
5. Modify the `interBlockScan` device function in Fig. 11.17 to use decoupled lookback instead of single-lookback.
6. Consider the following array: [4 6 7 1 2 8 5 2]. Perform a parallel inclusive prefix scan on the array using the Brent-Kung algorithm. Report the intermediate states of the array after each step.
7. Implement a CUDA kernel that executes the Brent-Kung algorithm to scan an array at the block level.

References

[1] N. Bell, J. Hoberock, Thrust: a productivity-oriented library for cuda, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.
[2] NVIDIA, Thrust: the C++ parallel algorithms library, https://nvidia.github.io/cccl/thrust/index.html, 2025.
[3] NVIDIA, CUB, https://nvidia.github.io/cccl/cub/index.html, 2025.
[4] P.M. Kogge, H.S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Transactions on Computers 100 (8) (1973) 786–793.

[5] Y. Dotsenko, N.K. Govindaraju, P.-P. Sloan, C. Boyd, J. Manferdelli, Fast scan algorithms on graphics processors, in: Proceedings of the 22nd Annual International Conference on Supercomputing, 2008, pp. 205–213.
[6] S. Yan, G. Long, Y. Zhang, Streamscan: fast scan algorithms for gpus without global barrier synchronization, in: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2013, pp. 229–238.
[7] D. Merrill, M. Garland, Single-pass parallel prefix scan with decoupled look-back, Tech. Rep. NVR-2016-002, NVIDIA, 2016.
[8] Brent Kung, A regular layout for parallel adders, IEEE Transactions on Computers 100 (3) (1982) 260–264.

This page intentionally left blank

CHAPTER

12 Filter
And warp voting

CONTENTS

12.1 Background 289
12.2 A simple parallel unstable filter 290
12.3 Coalescing atomic operations with warp-level primitives 291
12.4 Privatization 295
12.5 A simple parallel stable filter 297
12.6 Improving memory coalescing with shared memory and thread coarsening 298
12.7 In-place stable filter 300
12.8 Related patterns 301
12.9 Summary 302
12.10 Exercises 302
References 302

Filter operations rearrange data in a list to make more efficient use of memory after some of the list elements are deleted. If we view the deleted elements and the remaining elements as two sub-lists, a filter operation can be viewed as the inverse of a merge operation. In this chapter, we show that using a technique called *warp voting* we can enable coalescing of atomic operations across threads and improve the speed of parallel implementations of filter. We look at the considerations for performing the stable filter operation in place instead of out of place. Finally, we discuss other related patterns with similar implementations and optimizations, such as removing duplicates from a sorted list and adding/removing rows and columns from a matrix.

12.1 Background

Filtering refers to a computation that removes one or more selected items from a list according to a given criterion and compacts the list to eliminate the holes thus created. An example use case is garbage collection. After the application de-allocates some objects and frees their chunks of memory from the heap, there can be many small chunks of available memory. At some point, it may be desirable to compact the remaining objects on the heap into consecutive locations and create a much larger chunk of available memory space at the end of the heap. In this way, if the application needs to allocate a large object, there is sufficient consecutive memory available.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00022-6
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

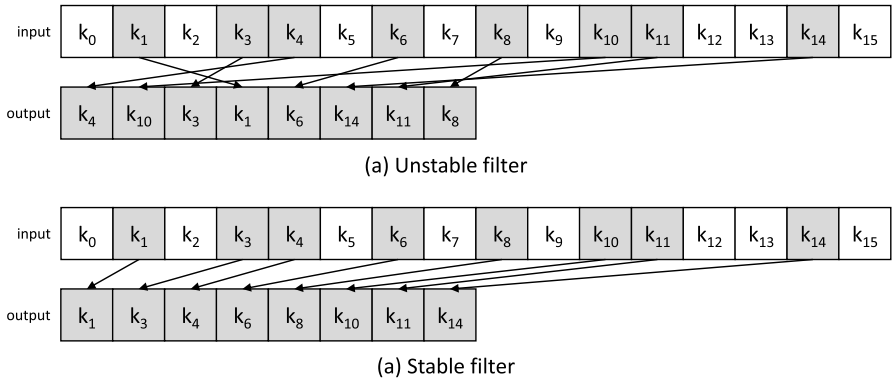


FIGURE 12.1

Unstable vs. stable variants of the out-of-place filter pattern.

When the list is very large, it becomes desirable to employ many threads to move the items in parallel to reduce execution time.

Filters can be in-place or out-of-place. In-place filters compact the remaining items in the original list, whereas out-of-place filters compact them into a new list. In the memory management example, creating a new list means that enough memory must be set aside to accommodate the entire new heap, drastically increasing the inefficiency of memory usage. However, moving an item to an earlier position in the list may overwrite the element that is currently in that position waiting to be moved to an earlier position. Thus, when multiple threads are employed to move the items in a list, the threads must synchronize so that no items would be incorrectly overwritten before they can be moved.

In this chapter, we will first learn to implement out-of-place filters that are often used in data search/retrieval applications to generate a relevant list by removing irrelevant items. The general approach to parallelizing out-of-place filters is to have multiple threads to simultaneously move the remaining items into the new list. We then introduce in-place filter implementation techniques that can be built on top of the scan pattern introduced in Chapter 11.

Fig. 12.1 shows examples of the two variants of the filter operation. Following the convention in computer science, we will refer to the items in a filtered list as keys. The shaded keys in the input list indicate the keys that remain where as unshaded keys are deleted. Stable filters preserve the original order of the remaining keys, whereas unstable filters do not. Unstable filters are used when the list does not need to be ordered. We will begin with parallel implementation of unstable filters.

12.2 A simple parallel unstable filter

Fig. 12.2 shows the code for a simple unstable filter kernel. The kernel takes an `input` array of size `N` and filters it into an `output` array. The parameter `outputSize` points to a

```
01  __global__ void filter_kernel(unsigned int* input, unsigned int* output,
02      unsigned int N, unsigned int* outputSize) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) {
05          unsigned int val = input[i];
06          if(cond(val)) {
07              cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
08                  outputSize_ref(*outputSize);
09              unsigned int j = outputSize_ref.fetch_add(1, cuda::memory_order_relaxed);
10              output[j] = val;
11          }
12      }
13  }
```

FIGURE 12.2

Code for a simple unstable filter kernel.

global memory counter variable where the size of the output array is to be stored and is initialized to zero before the kernel is called. Every thread identifies the index of the key in the input that it is responsible for (line 03), checks if that index is within bounds (line 04), and loads the key at that index from global memory (line 05). The thread then checks if the loaded value meets the condition `cond()` by which we are filtering (line 06). If the condition is met, the thread allocates space for that value in the output array by atomically incrementing the counter at `outputSize` (line 07-09). Finally, the thread stores the value to the allocated space (line 10).

Note that this implementation results in an unstable filter operation because the atomic operations on the counter may be performed by the hardware in any order. Hence, each thread will acquire an arbitrary place in the output array to put its value in irrespective of the input key's position in the input array.

A clear bottleneck in this implementation is the contention caused by the atomic operations on the output counter in global memory. This contention is exacerbated by the fact that all the atomic operations are performed on the *same* counter, so they are all serialized by the hardware. As we learned in Chapter 9, output contention can be mitigated via the *privatization* optimization which we look at in Section 12.4. However, before looking at privatization, we will first look at another technique for reducing contention, called *coalesced atomic operations*, that leverages the fact that the atomic operations are performed on the same counter by threads of the same warp.

12.3 Coalescing atomic operations with warp-level primitives

In the simple unstable filter kernel in Fig. 12.2, atomic operations contend on the same output counter causing a performance bottleneck. This contention is particularly pronounced for threads in the same warp that issue their atomic operations at the same time. In this case, the atomic operations are certain to conflict with each other and get serialized by the hardware. However, we can take advantage of the fact that threads in the same warp can coordinate quickly with each other to avoid this situation. Rather

```
01  __global__ void filter_kernel(unsigned int* input, unsigned int* output,
02      unsigned int N, unsigned int* outputSize) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) {
05          unsigned int val = input[i];
06          if(cond(val)) {
07              unsigned int activeThreads = __activemask();
08              unsigned int j;
09              // Assign a leader thread
10              unsigned int leader = __ffs(activeThreads) - 1;
11              if(laneIdx() == leader) {
12                  // Find how many threads are active
13                  unsigned int numActive = __popc(activeThreads);
14                  // Have the leader perform the atomic operation
15                  cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
16                      outputSize_ref(*outputSize);
17                  j = outputSize_ref.fetch_add(numActive, cuda::memory_order_relaxed);
18              }
19              // Broadcast result to other threads
20              j = __shfl_sync(activeThreads, j, leader);
21              // Find the position of each active thread in the output
22              unsigned int previousThreads = (1 << laneIdx()) - 1;
23              unsigned int previousActiveThreads = activeThreads & previousThreads;
24              unsigned int offset = __popc(previousActiveThreads);
25              // Store the result
26              output[j + offset] = val;
27          }
28      }
29  }
```

FIGURE 12.3

Code for unstable filter kernel with coalesced atomic operations.

than having each thread in the warp independently issue an atomic operation, we can have the threads coordinate to identify the total amount by which they would like to collectively increment the counter, then have one thread in the warp issue a single atomic operation on behalf of the rest. We will refer to the combined atomic operation as a *coalesced atomic operation*. It is similar to how loads and stores by threads in the same warp are coalesced by the hardware (Chapter 5).

The detailed steps for coalescing the atomic operations are as follows. First, we assign a leader thread from among the threads in the warp that are active when the atomic operation is performed. Second, we identify the total amount that needs to be added by all the threads. Third, we have the leader thread perform the atomic operation on the global counter and broadcast the beginning of the acquired space to the other threads. Fourth, each thread identifies its offset within the acquired space where it will store its output key. These steps require the use of special warp-level primitives, including shuffle instructions which were introduced in Chapter 10 as well as other primitives which are introduced in this section.

Fig. 12.3 shows the code for implementing an unstable filter kernel that uses coalesced atomic operations. Compared to the simple implementation in Fig. 12.2, the difference starts on line 07 where the atomic operation is performed. In Fig. 12.3, we start by identifying which threads are active by calling the `__activemask()` primitive (line 07). This primitive returns a 32-bit integer where bit i is set to 1 if the ith thread in the warp is active. For example, if the threads that are active inside the if-statement are threads 1, 2, 4, and 5, then `__activemask()` will return 00000000000000000000000000110110.

Once we know which threads are active, we need to assign a leader thread from among the active threads. A deterministic way of doing so is by picking the active thread with the smallest index within the warp as the leader. In other words, we need to find the index of the least significant bit that is set to 1 in the active mask. To do so, we call the `__ffs()` (Find First Set) intrinsic on the active mask (line 10) which finds the least significant bit that is set to 1 in the mask. We subtract one from the value returned by `__ffs()` because the intrinsic adds a one to the index it returns because the return value of zero is reserved for when none of the bits in the mask are set.

Once we have identified a leader thread, the leader thread needs to find out by how much the threads in the warp collectively want to increment the counter. Since in the original code, each active thread incremented the counter by one, that means that the total amount to increment the counter is equal to the total number of active threads in the warp. This number can be obtained by counting the number of bits that are set to one in the active mask, which is done by calling the `__popc()` (Population Count) intrinsic on the mask (line 13).

Once the leader thread knows the total amount to increment the counter by, it proceeds to increment the global counter by this total amount (lines 15-17). The returned value, which is saved in `j`, marks the beginning index into the output array where the active threads in the warp will place their values. This value is shuffled from the leader thread to all the other active threads in the warp (line 20).

Once each active thread knows the warp's beginning index into the output array, what remains is for each active thread to find its offset from this index. That is, the first active thread will have an offset of 0, the second active thread will have an offset of 1, etc. In other words, a thread's offset is equivalent to the number of active threads that precede it in the warp. Finding this value is equivalent to performing a binary prefix sum operation [1] on the active mask. To find the value, each thread starts by generating a mask that represents the preceding threads in the warp. That is, thread i in the warp generates a mask where bits 0 to $i-1$ are set. This mask can be generated by computing $2^i - 1$ (line 22). After that, the mask of preceding threads is intersected with the mask of active threads to obtain the active preceding threads (line 23). Finally, calling `__popc()` on the mask of active preceding threads will return the number of active preceding threads (line 24), which is the offset we're looking for.

Once the offset is known, the active threads store their filtered values at the starting index plus the thread's offset (line 26). Here, we observe another benefit of using coalesced atomic operations in this context. In addition the reducing the number of atomic operations, using coalesced atomic operations also results in active threads in the warp storing their filtered values at adjacent memory locations. Hence, the stores to global memory are more likely to be coalesced by the hardware.

To coalesce the atomic operations, we have had to perform many operations that involve managing the active threads in a warp. To facilitate the management of active warp threads, the cooperative groups API provides a special kind of group called a coalesced group. A *coalesced group* is a cooperative group that represents the active threads in a warp. Programmers can get a handle on a coalesced group as follows:

```
01  __global__ void filter_kernel(unsigned int* input, unsigned int* output,
02      unsigned int N, unsigned int* outputSize) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) {
05          unsigned int val = input[i];
06          if(cond(val)) {
07              coalesced_group activeThreads = coalesced_threads();
08              unsigned int j;
09              // Assign a leader thread
10              if(activeThreads.thread_rank() == 0) {
11                  // Find how many threads are active
12                  unsigned int numActive = activeThreads.size();
13                  // Have the leader perform the atomic operation
14                  cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
15                      outputSize_ref(*outputSize);
16                  j = outputSize_ref.fetch_add(numActive, cuda::memory_order_relaxed);
17              }
18              // Broadcast result to other threads
19              j = activeThreads.shfl(j, 0);
20              // Find the position of each active thread in the output
21              unsigned int offset = activeThreads.thread_rank();
22              // Store the result
23              output[j + offset] = val;
24          }
25      }
26  }
```

FIGURE 12.4

Code for unstable filter kernel with coalesced atomic operations using cooperative groups.

```
coalesced_group activeThreads = coalesced_threads();
```

Using this handle, it becomes easy to find the number of active threads in the warp and the index of an active thread with respect to other active threads. The number of active threads in the warp is simply the size of the group, i.e., `activeThreads.size()`. The index of an active thread with respect to other active threads is simply the thread's rank in the group, i.e., `activeThreads.thread_rank()`.

Fig. 12.4 shows how the code in Fig. 12.3 can be simplified using the cooperative groups API. Instead of using the `__activemask()` intrinsic function to find the active threads in the warp (line 07 in Fig. 12.3), we declare and initialize a cooperative group (line 07 in Fig. 12.4). Instead of using the `__ffs()` intrinsic function to assign a leader thread (lines 10-11 in Fig. 12.3), we designate the thread whose rank is zero as the leader (line 10 in Fig. 12.4). Instead of using the `__popc()` intrinsic function to find the number of active threads (line 13 in Fig. 12.3), we get the size of the coalesced group (line 12 in Fig. 12.4). The shuffle instruction is also performed using the coalesced group (line 19 in Fig. 12.4). Finally, instead of using bitwise calculations to find the thread's index with respect to other threads in the warp (lines 22-24 in Fig. 12.3), we get the thread's rank in the group (line 21 in Fig. 12.4).

In practice, the compiler already implements the optimization of coalescing atomic operations that we discussed in this section, so it is unnecessary for programmers to apply it manually. Moreover, it is unnecessary to apply the optimization if there is only one thread in the warp that executes the atomic operation, and the compiler indeed refrains from applying the optimization if it can establish that only one thread in the warp is active. Despite programmers not needing to apply this optimization in practice, it is still a good exercise to see how such an optimization would

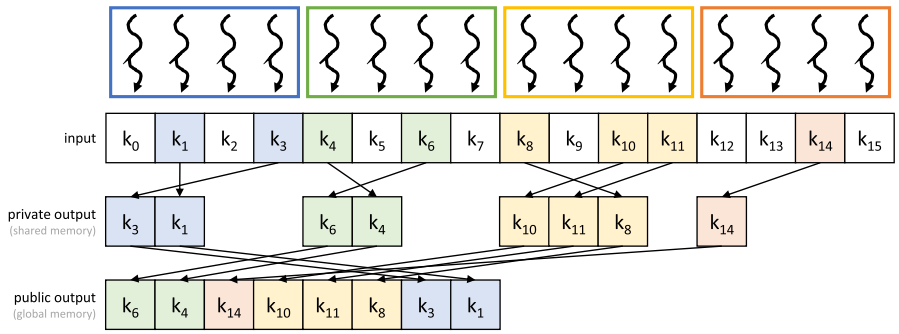


FIGURE 12.5

Unstable filter with privatization.

be implemented because it is a good opportunity to introduce warp voting functions (namely `__activemask()`), intrinsic functions such as `__ffs()` and `__popc()`, and coalesced groups from the cooperative groups API. Other warp voting functions that programmers might encounter include `__all_sync()` which checks if all threads in the warp evaluate some condition as true, `__any_sync()` which checks if some thread in the warp evaluates some condition as true, and `__ballot_sync` which returns a mask of which threads evaluate some condition as true.

12.4 Privatization

We have seen in Section 12.3 how coalesced atomic operations can reduce the contention of the atomic operations on the output list counter. In this section, we look at another optimization for reducing the contention on the counter, namely privatization. Recall from Chapter 6 and Chapter 9 that privatization is an optimization where private versions of the output data are created and updated to reduce contention on the public output data, and then the private versions are merged with the public version at the end. We apply this same optimization to the filter pattern.

Fig. 12.5 illustrates how privatization can be applied in the context of the filter pattern. We create private versions of the output array and the output counter for each thread block in the shared memory. While processing the input keys, threads atomically update their thread block's private counter and place their filtered keys in their thread block's private output array. When all the input keys have been processed, one thread in the thread block atomically updates the public output counter and the threads collaboratively write the contents of the block's private output array to the public output array. The private counters and private output arrays are placed in shared memory to reduce latency and increase throughput for atomically updating the counter and writing to the output private array.

Fig. 12.6 shows the code for implementing an unstable filter kernel with privatization applied. We start by declaring a private output list (line 05) and private counter

```
01  __global__ void filter_kernel(unsigned int* input, unsigned int* output,
02      unsigned int N, unsigned int* outputSize) {
03
04      // Declare and intialize private output list
05      __shared__ unsigned int output_s[BLOCK_DIM];
06      __shared__ unsigned int outputSize_s;
07      if(threadIdx.x == 0) {
08          outputSize_s = 0;
09      }
10      __syncthreads();
11
12      // Filter in the private lists
13      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
14      if(i < N) {
15          unsigned int val = input[i];
16          if(cond(val)) {
17              cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
18                  outputSize_s_ref(outputSize_s);
19              unsigned int j = outputSize_s_ref.fetch_add(1, cuda::memory_order_relaxed);
20              output_s[j] = val;
21          }
22      }
23      __syncthreads();
24
25      // Update the public counter
26      __shared__ unsigned int j;
27      if(threadIdx.x == 0) {
28          cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
29              outputSize_ref(*outputSize);
30          j = outputSize_ref.fetch_add(outputSize_s, cuda::memory_order_relaxed);
31      }
32      __syncthreads();
33
34      // Write to the public list
35      if(threadIdx.x < outputSize_s) {
36          output[j + threadIdx.x] = output_s[threadIdx.x];
37      }
38
39  }
```

FIGURE 12.6

Code for unstable filter kernel with privatization.

(line 06) for each thread block in shared memory. Next, one of the threads in the block initializes the private counter to zero (lines 07-09). The threads then execute a barrier synchronization to ensure that the counter is initialized before any of the threads start updating it (line 10).

The threads then proceed to perform the filter operation, but place the filtered values in the private output list instead of the public output list (lines 13-22). This code is similar to the code in Fig. 12.2, except that the private counter and private output list are used instead of the public ones. Also, the scope of the atomic reference to the counter is `cuda::thread_scope_block` instead of `cuda::thread_scope_device` since the counter in shared memory is only accessed by threads in the same block. Another barrier synchronization is then executed to ensure that all threads finish adding to the private output list before it is committed to the public output list (line 23).

The next step is to reserve space for the private output list in the public output list. Here, one thread in the block atomically updates the public counter, incrementing it by the total number of keys in the private output list (lines 26-31). The beginning index of the reserved space, `j`, is placed in shared memory so that it can be accessed

by the other threads in the block. A barrier synchronization ensures that `j` is written to shared memory before any of the other threads start using it (line 32).

The final step is to write the private output list to the public output list. Consecutive threads will write consecutive keys, hence we use the thread index to read from the private output list and write to the public output list starting at `j` (lines 35-37). Similar to the case of coalesced atomic operation, we observe the additional benefit of privatization besides reduced contention on the public counter. Since consecutive threads write consecutive keys to global memory, the memory stores will be coalesced.

12.5 A simple parallel stable filter

We now turn our attention to parallelizing the *stable* filter operation, where the filtered keys retain the same order in the output list that they had in the input list. The need to retain the order of the keys means that a thread cannot place a filtered key in an arbitrary place in the output list by atomically incrementing a counter. Instead, the thread needs to find out the specific place where the key should go in the output list by finding out how many of the preceding input keys were filtered. Stable filters are used when some select elements are to be deleted or extracted from a sorted list.

Fig. 12.7 illustrates how the stable filter operation can be parallelized. A thread is launched for each key in the input list. Each thread evaluates the condition on its input value and sets a "keep" flag that indicates whether the input value should be kept or not. Next, the thread must identify the offset of its key in the output list by identifying how many preceding keys will be placed in the output list. In other words,

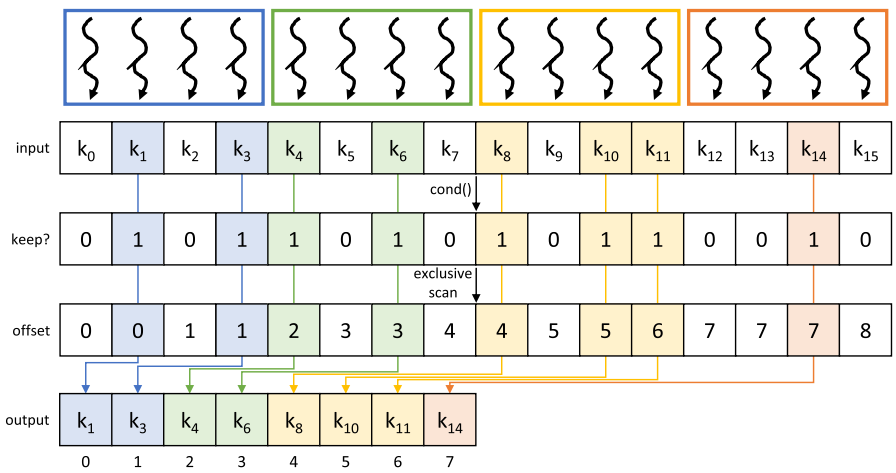


FIGURE 12.7

Parallelizing the stable filter operation.

```
01  __global__ void filter_kernel(unsigned int* input, unsigned int* output,
02    unsigned int N, unsigned int* outputSize) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      unsigned int val = input[i];
05      unsigned int keep = cond(val)?1:0;
06      unsigned int offset = gridExclusiveScan(keep);
07      if(keep) {
08          output[offset] = val;
09      }
10      if(i == N - 1) {
11          *outputSize = offset + keep;
12      }
13  }
```

FIGURE 12.8

Code for a simple stable filter kernel.

the thread needs to find the sum of the "keep" values of all the preceding threads. Finding this sum for all preceding threads amounts to an exclusive scan operation, which we have studied in Chapter 11. We can also optimize this prefix sum operation by leveraging the fact that its input is binary and performing a binary prefix sum [1] on small segments using bit-level operations, as shown in Section 12.3. Once the thread identifies its key's offset in the output list, it proceeds to store the key in the output list.

Fig. 12.8 shows the code for a simple stable filter kernel. First, each thread identifies the index of the key that it is responsible for (line 03) and loads it from global memory (line 04). Next, the thread evaluates the filter condition on the value and sets the integer `keep` to 1 if the condition is true, and 0 otherwise (line 05). After that, the threads in the grid execute a grid-wide exclusive scan on the `keep` value to obtain the offset of the key in the output list (line 06). This grid-wide exclusive scan is performed within a single kernel following the single-kernel scan implementation discussed in Chapter 11. The threads with a key to be kept write their key to the output list at the offset obtained from the scan operation (lines 07-09). Finally, the last thread in the grid writes out the total size of the resulting output list (lines 10-12).

12.6 Improving memory coalescing with shared memory and thread coarsening

One source of inefficiency in the simple stable filter implementation presented in Section 12.5 is that not all threads in the block participate when storing the filtered keys to global memory. This situation misses opportunities for memory coalescing because keys being written to adjacent memory locations in global memory may be written by threads in different warps with many inactive threads between them. These missed opportunities can be recovered by gathering the filtered keys in shared memory, and then writing them from shared memory to global memory as one contiguous chunk. Fig. 12.9 illustrates how this form of privatization can be applied. In this case,

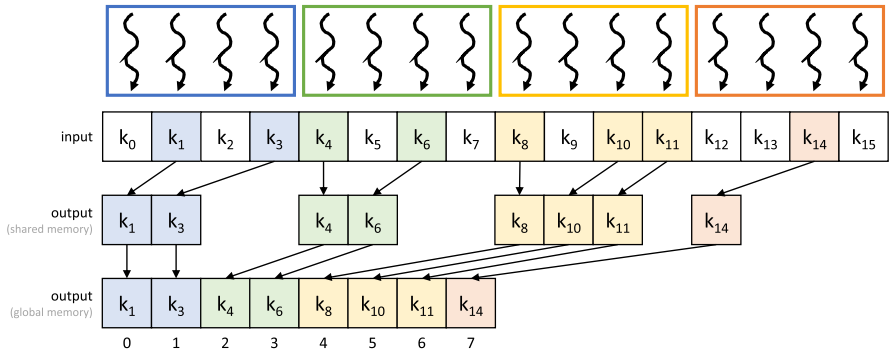


FIGURE 12.9

Using shared memory to improve memory coalescing.

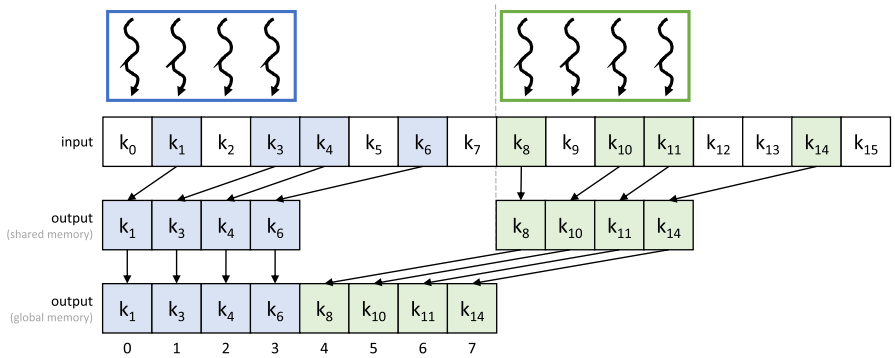


FIGURE 12.10

Thread coarsening to further improve memory coalescing. This toy example assumes that two thread blocks are used and each thread block consists of four threads.

adjacent output keys are written from shared memory to global memory by consecutive threads, increasing the number of adjacent keys written by the same warp, and hence the opportunities for memory coalescing.

To further improve memory coalescing, thread coarsening can also be applied. Fig. 12.10 illustrates how thread coarsening can be applied in the context of stable filter. Each thread block is assigned more keys to filter than the number of threads in the block, which results in fewer and larger contiguous chunks of output keys being written from shared memory to global memory. Having fewer and larger chunks exposes more opportunities for memory coalescing which further improves performance. Here, the price of finer-grain parallelization can be thought of as the need to divide the output lists into smaller chunks which impacts memory coalescing.

Perhaps an even more important benefit of thread coarsening is its impact on the scan operation. The scan operation is the most expensive part of the stable filter

process. Recall from Chapter 11 that scan benefits substantially from thread coarsening because it improves work efficiency and reduces synchronization overhead. By coarsening the filter operation, we naturally coarsen the scan operation within it, which substantially improves the performance of stable filter as a whole.

We leave the detailed stable filter implementation with exclusive scan, privatization, and thread coarsening as an exercise.

12.7 In-place stable filter

In many situations, we may want to perform a filter operation *in place*. In other words, we may want for the output of the filter operation to occupy the same memory array as the input, not a separate array. One situation we may prefer an in-place operation is where the array being filtered is very large relative to the capacity of the GPU's global memory. For example, in our heap object de-allocation example (Section 12.1), there may not be enough space in the global memory to accommodate a new filtered heap. In this case, we may not want to allocate a separate array for storing the output keys.

When performing a filter operation in place, we must be careful not to write any output keys to the array before reading the keys that used to occupy the same memory location. For example, in Fig. 12.10, if the filter operation was done in place, where the input and output arrays are the same, then k_3 would occupy the same location that k_1 used to occupy. For this reason, we must ensure that thread 1 finishes reading k_0 before thread 3 writes k_3. This ordering can easily be enforced by making sure that the thread block executes a barrier synchronization between the step where all the threads read the input values and the step where they write the output values.

A less trivial scenario arises where the threads that read and write in the same location are in different thread blocks. For example, in Fig. 12.10, if the filter operation was done in place, then k_8 would occupy the same location 4 that k_4 used to occupy. However, the threads that read and write these keys are in different thread blocks. Hence, we must ensure that the second block does not write any values before the first block reads its values. In general, we must ensure that no block writes any values before all preceding blocks read their values. It turns out that this ordering is already enforced by the device-wide scan operation. Hence, the code that we have written also works if an in-place filter operation is desired.

To demonstrate that the desired ordering for in-place filter is already enforced, Fig. 12.11 illustrates the dependence between different operations in the stable filter implementation. The desired ordering is that for any pair of blocks i and j such that $j > i$, the read of i happens before the write of j. We can see from the diagram that this is true. The read of i happens before the scan of i, which happens before the scan of j since $j > i$, which happens before the write of j.

We leave it as an exercise for the reader to adapt a single-kernel implementation of exclusive scan into a single kernel implementation of in-place stable filter.

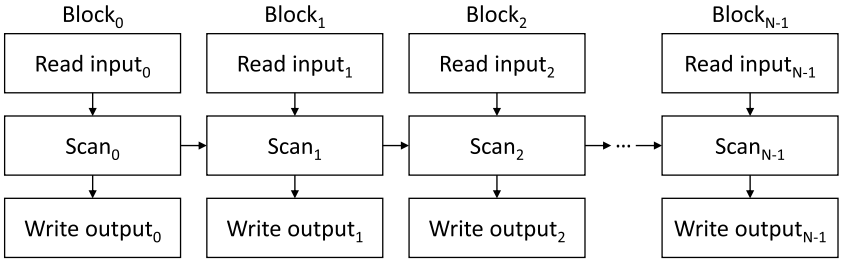


FIGURE 12.11

Dependence diagram for stable filter.

12.8 Related patterns

The stable filter operation is one example of a more general class of patterns where data needs to be moved in a one-directional manner. We review some of these patterns in the section, while noting that many of the optimizations discussed in the context of stable filter are applicable to these related patterns as well.

One similar pattern to stable filter is removing duplicate keys from a sorted list. This pattern can be viewed as a special case of the filter pattern where the condition for the key to be preserved is that the key is not equal to the preceding key.

Another similar pattern is removing rows and/or columns from a matrix. This pattern resembles filtering in that specific keys need to be removed from the input array, however, the difference is that the keys are removed based on their index, not based on their values. Consequently, threads can analytically determine their keys' positions in the output without the need to perform a scan operation. However, if the operation were to be performed in place, we would still need to make sure that earlier thread blocks read their input values before later thread blocks write their output values. In this case, we can use the one-directional synchronization used in the single-pass scan operation just for enforcing an ordering between thread blocks, but without the need to propagate a partial sum.

One more similar pattern is adding rows and/or columns to a matrix. Such pattern often occurs when adjusting the matrix layout to improve memory alignment. Like removing rows/columns, this pattern also determines where keys are to be moved analytically without the need to perform a scan operation. However, one notable difference is that the keys are moved outward in the array, not inward. If the operation were out-of-place, this difference is not significant. However, if the operation were in-place, then this difference affects the direction of synchronization. Since keys are moving outwards, it is now the earlier thread blocks that write to memory locations previously occupied by keys read by earlier thread blocks. Hence, we need to ensure that later thread blocks finish reading their input values before earlier thread blocks write their output values. In this case, the one-directional synchronization is performed in the opposite direction, from the thread block processing the last set of keys to the thread block processing the first set of keys.

Interested readers are referred to the literature for more complex data in-place data movement patterns such as transposition of matrices and tensors [2].

12.9 Summary

In this chapter, we discussed the parallel implementations and optimizations of both stable and unstable filters. The unstable out-of-place filters are based on atomic operations and the optimizations are focused on minimizing the number of atomic operations on the counter and the coalescing of writes to the output. The stable out-of-place filters are based on the scan operation and can be built on the single-pass scan kernel in Chapter 11. In practice, the Thrust [3] and CUB [4] libraries offer filter implementations. Once the reader has mastered the introduction and optimizations of different types of filters, the reader is well equipped to select the Thrust or CUB API functions that are appropriate for the application.

12.10 Exercises

1. Implement a stable filter with exclusive scan as three separate kernels: condition evaluation, exclusive scan, and output generation.
2. Implement stable filter as a single kernel by adding the filtering code into a one-pass scan kernel. What are the major advantages of a single-kernel implementation?
3. Apply privatization to the kernel in Exercise 2.
4. Apply thread coarsening to the kernel in Exercise 3.

References

[1] M. Harris, M. Garland, Optimizing parallel prefix operations for the Fermi architecture, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 29–38.
[2] I.-J. Sung, J. Gómez-Luna, J.M. González-Linares, N. Guil, W.-M.W. Hwu, In-place transposition of rectangular matrices on accelerators, ACM SIGPLAN Notices 49 (8) (2014) 207–218.
[3] N. Bell, J. Hoberock, Thrust: a productivity-oriented library for cuda, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.
[4] NVIDIA, CUB, https://nvidia.github.io/cccl/cub/index.html, 2025.

CHAPTER

13 Merge

And dynamic input data identification

With special contributions from Li-Wen Chang and Jie Lv

CONTENTS

13.1 Background 303
13.2 A sequential merge algorithm 305
13.3 A parallelization approach 306
13.4 Co-rank function implementation 308
13.5 A basic parallel merge kernel 313
13.6 A tiled merge kernel to improve coalescing 314
13.7 A circular-buffer merge kernel 321
13.8 Thread coarsening for merge 327
13.9 Summary 327
13.10 Exercises 327
References 328

Our next parallel pattern is an ordered merge operation, which takes two sorted lists and generates a combined, sorted list. Ordered merge operations can be used as a building block of sorting algorithms as we will see in Chapter 14. Ordered merge operations also form the basis of modern map-reduce frameworks. This chapter presents a parallel ordered merge algorithm where the input data for each thread is dynamically determined. The dynamic nature of the data access makes it challenging to exploit locality and tiling techniques for improved memory access efficiency and performance. The principles behind dynamic input data identification are also relevant to many other important computations, such as set intersection and set union. We present increasingly sophisticated buffer management schemes for achieving increasing levels of memory access efficiency for order merged and other operations that determine their input data dynamically.

13.1 Background

An ordered merge function takes two sorted lists A and B and merges them into a single sorted list C. For this chapter, we assume that the sorted lists are stored in

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00023-8
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

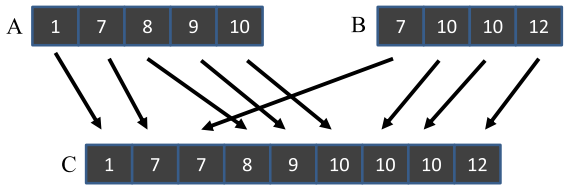


FIGURE 13.1

Example of a merge operation.

arrays. We further assume that each element in such an array has a key. An order relation denoted by $\leq$ is defined on the keys. For example, the keys may be simply integer values and $\leq$ may be defined as the conventional *less than or equal to* relation between these integer values. In the simplest case, the elements consist of just keys.

Suppose that we have two elements e_1 and e_2 whose keys are k_1 and k_2 respectively. In a sorted list based on the relation $\leq$, if e_1 appears before e_2, then $k_1 \leq k_2$. A merge function based on an ordering relation R takes two sorted input arrays A and B having m and n elements respectively, where m and n do not have to be equal. Both array A and array B are sorted based on the ordering relation R. The function produces an output sorted array C having $m+n$ elements. Array C consists of all the input elements from arrays A and B, and is sorted by the ordering relation R.

Fig. 13.1 shows the operation of a simple merge function based on the conventional numerical ordering relation. A has five elements ($m = 5$) and B has four elements ($n = 4$). The merge function generates C with all its 9 elements ($m + n$) from A and B. These elements must be sorted. The arrows in Fig. 13.1 show how elements of A and B should be placed into C to complete the merge operation.

Whenever the numerical values are equal between an element of A and an element of B, the element of A may or may not be required to appear first in the output list C. This requirement is referred to as the *stability* of the ordered merge operation. In general, an ordering operation is stable if elements with equal key values are placed in the same order in the output as they appear in the input. The example in Fig. 13.1 demonstrates stability both within and across the input lists of the merge operation. For example, the two elements whose values are 10 are copied from B into C while maintaining their original order. This illustrates stability within an input list of the merge operation. For another example, the A element whose value is 7 goes into C before the B element of the same value. This illustrates stability across input lists of the merge operation. The stability property allows the ordering operation to preserve previous orderings that are not captured by the key used in the current ordering operation. For example, the lists A and B might have been previously sorted according to a different key before being sorted by the current key to be used for merging. Maintaining stability in the merge operation allows the merge operation to preserve the work done in the previous steps.

The merge operation is the core of merge sort, an important parallelizable sorting algorithm. As we will see in Chapter 14, a parallel merge sort function divides up the

```
01    void merge_sequential(int *A, int m, int *B, int n, int *C) {
02        int i = 0; // Index into A
03        int j = 0; // Index into B
04        int k = 0; // Index into C
05        while ((i < m) && (j < n)) { // Handle start of A[] and B[]
06            if (A[i] <= B[j]) {
07                C[k++] = A[i++];
08            } else {
09                C[k++] = B[j++];
10            }
11        }
12        if (i == m) { // Done with A[], handle remaining B[]
13            while(j < n) {
14                C[k++] = B[j++];
15            }
16        } else { // Done with B[], handle remaining A[]
17            while(i < m) {
18                C[k++] = A[i++];
19            }
20        }
21    }
```

FIGURE 13.2

A sequential merge function.

input list into multiple sections and distributes them to parallel threads. The threads sort the individual section(s) and then cooperatively merge the sorted sections. Such divide-and-concur approach allows efficient parallelization of sorting.

In modern map-reduce distributed computing frameworks such as Hadoop, the computation is distributed to a massive number of compute nodes. The reduce process assembles the result of these compute nodes into the final result. Many applications require that the results be sorted according to an ordering relation. These results are typically assembled using the merge operation in a reduction tree pattern. As a result, efficient merge operations are critical to the efficiency of these frameworks.

13.2 A sequential merge algorithm

The merge operation can be implemented with a straightforward sequential algorithm. Fig. 13.2 shows a sequential merge function. The sequential function in Fig. 13.2 consists of two main parts. The first part consists of a while-loop (line 05) that visits the `A` and `B` list elements in order. The loop starts with the first elements: `A[0]` and `B[0]`. Every iteration fills one position in the output array `C`; either one element of `A` or one element of `B` will be selected for the position (lines 06-10). The loop uses `i` and `j` to identify the `A` and `B` elements currently under consideration; `i` and `j` are both 0 when the execution first enters the loop. The loop further uses `k` to identify the current position to be filled in the output list array `C`. In each iteration, if element `A[i]` is less than or equal to element `B[j]`, the value of `A[i]` is assigned to `C[k]`. In this case, the execution increments both `i` and `k` before going to the next

iteration. Otherwise, the value of `B[j]` is assigned to `C[k]`. In this case, the execution increments both `j` and `k` before going to the next iteration.

The execution exits the while-loop when it reaches either the end of array `A` or the end of array `B`. The execution moves on to the second part, which is on the right Fig. 13.2. If array `A` is the one that has been completely visited, as indicated by the fact that `i` is equal to `m`, then the code copies the remaining elements of array `B` to the remaining positions of array `C` (lines 13 through 15). Otherwise, array `B` is the one that was completely visited, so the code copies the remaining elements of `A` to the remaining positions of `C` (lines 17-19). Note that the if-else construct is unnecessary for correctness. We can simply have the two while-loops (lines 13-15 and 17-19) to follow the first while-loop. Only one of the two while loops will be entered depending on whether `A` or `B` was exhausted by the first while-loop. However, we include the if-else construct to make the code more intuitive for the reader.

We can illustrate the operation of the sequential merge function using the simple example from Fig. 13.1. During the first three (0-2) iterations of the while-loop, $A[0]$, $A[1]$, and $B[0]$ are assigned to $C[0]$, $C[1]$, and $C[2]$. The execution continues until the end of iteration 5. At this point, list A is completely visited and the execution exits the while loop. A total of six C positions have been filled by $A[0]$ through $A[4]$ and $B[0]$. The loop in the true branch of the if-construct is used to copy the remaining B elements – $B[1]$ through $B[3]$ into the remaining C positions.

The sequential merge function visits every input element from both A and B once and writes into each C position once. Its algorithm complexity is $O(m+n)$ and its execution time is linearly proportional to the total number of elements to be merged.

13.3 A parallelization approach

Siebert et al. [1] proposed an approach to parallelizing the merge operation. In their approach, each thread first determines the range of output positions (output range) that it is going to produce, and uses that output range as the input to a *co-rank function* to identify the corresponding input ranges that will be merged to produce the output range. Once the input and output ranges are determined, each thread can independently access its two input sub-arrays and one output sub-array. Such independence allows each thread to perform the sequential merge function on their sub-arrays to do the merge in parallel. It should be clear that the key to the proposed parallelization approach is the co-rank function. We will now formulate the co-rank function.

Let A and B be two input arrays with m and n elements respectively. We assume that both input arrays are sorted according to an ordering relation. The index of each array starts from 0. Let C be the sorted output array generated by merging A and B. Obviously, C has $m+n$ elements. We can make the following observation:

Observation 1. For any k such that $0 \leq k < m+n$, there is either (case 1) an i such that $0 \leq i < m$ and $C[k]$ receives its value from $A[i]$, or (case 2) a j such that $0 \leq j < n$ and $C[k]$ receives its value from $B[j]$ in the merge process.

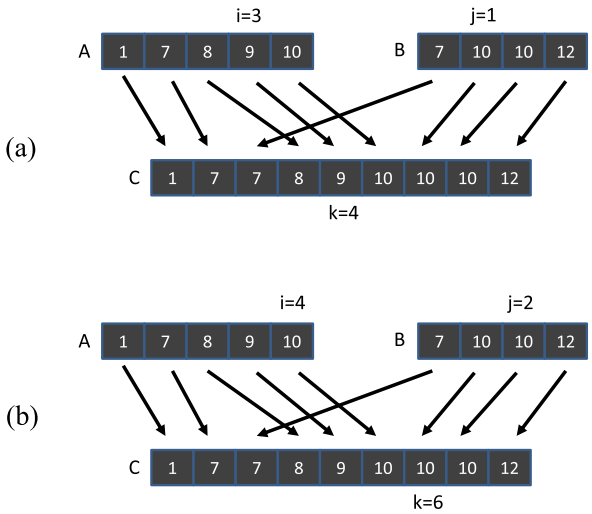


FIGURE 13.3

Examples of Observation 1.

Fig. 13.3 shows the two cases of Observation 1. In the first case, the C element in question comes from array A. For example, in Fig. 13.3(a), $C[4]$ (value 9) receives its values from $A[3]$. In this case, $k = 4$ and $i = 3$. We can see that the prefix subarray of $C[4]$ (the subarray of 4 elements that precedes $C[4]$, i.e., $C[0]$ to $C[3]$) is the result of merging the prefix subarray of $A[3]$ (the subarray of 3 elements that precedes $A[3]$, i.e., $A[0]$ to $A[2]$) and the prefix subarray of $B[1]$ (the subarray of $4 - 3 = 1$ element that precedes $B[1]$. i.e., $B[0]$). The general formula is that subarray $C[0]$ to $C[k - 1]$ (k elements) is the result of merging $A[0]$ to $A[i - 1]$ (i elements) and $B[0]$ to $B[k - i - 1]$ ($k - i$ elements).

In the second case, the C element in question comes from array B. For example, in Fig. 13.3(b), $C[6]$ receives its value from $B[1]$. In this case, $k = 6$ and $j = 1$. The prefix subarray of $C[6]$ (the subarray of 6 elements that precedes $C[6]$, i.e., $C[0]$ to $C[5]$) is the result of merging the prefix subarray of $A[5]$ (the subarray of 5 elements preceding $A[5]$, i.e., $A[0]$ to $A[4]$) and the prefix subarray of $B[1]$ (the subarray of $6 - 5 = 1$ element that precedes $B[1]$. i.e., $B[0]$). The general formula for this case is that subarray $C[0]$ to $C[k - 1]$ (k elements) is the results of merging $A[0]$ to $A[k - j - 1]$ ($k - j$ elements) and $B[0]$ to $B[j - 1]$ (j elements).

In the first case, we find i and derive j as $k - i$. In the second case, we find j and derive i as $k - j$. We can take advantage of the symmetry and summarize the two cases into one observation:

Observation 2. For any k such that $0 \leq k \leq m + n$, we can find i and j such that $k = i + j$, $0 \leq i \leq m$, $0 \leq j \leq n$, and the subarray $C[0]$ to $C[k - 1]$ is the result of merging subarray $A[0]$ to $A[i - 1]$ and subarray $B[0]$ to $B[j - 1]$.

Siebert et al. also proved that i and j, which define the prefix subarrays of A and B, needed to produce the prefix subarray of C of length k, are unique. For an element $C[k]$, the index k is referred to as its rank. The unique indices i and j are referred to as its co-ranks. For example, in Fig. 13.3(a), the rank and co-ranks of $C[4]$ are 4, 3, and 1. For another example, the rank and co-ranks of $C[6]$ are 6, 5, and 1.

The concept of co-rank gives us a path to parallelizing the merge function. We can divide the work among threads by dividing the output array into subarrays and assign the generation of one subarray to each thread. Once the assignment is done, the rank of output elements to be generated by each thread is known. Each thread then uses the co-rank function to determine the two input subarrays that it needs to merge into its output subarray.

Note that the main difference between the parallelization of the merge function and the parallelization of all our previous patterns is that the range of input data to be used by each thread cannot be determined with a simple index calculation. The range of input elements to be used by each thread depends on the actual input values. This makes the parallelized merge operation an interesting and challenging parallel computation pattern.

13.4 Co-rank function implementation

We define the co-rank function as a function that takes the rank (`k`) of an element in an output array `C` and information about the two input arrays `A` and `B`, and returns the co-rank value (`i`) for the corresponding element in the input array `A`. The co-rank function has the following signature:

```
int co_rank(int k, int * A, int m, int * B, int n)
```

where `k` is the rank of the `C` element in question, `A` is a pointer to the first input array, `m` is the size of the `A` array, `B` is a pointer to the second input array, `n` is the size of the input `B` array, and the return value is `i`, the co-rank of `k` in `A`. The caller can then derive `j`, the co-rank value of `k` in `B`, as `k - i`.

Before we study the implementation details of the co-rank function, it is beneficial to first learn about the ways a parallel merge function will use it. Such use of the co-rank function is illustrated in Fig. 13.4 where we use two threads to perform the merge operation. We assume that thread 0 generates $C[0]$ to $C[3]$ and thread 1 generates $C[4]$ to $C[8]$.

Intuitively, each thread calls the co-rank function to derive the beginning positions of the subarrays of A and B that will be merged into the C subarray that is assigned to the thread. For example, thread 1 calls the co-rank function with parameters $(4, A, 5, B, 4)$. The goal of the co-rank function for thread 1 is to identify for its rank value $k_1 = 4$ the co-rank values $i_1 = 3$ and $j_1 = 1$. That is, the subarray starting at $C[4]$ is to be generated by merging the subarrays starting at $A[3]$ and $B[1]$. Intuitively, we are looking for a total of 4 elements from A and B that will fill the first 4 elements of the output array prior to where thread 1 will merge its elements. By

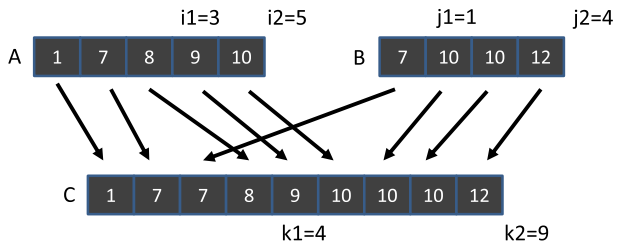


FIGURE 13.4

Example of co-rank function execution.

visual inspection, we see that the choice of $i_1 = 3$ and $j_1 = 1$ meets our need. Thread 0 takes $A[0]$ to $A[2]$ and $B[0]$, leaving out $A[3]$ (value 9) and $B[1]$ (value 10) which is where thread 1 starts merging.

If we changed the value of i_1 to 2, we need to set the j_1 value to 2 so that we can still have a total of 4 elements prior to thread 1. However, this means that we would include $B[1]$ whose value is 10 in thread 0's elements. This value is larger than $A[2]$ (value 8) that would be included in thread 1's elements. Such a change would make the resulting C array not properly sorted. On the other hand, if we changed the value of i_1 to 4, we need to set the j_1 value to 0 to keep the total number of elements at 4. However, this would mean that we include $A[3]$ (value 9) in thread 0's elements, which is larger than $B[0]$ (value 7) which would be incorrectly included in thread 1's elements. These two examples point to a search algorithm that can quickly identify the value.

In addition to identifying where its input segments start, thread 1 also needs to identify where they end. For this reason, thread 1 also calls the co-rank function with parameters $(9, A, 5, B, 4)$. From Fig. 13.4, we see that the co-rank function should produce co-rank values $i_2 = 5$ and $j_2 = 4$. That is, since $C[9]$ is beyond the last element of the C array, all elements of the A and B arrays should have been exhausted if one were trying to generate a C subarray starting at $C[9]$. In general, the input subarrays to be used by thread t are defined by the co-rank values for thread t and thread $t + 1$: $A[i_t]$ to $A[i_t + 1]$ and $B[j_t]$ to $B[j_t + 1]$.

The co-rank function is essentially a search operation. Since both input arrays are sorted, we can use a binary search or even a higher radix search to achieve a computational complexity of $O(log\ N)$ for the search. Fig. 13.5 shows a co-rank function based on binary search. The co-rank function uses two pairs of marker variables to delineate the range of `A` array indices and the range of `B` array indices being considered for the co-rank values. Variables `i` and `j` are the candidate co-rank return values being considered in the current binary search iteration. Variables `i_low` and `j_low` are the smallest possible co-rank values that could be generated by the function. Line 02 initializes `i` to its largest possible value. If the `k` value is greater than `m`, line 02 initializes `i` to `m`, since the co-rank `i` value cannot be larger than the size of the `A` array. Otherwise, line 02 initializes `i` to `k`, since `i` cannot be larger than `k`. The co-rank `j` value is initialized as `k - i` (line 03). Throughout the execution, the co-rank function

```
01    int co_rank(int k, int* A, int m, int* B, int n) {
02        int i = k < m ? k : m; // i = min(k,m)
03        int j = k - i;
04        int i_low = 0 > (k-n) ? 0 : k-n; // i_low = max(0,k-n)
05        int j_low = 0 > (k-m) ? 0 : k-m; // i_low = max(0,k-m)
06        int delta;
07        bool active = true;
08        while(active) {
09            if (i > 0 && j < n && A[i-1] > B[j]) {
10                delta = ((i - i_low +1) >> 1) ; // ceil(i-i_low)/2)
11                j_low = j;
12                j = j + delta;
13                i = i - delta;
14            } else if (j > 0 && i < m && B[j-1] >= A[i]) {
15                delta = ((j - j_low +1) >> 1) ;
16                i_low = i;
17                i = i + delta;
18                j = j - delta;
19            } else {
20                active = false;
21            }
22        }
23        return i;
24    }
```

FIGURE 13.5

A co-rank function based on binary search.

maintains this important invariant relation. The sum of the `i` and `j` variable is always equal to the value of the input variable `k` (the rank value).

The initialization of the `i_low` and `j_low` variables (lines 4 and 5) requires a little more explanation. These variables allow us to limit the scope of the search and make it faster. Functionally, we could set both values to zero and let the rest of the execution elevate them to more accurate values. This indeed makes sense when the `k` value is smaller than `m` and `n`. However, when `k` is larger than `n`, we know that the `i` value cannot be less than `k - n`. The reason is that the greatest number of `C[k]` prefix subarray elements that can come from the `B` array is `n`. Therefore, a minimal of `k - n` elements must come from `A`. Therefore, the `i` value can never be smaller than `k - n`; we may as well set `i_low` to `k - n`. Following the same argument, the `j_low` value cannot be less than `k - m`, which is the least number of elements of `B` that must be used in the merge process and thus the lower bound of the final co-rank `j` value.

We use the example in Fig. 13.6 to illustrate the operation of the co-rank function in Fig. 13.5. The example assumes that three threads are used to merge arrays A and B into C. Each thread is responsible for generating an output subarray of three elements. We will first trace through the binary search steps of the co-rank function for thread 1 which is responsible for generating $C[3]$ to $C[5]$. The reader should be able to determine that thread 1 calls the co-rank function with parameters $(3, A, 5, B, 4)$.

As shown in Fig. 13.5, line 2 of the co-rank function initializes `i` to 3, which is the `k` value since `k` is smaller than `m` (value 5) in this example. Also, `i_low` is set 0. The `i` and `i_low` values define the section of the `A` array that is currently being searched to

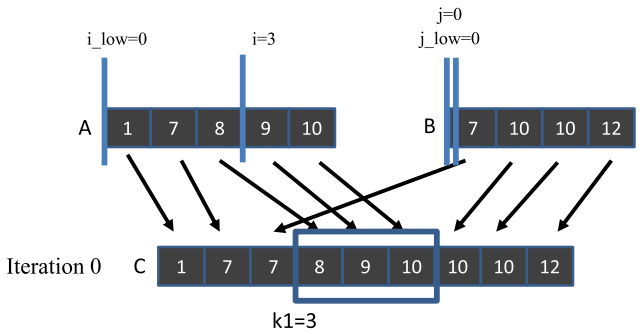


FIGURE 13.6

Iteration 0 of the co-rank function operation example for thread 1.

determine the final co-rank `i` value. Thus, only 0, 1, 2, and 3 are being considered for the co-rank `i` value. Similarly, the `j` and `j_low` values are set to 0 and 0.

The main body of the co-rank function is a while-loop (line 08) that iteratively zooms into the final co-rank `i` and `j` values. The goal is to find a pair of `i` and `j` values that result in `A[i-1] <= B[j]` and `B[j-1] < A[i]`. The intuition is that we choose the `i` and `j` values so none of the values in the `A` subarray used for generating the previous output subarray (referred to as the previous `A` subarray) should be greater than any elements in the `B` subarray used for generating the current output subarray (referred to as the current `B` subarray). Note that the largest `A` element in the previous subarray could be equal to the smallest element in the current `B` subarray since the `A` elements take precedence in placement into the output array whenever a tie occurs between an `A` element and a `B` element because of the stability requirement.

In Fig. 13.5, the first if-construction in the while-loop (line 09) tests if the current `i` value is too high. If so, it will adjust the marker values so that it reduces the search range for `i` by about half towards the smaller end. This is done by reducing the `i` value by about half the difference between `i` and `i_low`.

In Fig. 13.7, for iteration 0 of the while-loop, the if-construct finds that the `i` value (3) is too high since `A[i-1]`, whose value is 8, is greater than `B[j]`, whose value is 7. The next few statements proceed to reduce the search range for `i` by reducing its value by `delta` $= (3 - 0 + 1)/2 = 2$ (lines 10 and 13) while keeping the `i_low` value unchanged. Therefore, the `i_low` and `i` values for the next iteration will be 0 and 1.

The code also makes the search range for `j` to be comparable to that of `i` by shifting it to above the current `j` location. This adjustment maintains the property that the sum of `i` and `j` should be equal to `k`. The adjustment is done by assigning the current `j` value to `j_low` (line 11) and adding the delta value to `j` (line 12). In our example, the `j_low` and `j` values for the next iteration will be 0 and 2.

During iteration 1 of the while-loop, illustrated in Fig. 13.7, the `i` and `j` values are 1 and 2. The if-construct (line 9) finds the `i` value to be acceptable since `A[i-1]` is `A[0]` whose value is 1, while `B[j]` is `B[2]` whose value is 10, so `A[i-1]` is less than `B[j]`. Thus, the condition of the first if-construct fails, and the body of the if-construct is

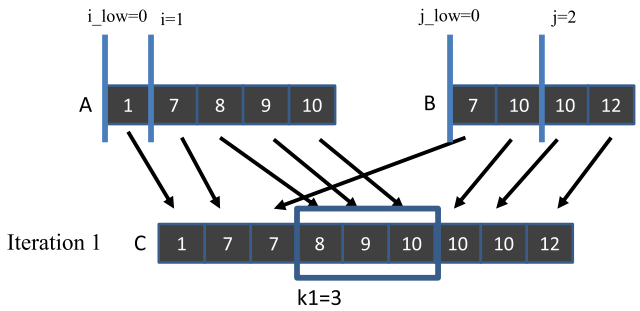


FIGURE 13.7

Iteration 1 of the co-rank function operation example for thread 1.

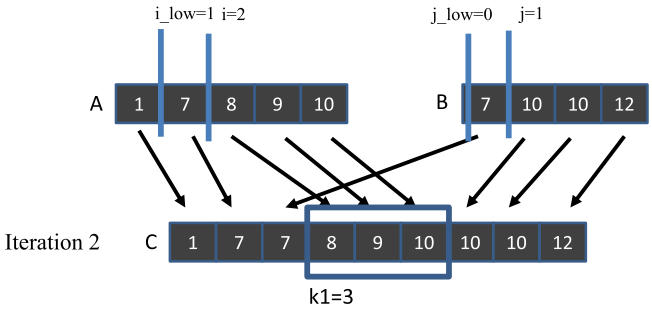


FIGURE 13.8

Iteration 2 of the co-rank function operation example for thread 0.

skipped. However, the `j` value is found to be too high during this iteration since `B[j-1]` is `B[1]` (line 14) whose value is 10, while `A[i]` is `A[1]` whose value is 7. So, the second if-construct will adjust the markers for the next iteration so that the search range for `j` will be reduced by about half towards the lower values. This is done by subtracting `delta = (j-j_low+1) >> 1` $= 1$ from `j` (lines 15 and 18). As a result, the `j_low` and `j` values for the next iteration will be 0 and 1. It also makes the next search range for `i` the same size as that for `j` but shifts it up by `delta` locations. This is done by assigning the current `i` value to `i_low` (line 16) and adding the `delta` value to `i` (line 17). Therefore, the `i_low` and `i` values for the next iteration will be 1 and 2.

During iteration 2, illustrated in Fig. 13.8, the `i` and `j` values are 2 and 1. Both if-constructs (lines 9 and 14) find both `i` and `j` values acceptable. For the first if-construct, `A[i-1]` is `A[1]` (value 7) and `B[j]` is `B[1]` (value 10) so the condition `A[i-1] <= B[j]` is satisfied. For the second if-construct, `B[j-1]` is `B[0]` (value 7) and `A[i]` is `A[2]` (value 8) so the `condition B[j-1] < A[i]` is also satisfied. The co-rank function sets a flag to exit the while-loop (lines 20 and 08) and returns the final `i` value 2 as the co-rank `i` value (line 23). The caller thread can derive the final co-rank `j` value as `k-i` $= 3 - 2 = 1$. An inspection of Fig. 13.8 confirms that co-rank values 2 and 1 indeed identify the correct `A` and `B` input subarrays for thread 1.

```
01  __global__ void merge_basic_kernel(int* A, int m, int* B, int n, int* C) {
02    int tid = blockIdx.x*blockDim.x + threadIdx.x;
03    int elementsPerThread = ceil((m+n)/(blockDim.x*gridDim.x));
04    int k_curr = tid*elementsPerThread; // start output index
05    int k_next = min((tid+1)*elementsPerThread, m+n); // end output index
06    int i_curr = co_rank(k_curr, A, m, B, n);
07    int i_next = co_rank(k_next, A, m, B, n);
08    int j_curr = k_curr - i_curr;
09    int j_next = k_next - i_next;
10    merge_sequential(&A[i_curr], i_next-i_curr, &B[j_curr], j_next-j_curr, &C[k_curr]);
11  }
```

FIGURE 13.9

A basic merge kernel.

The reader should repeat the same process for thread 2 as an exercise. Also, note that if the input streams are much longer, the `delta` values will be reduced by half in each step, so the algorithm is of $log_2 N$ complexity where N is the maximum of the two input array sizes.

13.5 A basic parallel merge kernel

For the rest of this chapter, we assume that the input `A` and `B` arrays reside in the global memory. We further assume that a kernel is launched to merge the two input arrays to produce an output array `C` that is also in the global memory. Fig. 13.9 shows a basic kernel that is a straightforward implementation of the parallel merge function described in Section 13.3.

As we can see, the kernel is simple. It first divides the work among threads by calculating the starting point of the output subarray to be produced by the current thread (`k_curr`) and of the next thread (`k_next`). Keep in mind that the total number of output elements may not be a multiple of the number of threads. Each thread then makes two calls to the co-rank function. The first call uses `k_curr` as the rank parameter which is the first (lowest-indexed) element of the output subarray that the current thread is to generate. The returned co-rank value, `i_curr`, gives the lowest-indexed input `A` array element that belongs to the input subarray to be used by the thread. This co-rank value can also be used to get `j_curr` for the `B` input subarray. The `i_curr` and `j_curr` values mark the beginning of the input subarrays for the thread. Therefore, `&A[i_curr]` and `&B[j_curr]` are the pointers to the beginning of the input subarrays to be used by the current thread.

The second call uses `k_next` as the rank parameter to get the co-rank values for the next thread. These co-rank values mark the positions of the lowest indexed input array elements to be used by the next thread. Therefore, `i_next-i_curr` and `j_next-j_curr` give the sizes of the subarrays of `A` and `B` to be used by the current thread. The pointer to the beginning of the output subarray to be produced by the current thread is `&C[k_curr]`. The final step of the kernel is to call the `merge_sequential` function (from Fig. 13.2) with these parameters.

The execution of the basic merge kernel can be illustrated with the example in Fig. 13.8. The `k_curr` values for the three threads (threads 0, 1, and 2) will be 0, 3, and 6. We will focus on the execution of thread 1 whose `k_curr` value will be 3. The `i_curr` and `j_curr` values determined from the first co-rank function call are 2 and 1. The `k_next` value for thread 1 will be 6. The second call to the co-rank function helps determine the `i_next` and `j_next` values of 5 and 1. Thread 1 then calls the merge function with parameters ($\&A[2], 3, \&B[1], 0, \&C[3]$). Note that the 0 value for parameter `n` indicates that none of the three elements of the output subarray for thread 1 should come from array `B`. This is indeed the case in Fig. 13.8 where output elements $C[3]$ to $C[5]$ all come from $A[2]$ to $A[4]$.

While the basic merge kernel is quite simple and elegant, it falls short in memory access efficiency. First, it is clear that when executing the `merge_sequential` function, adjacent threads in a warp are not accessing adjacent memory locations when they read and write the input and output subarray elements. For the example in Fig. 13.8, during the first iteration of the `merge_sequential` function execution, the three adjacent threads would read $A[0]$, $A[2]$, and $B[0]$. They will then write to $C[0]$, $C[3]$, and $C[6]$. Thus, their memory accesses are not coalesced, resulting in poor utilization of memory bandwidth.

Second, the threads also need to access `A` and `B` elements from the global memory when they execute the co-rank function. Since the co-rank function does a binary search, the access patterns are somewhat irregular and will unlikely be coalesced. As a result, these accesses can further reduce the efficiency of utilizing the memory bandwidth. It would be helpful if we can avoid these uncoalesced accesses to the global memory by the co-rank function.

13.6 A tiled merge kernel to improve coalescing

In Chapter 6, we mentioned three main strategies for improving memory coalescing in kernels: (1) rearranging the mapping of threads to data, (2) rearranging the data itself, and (3) transferring the data between the global memory and the shared memory in a coalesced manner and performing the irregular accesses in the shared memory. For the merge pattern, we will use the third strategy that leverages shared memory to improve coalescing. Using shared memory also has the advantage of capturing the small amount of data reuse across the co-rank functions and the sequential merge phase.

The key observation is that the input `A` and `B` subarrays to be used by the adjacent threads are adjacent to each other in memory. Essentially, all threads in a block will collectively use larger, block-level subarrays of `A` and `B` to generate a larger, block-level subarray of `C`. We can call the co-rank function for the entire block to get the starting and ending locations for the block-level `A` and `B` subarrays. Using these block-level co-rank values, all threads in the block can cooperatively load the elements of the block-level `A` and `B` subarrays into the shared memory in a coalesced pattern.

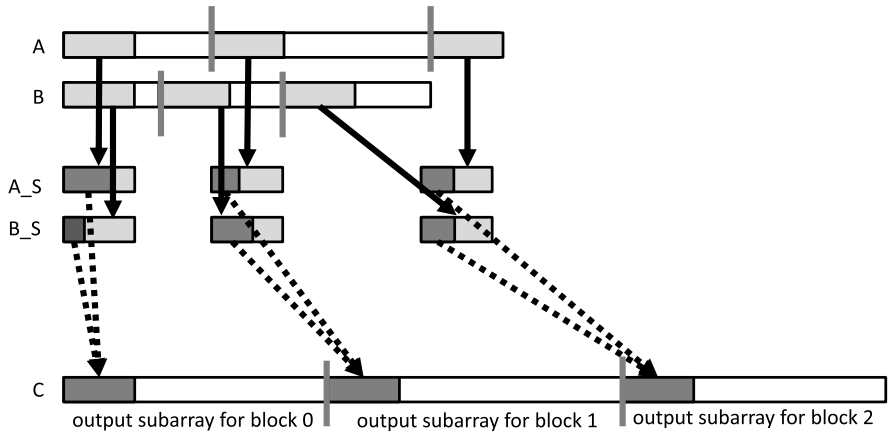


FIGURE 13.10

Design of a tiled merge kernel.

Fig. 13.10 shows the block-level design of a tiled merge kernel. In this example, we assume that three blocks are used for the merge operation. At the bottom of the figure, we show that `C` is partitioned into three block-level subarrays. We delineate these partitions with gray vertical bars. Based on the partition, each block calls the co-rank functions to partition the input array into subarrays to be used for each block. We also delineate the input partitions with gray vertical bars. Note that the input partitions can vary significantly in size according to the actual data element values in the input arrays. For example, in Fig. 13.8, the input `A` subarray is significantly larger than the input `B` subarray for thread 0. On the other hand, the input `A` subarray is significantly smaller than input `B` subarray for thread 1. Obviously, the combined size of the two input subarrays must always be equal to the size of the output subarray for each thread.

We will declare two shared memory arrays `A_S` and `B_S` for each block. Due to the limited shared memory size, `A_S` and `B_S` may not be able to cover the entire input subarrays for the block. Therefore, we will take an iterative approach. Assume that the `A_S` and `B_S` arrays can each hold x elements while each output subarray contains y elements. Each thread block will perform its operation in y/x iterations. During each iteration, all threads in a block will cooperatively load x elements from the block's input `A` subarray and x elements from its input `B` subarray.

The first iteration of each thread is illustrated in Fig. 13.10. We show that for each block, a light gray section of the input `A` subarray is loaded into `A_S` and a light gray section of the input `B` subarray is loaded into `B_S`. With x elements from `A` and x elements from `B` in the shared memory, the thread block has enough input elements to generate at least x output array elements. All threads are guaranteed to have all the input subarray elements they need for the iteration. One might ask why loading a total of $2\times$ input elements can only guarantee the generation of x output elements.

```
01 __global__ void merge_tiled_kernel(int* A,int m, int* B, int n, int* C, int tile_size) {
   /* shared memory allocation */
02    extern __shared__ int shareAB[];
03    int * A_S = &shareAB[0];                      // shareA is first half of shareAB
04    int * B_S = &shareAB[tile_size];              // shareB is second half of shareAB
05    int C_curr = blockIdx.x * ceil((m+n)/gridDim.x); // start point of block's C subarray
06    int C_next = min((blockIdx.x+1) * ceil((m+n)/gridDim.x), (m+n)); // ending point

07    if (threadIdx.x ==0){
08      A_S[0] = co_rank(C_curr, A, m, B, n); // Make block-level co-rank values visible
09      A_S[1] = co_rank(C_next, A, m, B, n); // to other threads in the block
10    }
11    __syncthreads();
12    int A_curr  = A_S[0];
13    int A_next  = A_S[1];
14    int B_curr = C_curr - A_curr;
15    int B_next   = C_next - A_next;
16    __syncthreads();
```

FIGURE 13.11

Part 1 – Identifying block-level output and input subarrays.

The reason is that in the worst case, all elements of the current output section may all come from one of the input sections. This uncertainty of input usage makes the tiling design for the merge kernel much more challenging than the previous patterns. One can be more accurate in loading the input tiles by first calling the co-rank function for the current and next output sections. In this case, we pay an additional binary search operation to save on redundant data loading. We leave this alternative implementation as an exercise. We will also increase the efficiency of memory bandwidth utilization with a circular buffer design in Section 13.7.

Fig. 13.10 also shows that threads in each block will use a portion of the `A_S` and a portion of the `B_S` in each iteration, shown as dark gray sections, to generate a section of x elements in their output `C` subarray. This process is illustrated with the dotted arrows going from the `A_S` and `B_S` dark gray sections to the `C` dark gray sections. Note that each thread block may well use a different portion of its `A_S` and `B_S` sections. Some blocks may use more elements from `A_S` and other may use more from `B_S`. The actual portions used by each block depend on the input data element values.

Fig. 13.11 shows the first part of a tiled merge kernel. A comparison against Fig. 13.9 shows remarkable similarity. This part is essentially the block-level version of the setup code for the thread-level basic merge kernel. Only one thread in the block needs to calculate the co-rank values for the rank values of the beginning output index of the current block and that of the beginning output index of the next block. The values are placed into the shared memory so that they can be visible to all threads in the block. Having only one thread to call the co-rank functions reduces the number of global memory accesses by the co-rank functions and should improve the efficiency of the global memory accesses. A barrier synchronization is used to ensure all threads wait until the block-level co-rank values are available in the shared memory `A_S[0]` and `A_S[1]` locations before they proceed to use the values.

Recall that since the input subarrays may be too large to fit into the shared memory, the kernel takes an iterative approach. The kernel receives a `tile_size` argument

```
17    int counter = 0;                                          //iteration counter
18    int C_length = C_next - C_curr;
19    int A_length = A_next - A_curr;
20    int B_length = B_next - B_curr;
21    int total_iteration = ceil((C_length)/tile_size);         //total iteration
22    int C_completed = 0;
23    int A_consumed = 0;
24    int B_consumed = 0;
25    while(counter < total_iteration){
        /* loading tile-size A and B elements into shared memory */
26        for(int i=0; i<tile_size; i+=blockDim.x){
27            if( i + threadIdx.x < A_length - A_consumed) {
28                A_S[i + threadIdx.x] = A[A_curr + A_consumed + i + threadIdx.x ];
29            }
30        }
31        for(int i=0; i<tile_size; i+=blockDim.x) {
32            if(i + threadIdx.x  < B_length - B_consumed) {
33                B_S[i + threadIdx.x] = B[B_curr + B_consumed + i + threadIdx.x];
34            }
35        }
36        __syncthreads();
```

FIGURE 13.12

Part 2 – loading A and B elements into the shared memory.

that specifies the number of A elements and B elements to be accommodated in the shared memory. For example, a tile_size value 1024 means that 1024 A array elements and 1024 B array elements are to be accommodated in the shared memory. This means that each block will dedicate $(1024 + 1024) \cdot 4 = 8,192$ bytes of shared memory to hold the A and B array elements.

As a simple example, assume that we would like to merge an A array of 33,000 elements ($m = 33,000$) with a B array of 31,000 elements ($n = 31,000$). The total number of output C elements is 64,000. Further assume that we will use 16 blocks (gridDim.x $= 16$) and 128 threads in each block (blockDim.x $= 128$). Each block will generate $64,000/16 = 4000$ output C array elements.

If we assume that the tile_size value is 1024, the while-loop in Fig. 13.12 will need to take four iterations for each block to complete the generation of its 4000 output elements. During iteration 0 of the while-loop, the threads in each block will cooperatively load 1024 elements of A and 1024 elements of B into the shared memory. Since there are 128 threads in a block, they can collectively load 128 elements in each iteration of the for-loop (line 26). So, the first for-loop in Fig. 13.12 will iterate 8 times for all threads in a block to complete the loading of the 1024 A elements. The second for-loop will also iterate 8 times to complete the loading the 1024 B elements. Note that threads use their threadIdx.x values to select the element to load so consecutive threads load consecutive elements. The memory accesses are coalesced. We will come back later and explain the if-conditions and how the index expressions for loading the A and B elements are formulated.

Once the input tiles are in the shared memory, individual threads can divide up the input tiles and merge their portions in parallel. This is done by assigning a section of the output to each thread and running the co-rank function to determine the sections of shared memory data that should be used for generating that output section. The code

```
37        int c_curr  = threadIdx.x   *  (tile_size/blockDim.x);
38        int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);
39        c_curr  = (c_curr <= C_length - C_completed) ? c_curr : C_length - C_completed;
40        c_next = (c_next <= C_length - C_completed) ? c_next : C_length - C_completed;
          /* find co-rank for c_curr and c_next */
41        int a_curr = co_rank(c_curr, A_S, min(tile_size, A_length-A_consumed),
                               B_S, min(tile_size, B_length-B_consumed));
42        int b_curr = c_curr - a_curr;
43        int a_next = co_rank(c_next, A_S, min(tile_size, A_length-A_consumed),
                               B_S, min(tile_size, B_length-B_consumed));
44        int b_next = c_next - a_next;

          /* All threads call the sequential merge function */
45        merge_sequential (A_S+a_curr, a_next-a_curr, B_S+b_curr, b_next-b_curr,
           C+C_curr+C_completed+c_curr);
          /* Update the number of A and B elements that have been consumed thus far */
46        counter ++;
47        C_completed += tile_size;
48        A_consumed += co_rank(tile_size,  A_S, tile_size, B_S, tile_size);
49        B_consumed = C_completed - A_consumed;
50        __syncthreads();
51    }
52  }
```

FIGURE 13.13

Part 3 – all threads merge their individual subarrays in parallel.

in Fig. 13.13 completes this step. Keep in mind that this is a continuation of the while-loop that started in Fig. 13.12. During each iteration of the while-loop, threads in a block will generate a total of `tile_size` elements of `C` using the data we loaded into shared memory. (The exception is the last iteration, which will be addressed later.) The co-rank function is run on the data in shared memory for individual threads. Each thread first calculates the starting position of its output range and that of the next thread, and then uses these starting positions as the inputs to the co-rank function to identify its input ranges. Each thread will then call the sequential merge function to merge its portions of A and `B` elements (identified by the co-rank values) from the shared memory into its designated range of `C` elements.

Let us resume our running example. In each iteration of the while-loop, all threads in a block will be collectively generating 1024 output elements using the two input tiles of `A` and `B` elements in the shared memory. (Once again, we will deal with the last iteration of the while-loop later.) The work is divided among 128 threads so each thread will be generating 8 output elements. While we know that each thread will consume a total of 8 input elements in the shared memory, we need to call the co-rank function to find out the exact number of `A` elements versus `B` elements that each thread will consume and their start and end locations. For example, one thread may use 3 `A` elements and 5 `B` elements, another may use 6 `A` elements and 2 `B` elements, and so on.

Collectively, the total number of `A` elements and `B` elements used by all threads in a block for the iteration will add up to 1024 in our example. For example, if all threads in a block use 476 `A` elements, we know that they also used $1024 - 476 = 548$ `B` elements. It may even be possible that all threads end up using 1024 `A` elements and 0 `B` elements. Keep in mind that a total of 2048 elements are loaded in the shared memory. Therefore, in each iteration of the while-loop, only half of the `A` and `B` elements that were loaded into the shared memory will be used by all the threads in the block.

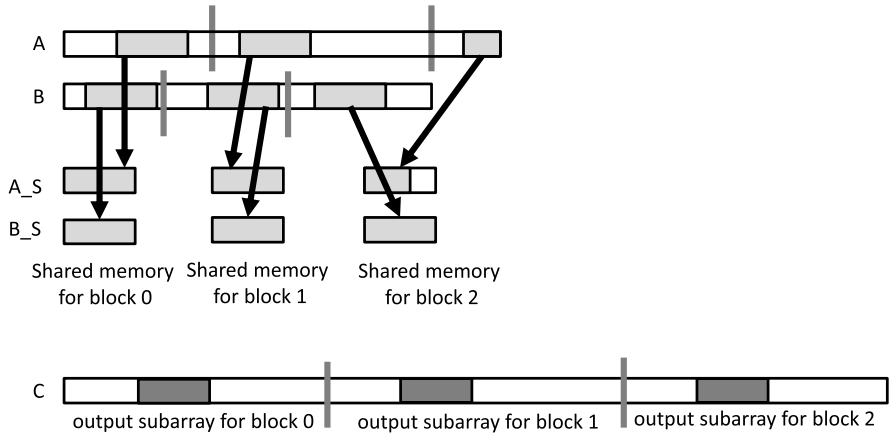


FIGURE 13.14

Iteration 1 of the while-loop in the running example.

We are now ready to examine more details of the kernel function. Recall that we skipped the explanation of the index expressions for loading the `A` and `B` elements from global memory into the shared memory. For each iteration of the while-loop, the starting point for loading the current tile in the `A` and `B` array depends on the total number of `A` and `B` elements that have been consumed by all threads of the block during the previous iterations of the while-loop. Assume that we keep track of the total number of `A` elements consumed by all the previous iterations of the while-loop in variable `A_consumed`. We initialize `A_consumed` to 0 before entering the while-loop. During iteration 0 of the while-loop, all blocks start their tiles from `A[A_curr]` since `A_consumed` is 0 at the beginning of iteration 0. During each subsequent iteration of the while-loop, the tile of `A` elements will start at `A[A_curr+A_consumed]`.

Fig. 13.14 illustrates the index calculation for iteration 1 of the while-loop. In our running example in Fig. 13.10, we show the `A_S` elements that are consumed by the block of threads during iteration 0 as the dark gray portion of the tile in `A_S`. During iteration 1, the tile to be loaded from the global memory for block 0 should start at the location right after the section that contains the `A` elements consumed in iteration 0. In Fig. 13.14, for each block, the section of `A` elements consumed in iteration 0 is shown as the small white section at the beginning of the `A` subarray (marked by the vertical bars) assigned to the block. Since the length of the small section is given by the value of `A_consumed`, the tile to be loaded for iteration 1 of the while-loop starts at `A[A_curr+A_consumed]`. Similarly, the tile to be loaded for iteration 1 of the while-loop starts at `B[B_curr+B_consumed]`.

Note that in Fig. 13.13, `A_consumed` (line 48) and `C_completed` are accumulated through the while-loop iterations. Also, `B_consumed` is derived from the accumulated `A_consumed` and `C_completed` values so it is also accumulated through the while-loop iterations. Therefore, they always reflect the number of `A` and `B` elements consumed

by all the iterations so far. At the beginning of each iteration, the tiles to be loaded for the iteration always start with `A[A_curr+A_consumed]` and `B[B_curr+B_consumed]`.

During the last iterations of the while-loop, there may not be enough input `A` or `B` elements to fill the input tiles in the shared memory for some of the thread blocks. For example, in Fig. 13.14, for thread block 2, the number of remaining `A` elements for iteration 1 is less than the tile size. An if-statement should be used to prevent the threads from attempting to load elements that are outside the input subarrays for the block. The first if-statement in Fig. 13.12 (line 27) detects such attempts by checking if the index of the `A_S` element that a thread is trying to load exceeds the number of remaining `A` elements given by the value of the expression `A_length-A_consumed`. The if-statement ensures that the threads only load the elements that are within the remaining section of the `A` subarray. The same is done for the `B` elements (line 32).

With the if-statements and the index expressions, the tile loading process should work well as long as `A_consumed` and `B_consumed` give the total number of `A` and `B` elements consumed by the thread block in previous iterations of the while-loop. This brings us to the code at the end of the while-loop in Fig. 13.13. These statements update the total number of `C` elements generated by the while-loop iterations thus far. For all but the last iteration, each iteration generates additional `tile_size` elements of `C`.

The next two statements update the total number of `A` and `B` elements consumed by the threads in the block. For all but the last iteration, the number of additional `A` elements consumed by the thread block is the returned value of:

```
co_rank(tile_size, A_S, tile_size, B_S, tile_size)
```

As we mentioned before, the calculation of the number of elements consumed may not be correct at the end of the last iteration of the while-loop. There may not be a full tile of elements left for the final iteration. However, since the while-loop will not iterate any further, the `A_consumed`, `B_consumed`, and `C_completed` values will not be used so the incorrect results will not cause any harm. However, one should remember that if for any reason these values are needed after exiting the while-loop, the three variables will not have the correct values. The values of `A_length`, `B_length`, and `C_length` should be used instead since all the elements in the designated subarrays to the thread block will have been consumed at the exit of the while-loop.

The tiled kernel achieves substantial reduction in global memory accesses by the co-rank function and makes the global memory accesses coalesced. However, as is, the kernel has a significant deficiency. It only makes use of half of the data that is loaded into the shared memory in each iteration. The unused data in the shared memory are simply reloaded in the next iteration. This wastes half of the memory bandwidth. In the next section, we will present a circular buffer scheme for managing the tiles of data elements in the shared memory, which allows the kernel to fully utilize all the `A` and `B` elements loaded into the shared memory. As we will see, this increased efficiency comes with a substantial increase in code complexity.

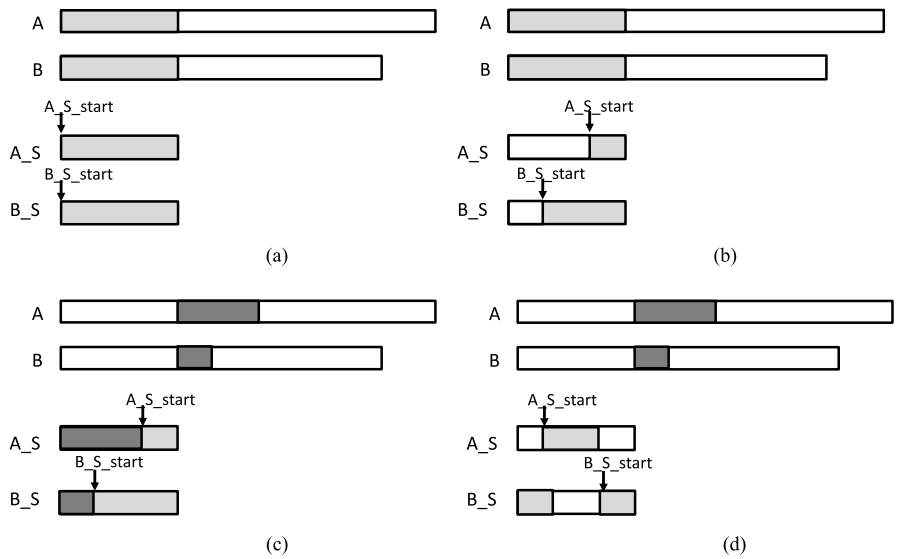


FIGURE 13.15

A circular buffer scheme for managing the shared memory tiles.

13.7 A circular-buffer merge kernel

The design of the circular-buffer merge kernel, which will be referred to as `merge_circular_buffer_kernel`, is largely the same as the `merge_tiled_kernel` kernel in the previous section. The main difference lies in the management of `A` and `B` elements in the shared memory to enable full utilization of all the elements loaded from the global memory. The overall structure of the `merge_tiled_kernel` is shown in Figs. 13.12 through 13.14, which assumes that the tiles of `A` and `B` elements always start at `A_S[0]` and `B_S[0]`. After each while-loop iteration, the kernel loads the next tile starting from `A_S[0]` and `B_S[0]`. The inefficiency of the `merge_tiled_kernel` comes from the fact that part of the next tiles of elements are in the shared memory, but we reload the entire tile from the global memory and write over these remaining elements from the previous iteration.

Fig. 13.15 shows the main idea of `merge_circular_buffer_kernel`. We will continue to use the example from Figs. 13.10 and 13.14. Two additional variables `A_S_start` and `B_S_start` are added to allow each iteration of the while-loop in Fig. 13.12 to start its `A` and `B` tiles at dynamically determined positions inside `A_S[0]` and `B_S[0]`. This added tracking allows each iteration of the while-loop to start the tiles with the remaining `A` and `B` elements from the previous iteration. Since there is no previous iteration when we first enter the while-loop, these two variables are initialized to 0 before entering the while-loop.

During iteration 0, since the values of `A_S_start` and `B_S_start` are both 0, the tiles will start with `A_S[0]` and `B_S[0]`. This is illustrated in Fig. 13.15(a), where we

show the tiles that will be loaded from the global memory (A and B) into the shared memory (A_S and B_S) as light gray sections. Once these tiles are loaded into the shared memory, merge_circular_buffer_kernel will proceed with the merge operation in the same way as the merge_tile_kernel.

We also need to update the A_S_start and B_S_start variables for use in the next iteration by advancing the value of these variables by the number of A and B elements consumed from the shared memory during the current iteration. Keep in mind that the size each buffer is limited to tile_size. At some point, we will need to reuse the buffer locations at the beginning part of the A_S and B_S arrays. This is done by checking if the new A_S_start and B_S_start values exceed the tile_size. If so, we subtract tile_size from them as shown in the if-statement below:

```
A_S_start = (A_S_start + A_S_consumed)%tile_size;
B_S_start = (B_S_start + B_S_consumed)%tile_size;
```

Fig. 13.15(b) illustrates the update of the A_S_start and B_S_start variables. At the end of iteration 0, a portion of the A tile and a portion of the B tile have been consumed. The consumed portions are shown as white sections in A_S and B_S in Fig. 13.15(b). We update the A_S_start and B_S_start values to the position immediately after the consumed sections in the shared memory.

Fig. 13.15(c) illustrates the operations for filling the A and B tiles at the beginning of iteration 1 of the while-loop. A_S_consumed is a variable added to track the number of A elements used in the current iteration. The variable is useful for filling the tile in the next iteration. At the beginning of each iteration, we need to load a section of up to A_S_consumed elements to fill up the A tile in the shared memory. Similarly, we need to load a section of up to B_S_consumed elements to fill up the B tile in the shared memory. The two sections loaded are shown as dark gray sections in Fig. 13.15(c). Note that the tiles effectively "wrap around" in the A_S and B_S arrays since we are reusing the space of the A and B elements that were consumed during iteration 0.

Fig. 13.15(d) illustrates the updates to A_S_start and B_S_start at the end of iteration 1. The sections of elements consumed during iteration 1 are shown as the white sections. Note that in A_S, the consumed section wraps around to the beginning part of A_S. The value of the A_S_start variable is also wrapped around by the % modulo operator. It should be clear that we will need to adjust the code for loading and using the tiled elements to support this circular usage of the A_S and B_S arrays.

Part 1 of merge_circular_buffer_kernel is identical to that of merge_tiled_kernel in Fig. 13.11 so we will not present it. Fig. 13.16 shows Part 2 of the circular buffer kernel. Refer to Fig. 13.12 for variable declarations that remain the same. New variables A_S_start, B_S_start, A_S_consumed, and B_S_consumed are initialized to 0 before we enter the while-loop.

Note that the exit conditions of the two for-loops have been adjusted. Instead of always loading a full tile, as was the case in the merge kernel in Fig. 13.12, each for-loop in Fig. 13.16 is set up to only load the number of elements needed to refill the tiles, given by A_S_consumed. The section of the A elements to be loaded by a thread block in the ith for-loop iteration starts at global memory location

```
25  int A_S_start = 0;
26  int B_S_start = 0;
27  int A_S_consumed = tile_size;  //in the first iteration, fill the tile_size
28  int B_S_consumed = tile_size;  //in the first iteration, fill the tile_size
29  while(counter < total_iteration) {
      /* loading A_S_consumed elements into A_S */
30      for(int i=0; i<A_S_consumed; i+=blockDim.x) {
31        if(i+threadIdx.x < A_length-A_consumed && (i+threadIdx.x) < A_S_consumed) {
32             A_S[(A_S_start + (tile_size - A_S_consumed)+ i + threadIdx.x)%tile_size] =
                 A[A_curr + A_consumed + i + threadIdx.x ];
33        }
34      }
      /* loading B_S_consumed elements into B_S */
35      for(int i=0; i<B_S_consumed; i+=blockDim.x) {
36        if(i+threadIdx.x < B_length-B_consumed && (i+threadIdx.x) < B_S_consumed) {
37           B_S[(B_S_start + (tile_size - A_S_consumed) +i + threadIdx.x)%tile_size] =
                 B[B_curr + B_consumed + i + threadIdx.x];
38        }
39    }
```

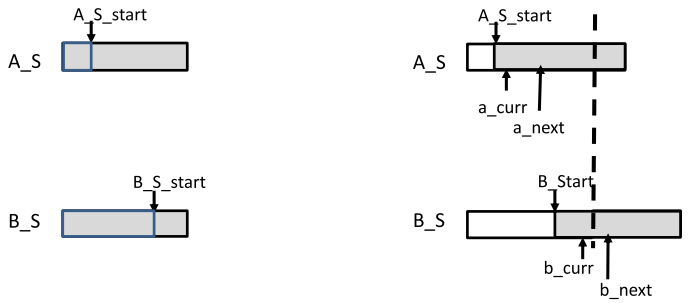
FIGURE 13.16

Part 2 of a circular-buffer merge kernel.

`A[A_curr+A_consumed+i]`. Note that `i` is incremented by `blockDim.x` after each iteration. Thus, the `A` element to be loaded by a thread in the ith for-loop iteration is `A[A_curr+A_consumed+i+threadIdx.x]`. The index for each thread to place its `A` element into the `A_S` array is `A_S_start+(tile_size-A_S_consumed)+i+threadIdx` since the tile starts at `A_S[A_S_start]` and there are `(tile_size-A_S_consumed)` elements remaining in the buffer from the previous iteration of the while-loop. The modulo (%) operation checks if the index value is greater than or equal to `tile_size`. If so, it is wrapped back into the beginning part of the array by subtracting `tile_size` from the index value. The same analysis applies to the for-loop for loading the `B` tile and is left as an exercise.

Using the `A_S` and `B_S` arrays as circular buffers also incurs additional complexity in the implementation of the co-rank and merge functions. Part of the additional complexity could be reflected in the thread-level code that calls these functions. However, in general, it is better if one can efficiently handle the complexities inside the library functions to minimize the increased level of complexity in the user code. We show such an approach in Fig. 13.17. Fig. 13.17(a) shows the implementation of the circular buffer. `A_S_start` and `B_S_start` mark the beginning of the tile in the circular buffer. The tiles wrap around in the `A_S` and `B_S` arrays, shown as the light gray section to the left of `A_S_start` and `B_S_start`.

Keep in mind that the co-rank values are used for threads to identify the starting position, ending position, and length of the input subarrays that they are to use. When we employ circular buffers, we could provide the co-rank values as the actual indices in the circular buffer. However, this would incur quite a bit of complexity in the `merge_circular_buffer_kernel` code. For example, the `a_next` value could be smaller than the `A_curr` value since the tile is wrapped around in the `A_S` array. Thus, one would need to test for the case and calculate the length of the section as `a_next-A_curr+tile_size`. However, in other cases when `a_next` is larger than `A_curr`, the length of the section is simply `a_next-A_curr`.



(a) reality (b) simplified

FIGURE 13.17

A simplified model for the co-rank values when using a circular buffer.

Fig. 13.17(b) shows a simplified model for defining, deriving, and using the co-rank values with the circular buffer. In this model, each tile appears to be in a continuous section starting at `A_S_start` and `B_S_start`. In the case of the `B_S` tile in Fig. 13.17(a), `b_next` is wrapped around and would be smaller than `B_curr` in the circular buffer. However, as shown in Fig. 13.17(b), the simplified model provides the illusion that all elements are in a continuous section of up to `tile_size` elements and thus `a_next` is always larger than or equal to `A_curr` and `b_next` is always larger than or equal to `B_curr`. It is up to the implementation of the `co_rank_circular` and `merge_sequential_circular` functions to map this simplified view of the co-rank values into the actual circular buffer indices so that they can carry out their functionalities correctly and efficiently.

The `co_rank_circular` and `merge_sequential_circular` functions have the same set of parameters as the original `co_rank` and merge functions plus three additional ones: `A_S_start`, `B_S_start`, and `tile_size`. These three additional parameters inform the functions where the current starting points of the buffers are and how big the buffers are. Fig. 13.18 shows the revised thread-level code based on the simplified model for the co-rank value using circular buffers. The only change to the code is that the `co_rank_circular` and `merge_sequential_circular` functions are called instead of the `co_rank` and merge functions. This demonstrates that a well-designed library interface can reduce the impact on the user code when employing sophisticated data structures.

Fig. 13.19 shows an implementation of the co-rank function that provides the simplified model for the co-rank values while correctly operating on circular buffers. It treats `i`, `j`, `i_low`, and `j_low` values in exactly the same way as the co-rank function in Fig. 13.5. The only change is that `i`, `i-1`, `j`, and `j-1` are no longer used directly as indices when accessing the `A_S` and `B_S` arrays. They are used as offsets that are to be added to the values of `A_S_start` and `B_S_start` to form the index values `i_cir`, `i_m_1_cir`, `j_cir`, and `j_m_1_cir`. In each case, we need to test if the actual index values need to be wrapped around to the beginning part of the buffer. Note that we cannot simply use `i_cir-1` to replace `i-1`. We need to form the final index value and

```
40        int c_curr  = threadIdx.x   *  (tile_size/blockDim.x);
41        int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);

42        c_curr  = (c_curr <= C_length-C_completed) ? c_curr : C_length-C_completed;
43        c_next = (c_next <= C_length-C_completed) ? c_next : C_length-C_completed;
        /* find co-rank for c_curr and c_next */
44        int a_curr = co_rank_circular(c_curr,
                        A_S, min(tile_size, A_length-A_consumed),
                        B_S, min(tile_size, B_length-B_consumed),
                        A_S_start, B_S_start, tile_size);
45        int b_curr = c_curr - a_curr;
46        int a_next = co_rank_circular(c_next,
                        A_S, min(tile_size, A_length-A_consumed),
                        B_S, min(tile_size, B_length-B_consumed),
                        A_S_start, B_S_start, tile_size);

47        int b_next = c_next - a_next;
        /* All threads call the circular-buffer version of the sequential merge function */
48        merge_sequetial_circular( A_S, a_next-a_curr,
                        B_S, b_next-b_curr,  C+C_curr+C_completed+c_curr,
                        A_S_start+a_curr, B_S_start+b_curr, tile_size);

        /* Figure out the work has been done */
49        counter ++;
50        A_S_consumed = co_rank_circular(min(tile_size,C_length-C_completed),
                        A_S, min(tile_size, A_length-A_consumed),
                        B_S, min(tile_size, B_length-B_consumed),
                        A_S_start, B_S_start, tile_size);

51        B_S_consumed = min(tile_size, C_length-C_completed) - A_S_consumed;
52        A_consumed += A_S_consumed;
53        C_completed += min(tile_size, C_length-C_completed);
54        B_consumed = C_completed - A_consumed;

55        A_S_start = (A_S_start + A_S_consumed) % tile_size;
56        B_S_start = (B_S_start + B_S_consumed) % tile_size;
57        __syncthreads();
58    }
59  }
```

FIGURE 13.18

Part 3 of a circular-buffer merge kernel.

check for the need to wrap it around. It should be clear that the simplified model also helps to keep the co-rank function code simple: all the manipulations of the `i`, `j`, `i_low`, and `j_low` values remain the same; they do not need to deal with the circular nature of the buffers.

Fig. 13.20 shows an implementation of the `merge_sequential_circular` function. Similar to the `co_rank_circular` function, the logic of the code remains essentially unchanged from the original merge function. The only change is in the way `i` and `j` are used to access the `A` and `B` elements. Since the `merge_sequential_circular` function will only be called by the thread-level code of `merge_circular_buffer_kernel`, the `A` and `B` elements accessed will be in the `A_S` and `B_S` arrays. In all four places where `i` or `j` is used to access the `A` or `B` elements, we need to form `i_cir` or `j_cir` and test if the index value needs to be wrapped around to the beginning part of the array. Otherwise, the code is the same as the merge function in Fig. 13.2.

Although we did not list all parts of `merge_circular_buffer_kernel`, the reader should be able to put it all together based on the parts that we discussed. The use of tiling and circular buffers adds quite a bit of complexity. In particular, each thread uses quite a few more registers to keep track of the starting point and remaining number of elements in the buffers. All these additional usages can potentially reduce the

```
int co_rank_circular(int k, int* A, int m, int* B, int n, int A_S_start,
                     int B_S_start, int tile_size) {
    int i= k < m ? k : m;  // i = min(k,m)
    int j = k- i;
    int i_low = 0 > (k-n) ? 0 : k-n; // i_low = max(0, k-n)
    int j_low = 0 > (k-m) ? 0 : k-m; // i_low = max(0,k-m)
    int delta;
    bool active = true;
    while(active) {
        int i_cir = (A_S_start+i) % tile_size;
        int i_m_1_cir = (A_S_start+i-1) % tile_size);
        int j_cir = (B_S_start+j) % tile_size);
        int j_m_1_cir = (B_S_start+i-1) % tile_size);
        if (i > 0 && j < n && A[i_m_1_cir] > B[j_cir]) {
            delta = ((i - i_low +1) >> 1) ; // ceil(i-i_low)/2)
            j_low = j;
            i = i - delta;
            j = j + delta;
        } else if (j > 0 && i < m && B[j_m_1_cir] >= A[i_cir]) {
            delta = ((j - j_low +1) >> 1) ;
            i_low = i;
            i = i + delta;
            j = j - delta;
        } else {
            active = false;
        }
    }
    return i;
}
```

FIGURE 13.19

A `co_rank_circular` function that operates on circular buffers.

```
void merge_sequential_circular(int *A, int m, int *B, int n, int *C, int A_S_start,
                               int B_S_start, int tile_size) {
    int i = 0;  //virtual index into A
    int j = 0;  //virtual index into B
    int k = 0; //virtual index into C
    while ((i < m) && (j < n)) {
        int i_cir = (A_S_start + i) % tile_size;
        int j_cir = (B_S_start + j) % tile_size;
        if (A[i_cir] <= B[j_cir]) {
            C[k++] = A[i_cir]; i++;
        } else {
            C[k++] = B[j_cir]; j++;
        }
    }
    if (i == m) { //done with A[] handle remaining B[]
        for (; j < n; j++) {
            int j_cir = (B_S_start + j) % tile_size;
            C[k++] = B[j_cir];
        }
    } else { //done with B[], handle remaining A[]
        for (; i <m; i++) {
            int i_cir = (A_S_start + i) % tile_size);
            C[k++] = A[i_cir];
        }
    }
}
```

FIGURE 13.20

Implementation of the `merge_sequential_circular` function.

occupancy, or the number of thread-blocks that can be assigned to each of the streaming multiprocessors when the kernel is executed. However, since the merge operation is memory bandwidth bound, the computational and register resources are likely underutilized. Thus, increasing the number of registers used and address calculations to conserve memory bandwidth are a reasonable tradeoff.

13.8 Thread coarsening for merge

The overhead of parallelizing merge across many threads is primarily the fact that each thread has to perform its own binary search operations to identify the co-ranks of its output indices. The number of binary search operations performed can be reduced by reducing the number of threads launched, which can be done by assigning more output elements per thread. All the kernels presented in this chapter already have thread coarsening applied because they are all written to processes multiple elements per thread.

In a completely uncoarsened kernel, each thread would be responsible for a single output element. However, this would require a binary search operation to be performed for every single element which would be expensive. Hence, coarsening is essential for amortizing the cost of the binary search operation across a substantial number of elements.

13.9 Summary

In this chapter, we introduced the ordered merge pattern whose parallelization requires each thread to dynamically identify its input position ranges. Because the input ranges are data dependent, we resort to a fast search implementation of the co-rank function to identify the input range for each thread. The fact that the input ranges are data dependent also creates extra challenges when we use tiling technique to conserve memory bandwidth and enable memory coalescing. As a result, we introduce the use of circular buffers to allow us to make full use of the data loaded from global memory. We showed that introducing a more complex data structure such as a circular buffer can significantly increase the complexity of the code that uses the data structure. Thus, we introduce a simplified buffer access model for the code that manipulates and uses the indices to remain largely unchanged. The actual circular nature of the buffers is only exposed when these indices are used to access the elements in the buffer.

13.10 Exercises

1. Assume that we need to merge two lists $A = \{1, 7, 8, 9, 10\}$ and $B = \{7, 10, 10, 12\}$. What are the co-rank values for $C[8]$?

2. Complete the calculation of co-rank functions for Thread 2 in Fig. 13.6.
3. For the for-loops that load the `A` and `B` tiles in Fig. 13.12, add a call to the co-rank function so that we can load only the `A` and `B` elements that will be consumed in the current iteration of the while-loop.
4. Consider a parallel merge of two arrays of size 1,030,400 and 608,000. Assume that each thread merges 8 elements, and that a thread block size of 1024 is used.
   a. In the basic merge kernel in Fig. 13.9, how many threads perform binary search on data in global memory?
   b. In the tiled merge kernel in Figs. 13.11-13.13, how many threads perform binary search on data in global memory?
   c. In the tiled merge kernel in Figs. 13.11-13.13, how many threads perform binary search on data in shared memory?

References

[1] C. Siebert, J.L. Träff, Efficient mpi implementation of a parallel, stable merge algorithm, in: European MPI Users' Group Meeting, Springer, 2012, pp. 204–213.

CHAPTER

14 Sorting

With special contributions from Michael Garland

CONTENTS

14.1 Background 330
14.2 Parallel odd-even sort 331
14.3 Parallel merge sort 333
14.4 Radix sort 334
14.5 Parallel radix sort 336
14.6 Optimizing for memory coalescing 339
14.7 Choice of radix value 342
14.8 Thread coarsening to improve coalescing 344
14.9 Other parallel sort methods 345
14.10 Summary 346
14.11 Exercises 347
References 347

Sorting algorithms place the data elements of a list into a certain order. Sorting is foundational to modern data and information services since the computational complexity of retrieving information from data sets can be significantly reduced if the data set is in proper order. For example, sorting is often used to canonicalize the data for fast comparison and reconciliation between data lists. Also, the efficiency of many data processing algorithms can be improved if the data is in a certain order. Due to their importance, efficient sorting algorithms have been the subject of many computer science research efforts. Even with these efficient algorithms, sorting large data lists is still time consuming and can benefit from parallel execution. Parallelizing efficient sorting algorithms is challenging and sometimes requires elaborate designs. This chapter presents the designs of three parallel sorting algorithms – odd-even sort, merge sort, and radix sort. Odd-even sort is covered briefly due to its simplicity and merge sort is covered briefly because it largely depends on the merge pattern already presented in Chapter 13. Most of the chapter is dedicated to radix sort and its optimizations, which rely on a more general form of the stable filter pattern which will be discussed further in Chapter 12. Other popular parallel sorting algorithms such as transposition sort and sampling sort are also discussed.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00024-X
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

14.1 Background

Sorting is one of the earliest applications for computers. A sorting algorithm arranges the elements of a list into a certain order. The order to be enforced by a sorting algorithm depends on the nature of these elements. Examples of popular orders are numerical order for numbers and lexicographical order for text strings. More formally, any sorting algorithm must satisfy the following two conditions:

- The output is in either nondecreasing or nonincreasing order. For nondecreasing order, each element is no smaller than the previous element according to the desired order. For nonincreasing order, each element is no larger than the previous element according to the desired order.
- The output is a permutation of the input. That is, the algorithm must retain all of the original input elements while reordering them into the output.

In its simplest form, the elements of a list can be sorted according to the values of each element. For example, the list [5, 2, 7, 1, 3, 2, 8] can be sorted into a nondecreasing order output [1, 2, 2, 3, 5, 7, 8]. A more complex and common use case is that each element consists of a key field and a value field and the list should be sorted based on the key field. For example, assume that each element is a tuple (age, income in thousands of dollars). The list [(30,150), (32,80), (22,45), (29,80)] can be sorted using the income as the key field into a nonincreasing order [(30,150), (32,80), (29,80), (22,45)]. If the elements of a list are simply values, these values serve as keys for sorting purposes.

Sorting algorithms can be classified into stable and unstable algorithms. A stable sort algorithm preserves the original order of appearance when two elements have equal key value. For example, when sorting the list [(30,150), (32,80), (22,45), (29,80)] into a nonincreasing order using income as the key field, a stable sorting algorithm must guarantee that (32, 80) appears before (29,80) because the former appear before the latter in the original input. On the other hand, an unstable sorting algorithm does not offer such a guarantee. Stable algorithms are required if one wishes to use multiple keys to sort a list in a cascaded manner. For example, if each element has a primary key and a secondary key. With stable sorting algorithms, one can first sort the list according to the secondary key and then sort one more time with the primary key. The second sort will preserve the order produced by the first sort.

Sorting algorithms can also be classified into comparison-based and non-comparison-based algorithms. Comparison-based sorting algorithms cannot achieve better than $O(N \cdot logN)$ complexity when sorting a list of N elements because they must perform a minimal number of comparisons among the elements. In contrast, some of the non-comparison-based algorithms can achieve better than $O(N \cdot logN)$ complexity, but they may not generalize to arbitrary types of keys. Both comparison-based and non-comparison-based sorting algorithms can be parallelized. In this chapter, we present two parallel comparison-based sorting algorithms (odd-even sort and merge sort) as well as a parallel non-comparison-based sorting algorithm (radix sort).

Due to the importance of sorting, the computer science research community has produced a great spectrum of sorting algorithms based on a rich variety of data structures and algorithmic strategies. As a result, introductory computer science classes often use sorting algorithms to illustrate a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst, and average case analysis, time–space tradeoffs, and upper and lower bounds. In this chapter, we continue this tradition and use three sorting algorithms to illustrate several important parallelization and performance optimization techniques [1].

14.2 Parallel odd-even sort

One of the simplest comparison-based sequential sorting algorithms is bubble sort. The high-level strategy of bubble sort is to compare adjacent pairs of list elements and swap them if they are out of order. This process is repeated until the entire list is sorted. In bubble sort, each iteration goes through the list of elements sequentially from beginning to end, sorting adjacent pairs. However, the order in which the pairs are sorted is not necessary for the strategy to work, as long as all the pairs are covered in each iteration. A parallel sorting algorithm could follow a similar iterative strategy, but have each iteration sort adjacent pairs in parallel rather than in sequential order.

The challenge with sorting pairs in parallel is that the pairs being sorted must be independent in order to avoid race conditions. For example, if one thread is assigned to sort the list elements at indices 0 and 1, we cannot assign another thread to sort the list elements at indices 1 and 2. If we do so and both threads decide to swap their pairs simultaneously, they would encounter a race condition when accessing the element at index 1. To maximize the number of independent pairs that are sorted simultaneously, we can sort all the pairs whose first elements have odd indices (odd pairs) in parallel, then sort all the pairs whose first element have even indices (even pairs) in parallel. This technique is the basis for *odd-even transposition sort*, or *odd-even sort* for short.

Odd-even sort is a comparison-based parallel sorting algorithm that iteratively alternates between odd and even pairs of adjacent list elements. In each iteration, the odd (or even) pairs of adjacent list elements are compared and swapped if they are out of order. Fig. 14.1 shows an example of how odd-even sort can be applied to a list of eight elements. In the first iteration, four threads are launched, one for each even pair. The first two threads find their pairs already in order, while the next two threads find their pairs out of order and swap the elements. In the second iteration, one thread is launched for each odd pair. Again, some threads find their pairs already in order, while others find their pairs out of order and thus swap the elements. In the third iteration, we go back to sorting the even pairs. We keep alternating between odd and even pairs until no elements are swapped, which indicates that the list is sorted.

Fig. 14.2 shows a kernel for implementing odd-even sort. The kernel is called once for each iteration with half as many threads as there are elements in the list. The `isOddStep` parameter (line 02) indicates whether the kernel is sorting odd pairs or

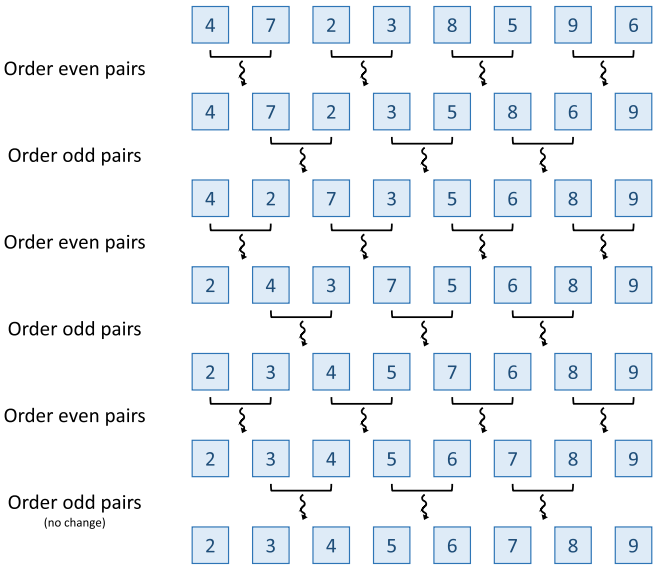


FIGURE 14.1

Parallel odd-even sort.

```
01    __global__ void sort_kernel(unsigned int* data, unsigned int* hasChanged,
02                                unsigned int N, unsigned int isOddStep) {
03        unsigned int i = 2*(blockIdx.x*blockDim.x + threadIdx.x)
04                           + (isOddStep ? 1 : 0);
05        if(i < N - 1) {
06            if(data[i] > data[i + 1]) {
07                unsigned int tmp = data[i];
08                data[i] = data[i + 1];
09                data[i + 1] = tmp;
10                *hasChanged = 1;
11            }
12        }
13    }
```

FIGURE 14.2

Parallel odd-even sort kernel.

even pairs. The kernel starts by identifying the index of the first element in the pair that the thread is responsible for (lines 03-04). For even pairs (`isOddStep` $= 0$), the index is just the thread's global index multiplied by two such that thread t would sort the pair with indices $(2 \cdot t, 2 \cdot t + 1)$. For odd pairs (`isOddStep` $= 1$), we further add 1 such that thread t would sort the pair with indices $(2 \cdot t + 1, 2 \cdot t + 2)$. After the boundary check (line 05), each thread compares the two list elements it is responsible for (line 06) and swaps them if they are out of order (lines 07-09). Finally, if a thread is successful at swapping a pair, it sets a global flag `hasChanged` to indicate that the list order has changed and that another iteration needs to be performed (line 10).

Note that when multiple threads write the value 1 to `hasChanged`, the situation is technically a race condition: multiple threads access the same memory location without ordering and at least one of the accesses is a write. In practice, such race conditions are benign because all threads write the same value so the final outcome is the same regardless of how many threads perform the write. This property of an operation having the same effect regardless of how many times it is applied is referred to as *idempotence*. However, in principle, in the C++ memory model, race conditions can cause undefined behavior even if the operation is idempotent. So although this code may work in practice, it violates the C++ memory model and is not guaranteed to work. Accordingly, conservative programmers should use an atomic operation to write 1 to `hasChanged`.

While odd-even sort is simple to implement, it has a high time and work complexity. The number of iterations needed is $O(N)$, where N is the size of the input list, because in the worst case, the largest element is at the beginning of the list and needs N iterations to reach the end. If there are sufficient execution resources to execute each iteration completely in parallel, then the time complexity would be $O(N)$. In each of the $O(N)$ iterations, we perform $N/2$ or $O(N)$ comparisons. Hence, the work complexity of odd-even sort is $O(N^2)$.

14.3 Parallel merge sort

The high time and work complexity of odd-even sort motivates the need to have more efficient comparison-based parallel sorting algorithms with lower complexity. One such comparison-based sorting algorithm that is amenable to parallelization is merge sort. Merge sort works by dividing the input list into segments, sorting each segment (using merge sort or another sorting algorithm), then performing an ordered merge of the sorted segments.

Fig. 14.3 shows one way for parallelizing merge sort. Initially, the input list is divided into many segments that are each sorted independently using some sorting algorithm. After that, every pair of segments is merged into a single segment. This process is repeated until all the keys become part of the same segment. At each stage, the computation can be parallelized by performing different merge operations in parallel as well as extracting parallelism from within each merge operations. We have already seen how to parallelize a merge operation in Chapter 13.

As the stages of the parallel merge sort in Fig. 14.3 progress, there is a trade-off between the parallelism that can be extracted across and within merge operations. In the earlier stages, there are more independent merge operations that can be performed in parallel. In the later stages, there are fewer independent merge operations, however, each merge operation merges more keys exposing more parallelism within the merge operation. For example, the first merge stage consists of four independent merge operations. Hence, our grid of eight thread blocks may assign two thread blocks to process each merge operation in parallel. In the next stage, there are only two merge operations, however, each operation merges twice the number of keys.

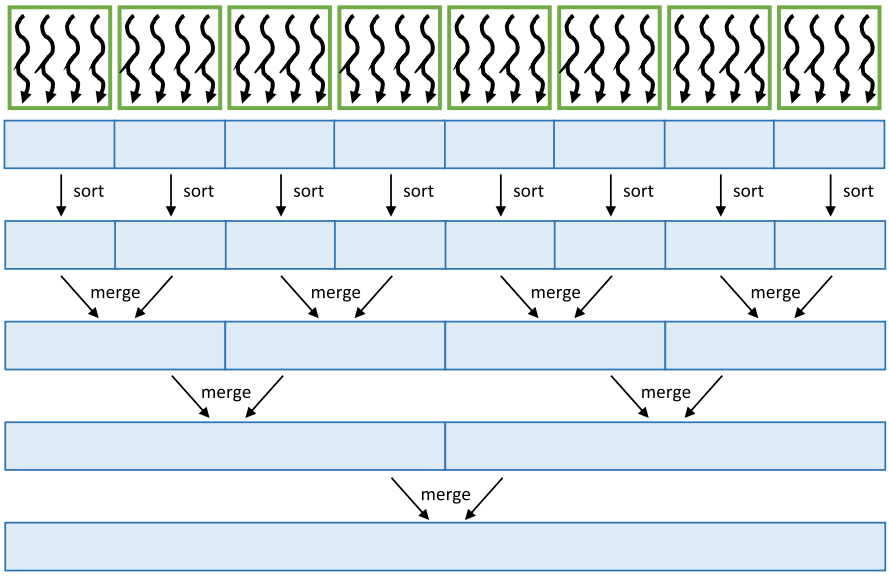


FIGURE 14.3

One way for parallelizing merge sort.

Hence, our grid of eight thread blocks may assign four thread blocks to process each merge operation in parallel. We leave the implementation of this parallel merge sort algorithm as an exercise for the reader.

The number of iterations needed by the parallel merge sort in Fig. 14.3 is $O(logN)$, since the number of segments that need to be merged is halved on every iteration. If there are sufficient execution resources to execute the merge operations of each iteration completely in parallel, then each iteration would have a time complexity of $O(logN)$ and a work complexity of $O(N \cdot logN)$, as seen in Chapter 13. Hence, the overall time and work complexity of the parallel merge sort algorithm presented are $O(log^2N)$ and $O(N \cdot log^2N)$, respectively. Note that this $O(N \cdot log^2N)$ work complexity is slightly higher than sequential merge sort's $O(N \cdot logN)$ complexity. The higher complexity is due to the fact that parallel merge operation has higher work complexity than sequential merge because of the execution of the co-rank function, as seen in Chapter 13. Cole's algorithm [2] for parallel merge sort overcomes this limitation and achieves time and work complexity of $O(logN)$ and $O(N \cdot logN)$, respectively.

14.4 Radix sort

To achieve better sorting complexity than $O(N \cdot logN)$, one has to use a non-comparison-based sorting algorithm. One non-comparison-based sorting algorithm

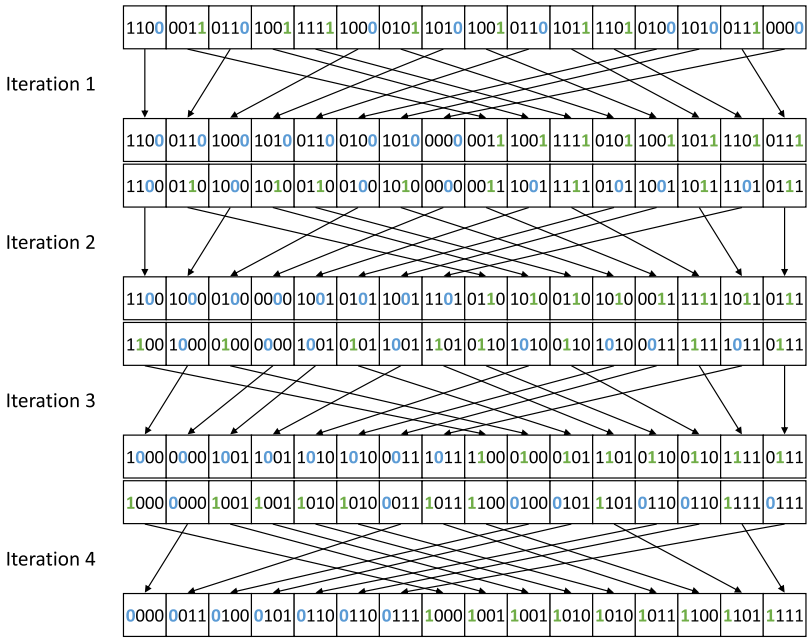


FIGURE 14.4

A radix sort example.

that is highly amenable to parallelization is radix sort. Radix sort works by distributing the keys being sorted into buckets based on a radix value (or base in a positional numeral system). If the keys consist of multiple digits, the distribution of the keys is repeated for each digit until all digits are covered. Each iteration is stable, preserving the order of the keys within each bucket from the previous iteration. When processing keys represented as binary numbers, choosing a radix value that is a power of two is convenient because it makes iterating over the digits and extracting them easy. Each iteration essentially handles a fixed-size slice of the bits from the key. We will start by using a radix of 2 (i.e., a 1-bit radix), and then extend to larger radix values later on in the chapter.

Fig. 14.4 shows an example of how a list of 4-bit integers can be sorted with radix sort using a 1-bit radix. Since the keys are four bits long and each iteration processes one bit, four iterations are required in total. In the first iteration, the least significant bit is considered. All the keys in the iteration's input list whose least significant bit is zero are placed on the left side of the iteration's output list forming a bucket for the zero bits. Similarly, all the keys in the iteration's input list whose least significant bit is one are placed on the right side of the iteration's output list forming a bucket for the one bits. Note that within each bucket in the output list, the order of the keys is preserved from that in the input list. In other words, keys that are placed into the same bucket (i.e., have the same least significant bit) must appear in the same order in the

output list as they did in the input list. We will see why this stability requirement is important when we discuss the next iteration.

In the second iteration in Fig. 14.4, the output list from the first iteration becomes the new input list, and the second least significant bit of each key is considered. Like the first iteration, the keys are separated into two buckets: a bucket for the keys whose second least significant bit is zero and another bucket for the keys whose second least significant bit is one. Since the order from the previous iterations is preserved, we observe that the keys in the second iteration's output list are now sorted by the lower two bits. In other words, all the keys whose lower two bits are 00 come first, followed by those whose lower two bits are 01, followed by those whose lower two bits are 10, followed by those whose lower two bits are 11.

In the third iteration in Fig. 14.4, the same process is repeated while considering the third bit in the keys. Again, since the order from previous iterations is preserved, the keys in the output list of the third iteration are sorted by the lower three bits. Finally, in the fourth and last iteration, the same process is repeated while considering the fourth or most significant bit. At the end of this iteration, the keys in the final output list are sorted by all four bits.

The operation performed by a radix sort iteration should remind the reader of the stable filter pattern discussed in Chapter 12. In the stable filter pattern, the input keys that meet some condition are brought to the beginning of the output list while preserving their order. A radix sort iteration performs a more general operation where the input keys that meet a condition (having their bit of interest equal to 0) are brought to the beginning of the output list, and those that do not meet the condition are brought to the end of the list, while preserving the order of keys in each sub-list. This more general pattern is sometimes referred to as the stable *partition* pattern because it divides the input list into multiple partitions. As we will see in the rest of this chapter, the techniques for parallelizing and optimizing a radix sort iteration strongly resemble those for parallelizing and optimizing the stable filter operation.

14.5 Parallel radix sort

Each iteration in radix sort depends on the entire result of the previous iteration. Hence, the iterations are performed sequentially with respect to each other. The opportunity for parallelizing radix sort arises within each iteration. For the rest of this chapter, we will focus on the parallelization and optimization of a single radix sort iteration, with the understanding that the iterations will be executed one after the other. In other words, we will focus on the implementation of a kernel that performs a single radix sort iteration and assume that the host code calls this kernel once for each iteration. We will cover multiple optimizations for this radix sort kernel. Interested readers are also encouraged to refer to OneSweep [3], which is a state-of-the-art implementation of radix sort for GPUs.

One straightforward approach to parallelize a radix sort iteration on GPUs is to make each thread responsible for one key in the input list. The thread must identify the position of the key in the output list and then store the key to that position.

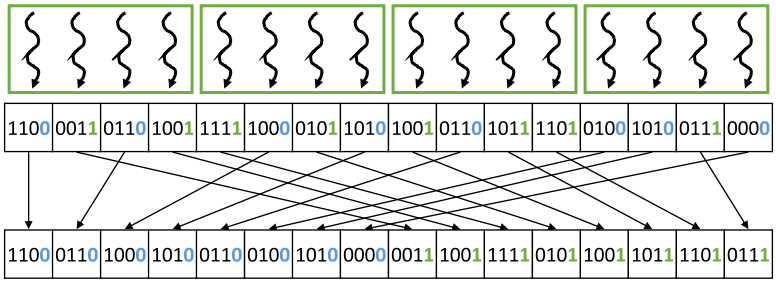


FIGURE 14.5

Parallelizing a radix sort iteration by assigning one input key to each thread.

Fig. 14.5 illustrates this parallelization approach applied to the first iteration from Fig. 14.4. Threads in Fig. 14.5 are illustrated as curvy arrows, and thread blocks are illustrated as boxes around the arrows. Each thread is responsible for the key below it in the input list. In this example, the 16 keys are processed by a grid with four thread blocks having four threads each. In practice, each thread block may have up to 1024 threads and the input is much larger resulting in many more thread blocks. However, we use a small number of threads per block to simplify the illustration.

With every thread assigned to a key in the input list, the challenge remains for each thread to identify the destination index of its key in the output list. Identifying the destination index of the key depends on whether the key maps to the zero bucket or the one bucket. For keys mapping to the zero bucket, the destination index can be found as follows:

```
destination of a zero = # zeros before
                      = # keys before - # ones before
                      = key index - # ones before
```

The destination index of a key that maps to the zero bucket (i.e., destination of a zero) is equivalent to the number of keys before the key that also map to the zero bucket (i.e., # zeros before). Since all keys either map to the zero bucket or the one bucket, the number of keys before the key mapping to the zero bucket is equivalent to the total number of keys before the key (i.e., # keys before) minus the number of keys before the key mapping to the one bucket (i.e., # ones before). The total number of keys before the key is just the index of the key in the input list (i.e., key index) which is trivially available. Hence, the only non-trivial part of finding the destination index of a key that maps to the zero bucket is counting the number of keys before it that map to the one bucket. This operation can be done using an exclusive scan as we will see shortly.

For keys mapping to the zero bucket, the destination index can be found as follows:

```
destination of a one = # zeros in total + # ones before
                     = (# keys in total - # ones in total) + # ones before
                     = input size - # ones in total + # ones before
```

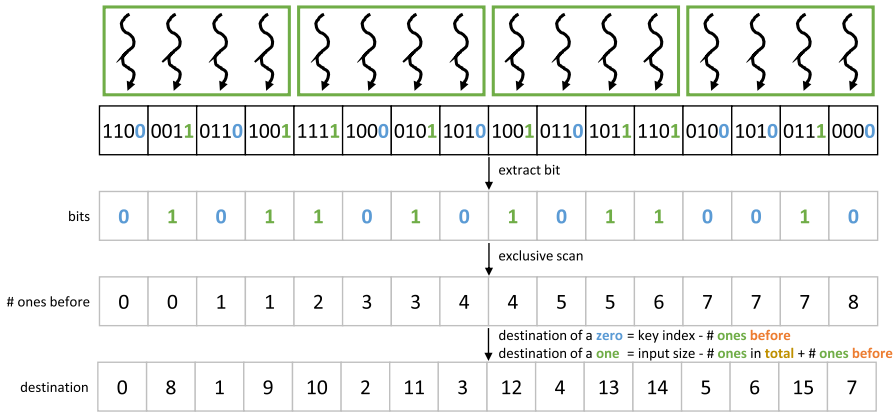


FIGURE 14.6

Finding the destination of each input key.

All keys mapping to the zero bucket must come before the keys mapping to the one bucket in the output array. For this reason, the destination index of a key that maps to the one bucket (i.e., destination of a one) is equivalent to the total number of keys mapping to the zero bucket (i.e., # zeros in total) plus the number of keys before the key that map to the one bucket (i.e., # ones before). Since all keys either map to the zero bucket or the one bucket, the total number of keys mapping to the zero bucket is equivalent to the total number of keys in the input list (i.e., # keys in total) minus the total number of keys mapping to the one bucket (i.e., # ones in total). The total number of keys in the input list is just the input size which is trivially available. Hence, the non-trivial part of finding the destination index of a key that maps to the one bucket is counting the number of keys before it that map to the one bucket, which is the same information needed for the zero bucket case. Again, this operation can be done using an exclusive scan as we will see shortly. The total number of keys mapping to the one bucket can be found as a byproduct of the exclusive scan.

Fig. 14.6 shows the operations that each thread performs to find its key's destination index in the example in Fig. 14.5. The corresponding kernel code to perform these operations is shown in Fig. 14.7. First, each thread identifies the index of the key it is responsible for (line 03), performs a boundary check (line 05), and loads the key from the input list (line 06). Next, each thread extracts from the key the bit for the current iteration to identify whether it is a zero or a one (line 07).

Here, the iteration number `iter` tells us the position of the bit that we are interested in. By shifting the key to the right by this amount, we move the bit to the rightmost position. By applying a bitwise-and operation (&) between the shifted key and a 1, we zero out all the bits in the shifted key except the rightmost bit. Hence, the value of bit will be the value of the bit we are interested in. In the example in Fig. 14.6, since the example is for iteration 0, the least significant bit is extracted as shown in the row labeled bits.

```
01  __global__ void radix_sort_iter(unsigned int* input, unsigned int* output,
02                      unsigned int* bits, unsigned int N, unsigned int iter) {
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      unsigned int key, bit;
05      if(i < N) {
06          key = input[i];
07          bit = (key >> iter) & 1;
08          bits[i] = bit;
09      }
10      gridExclusiveScan(bits, N);
11      if(i < N) {
12          unsigned int numOnesBefore = bits[i];
13          unsigned int numOnesTotal = bits[N];
14          unsigned int dst = (bit == 0)?(i - numOnesBefore)
15                                       :(N - numOnesTotal + numOnesBefore);
16          output[dst] = key;
17      }
18  }
```

FIGURE 14.7

Radix sort iteration kernel code.

Once each thread extracts the bit it is interested in from the key, it stores the bit to memory (line 08) and the threads collaborate to perform a grid-wide exclusive scan on the bits (line 10). We have already discussed how to perform a grid-wide parallel scan operation in Chapter 11. The array resulting from the exclusive scan operation contains, at each position, the sum of the bits before of that position. Since these bits are either zero or one, the sum of the bits before the position is equivalent to the number of the ones before the position (i.e., the number of keys that map to the one bucket). In the example in Fig. 14.6, the result of the exclusive scan is shown in the row labeled # ones before. Each thread accesses this array to obtain the number of ones before its position (line 12) and the total number of ones in the input list (line 13). Each thread can then identify the destination of its key using the expressions we derived previously (lines 14-15). Having identified its destination index, the thread can proceed to store the key it is responsible for at the corresponding location in the output list (line 16). In the example in Fig. 14.6, the destination indices are shown in the row labeled destination. The reader can refer to Fig. 14.5 to verify that the values obtained are indeed to right destination indices of each element.

14.6 Optimizing for memory coalescing

The approach we just described is effective at parallelizing a radix sort iteration. However, one major source of inefficiency in this approach is that the writes to the output list exhibit an access pattern that cannot be adequately coalesced. Consider how each thread in Fig. 14.5 writes its key to the output list. In the first thread block, the first thread writes to the zero bucket, the second thread writes to the one bucket, the third thread writes to the zero bucket, and the fourth thread writes to the one bucket. Hence, threads with consecutive index values are not necessarily writing to

consecutive memory locations, resulting in poor coalescing and requiring multiple memory requests to be issued per warp.

For a 1-bit radix where there are only two buckets, threads in the same warp will only write to one of two buckets. Consequently, the number of distinct cache lines written to by a warp is most likely two cache lines, or a little more depending on the alignment of the accesses. However, as we will see in Section 14.7, larger radix values result in a larger number of buckets, and consequently, more distinct cache lines to be potentially written to by the same warp. Hence, uncoalesced memory accesses can be a serious impediment to the performance of a radix sort iteration kernel.

Recall from Chapter 6 that there are various approaches to enable better memory coalescing in kernels: (1) rearranging the threads, (2) rearranging the data that the threads access, or (3) performing the uncoalesceable accesses on shared memory, and transferring data between shared memory and global memory in a coalesced way. To optimize for coalescing in this chapter, we will use the third approach. Instead of having all threads write their keys to global memory buckets in an uncoalesced manner, we will have each thread block maintain its own local buckets in the shared memory. That is, we will no longer perform a global sort as shown in Fig. 14.7. Rather, the threads in each block will first perform a block-level local sort to separate the keys mapping to the zero bucket and the keys mapping to the one bucket in shared memory. After that, the buckets will be written from shared memory to global memory in a coalesced manner.

Fig. 14.8 shows an example of how memory coalescing can be enhanced for the example in Fig. 14.5. In this example, each thread block first performs a local radix sort on the keys that it owns and stores the output list into the shared memory. The local sort can be done in the same way as the global sort was done previously, and only requires each thread block to perform a local exclusive scan instead of requiring a global one. After the local sort, each thread block writes its local buckets to the global buckets in a more coalesced way. For example, in Fig. 14.8, consider how the first thread block writes out its buckets to global memory. The first two threads both write to adjacent locations in global memory when writing the zero bucket, while the last two threads also write to adjacent locations in global memory when writing the one bucket. Hence, the majority of writes to global memory will be coalesced.

The main challenge in this optimization is for each thread block to identify the beginning position of each of its local buckets in the corresponding global bucket. The beginning position of a thread block's local buckets depends on the sizes of the local buckets in the other thread blocks. In particular, the position of a thread block's local zero bucket is after all the local zero buckets of the preceding thread blocks. On the other hand, the position of a thread block's local one bucket is after all the local zero buckets of all the thread blocks and all the local one buckets of the preceding thread blocks. These positions can be obtained by performing an exclusive scan on the thread blocks' local bucket sizes.

Fig. 14.9 shows an example of how an exclusive scan can be used to find the position of each thread block's local buckets. After completing the local radix sort, each thread block identifies the number of keys in each of its local buckets. Next,

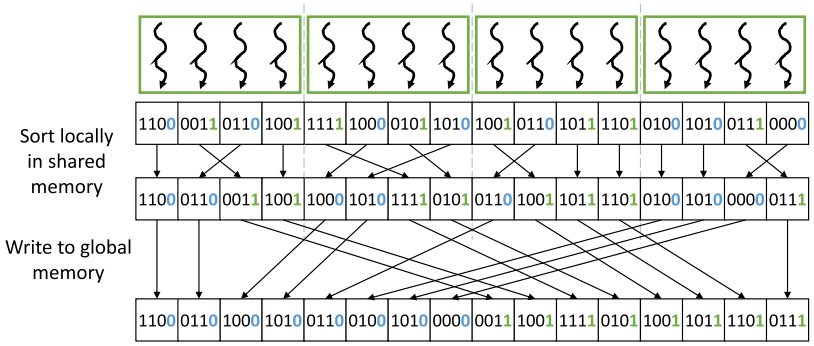


FIGURE 14.8

Optimizing for memory coalescing by sorting locally in shared memory before sorting into to the global memory.

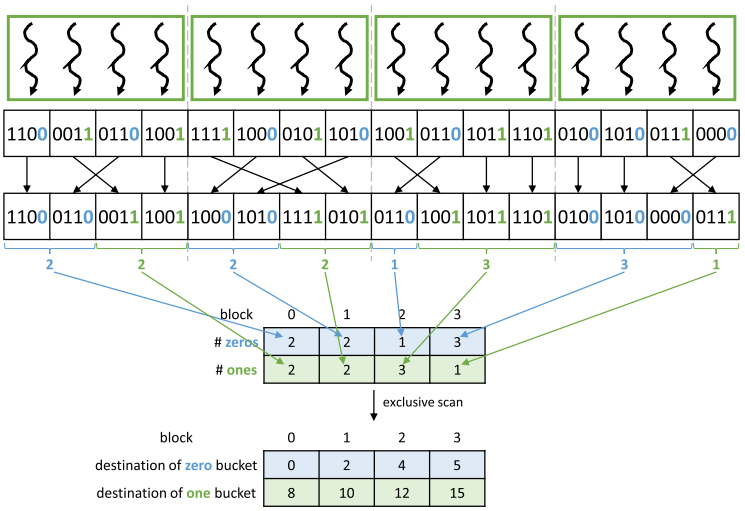


FIGURE 14.9

Finding the destination of each thread block's local buckets.

each thread block stores these values in a table as shown in Fig. 14.9. The table is stored in row-major order, meaning that it places the sizes of the local zero buckets for all thread blocks consecutively followed by the sizes of the local one buckets. After the table is constructed, an exclusive scan is executed on the linearized table. The resulting table consists of the beginning positions of each thread block's local buckets, which are the values we are looking for.

Once a thread block identifies the beginning position of its local buckets in global memory, the threads in the block can proceed to store their keys from the local buckets to the global buckets. To do so, each thread needs to keep track of the number of keys

in bucket zero vs. bucket one. During the write phase, threads in each block will be writing a key in either of the buckets depending on its thread index values. For example, for block 2 in Fig. 14.9, thread 0 writes the single key in the zero bucket and threads 1-3 write the three keys in the one bucket. In comparison, for block 3 in Fig. 14.9, threads 0-2 write the three keys in the zero bucket and thread 3 writes the one key in the one bucket. Hence, each thread needs to test if it is responsible for writing a key in the local zero bucket or the one bucket. Each block tracks the number of keys in each of its two local buckets so that the threads can determine where their threadIdx values fall into and participate in the writing of the zero bucket keys or one bucket keys. We leave the implementation of this optimization as an exercise for the reader.

14.7 Choice of radix value

So far, we have seen how radix sort can be parallelized using a 1-bit radix as an example. For the 4-bit keys in the example, four iterations (one for each bit) are needed for the keys to be fully sorted. In general, for N-bit keys, N iterations are needed to fully sort the keys. To reduce the number of iterations needed, a larger radix value can be used.

Fig. 14.10 shows an example of how radix sort can be performed using a 2-bit radix. Each iteration uses two bits to distribute the keys to buckets. Hence, the 4-bit keys can be fully sorted using only two iterations. In the first iteration, the lower two bits are considered. The keys are distributed across four buckets corresponding to the keys where the lower two bits are 00, 01, 10, and 11. In the second iteration, the upper two bits are considered. The keys are then distributed across four buckets based on the upper two bits. Similar to the 1-bit example, the order of the keys within each bucket is preserved from the previous iteration. Preserving the order of the keys within each bucket ensures that after the second iteration, the keys are fully sorted by all four bits.

Similar to the 1-bit example, each iteration can be parallelized by assigning a thread to each key in the input list to find the key's destination index and store it in the output list. To optimize for memory coalescing, each thread block can sort its keys locally in the shared memory, then write the local buckets to global memory in a coalesced manner. An example of how to parallelize a radix sort iteration and optimize it for memory coalescing using the shared memory is shown in Fig. 14.11.

The key distinction between the 1-bit example and the 2-bit example is how to separate the keys into four buckets instead of two. For the local sort inside of each thread block, a 2-bit radix sort is performed by applying two consecutive 1-bit radix sort iterations. Each of these 1-bit iterations requires its own exclusive scan operation, however, these operations are local to the thread block so there is no coordination across thread blocks in between the two 1-bit iterations. In general, for an r-bit radix, r local 1-bit iterations are needed to sort the keys into 2^r local buckets.

After the local sort is complete, each thread block must find the position of each of its local buckets in the global output list. Fig. 14.12 shows an example of how the

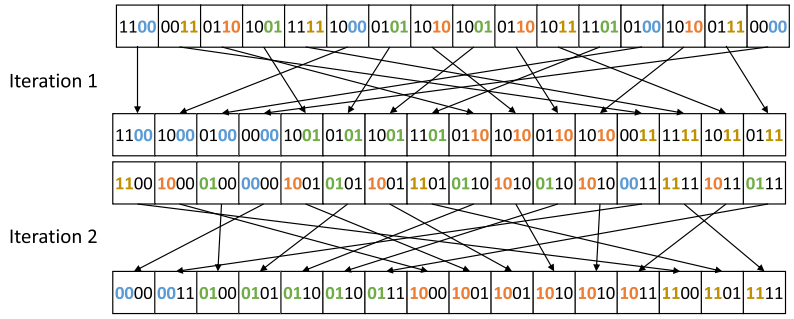


FIGURE 14.10

Radix sort example with 2-bit radix.

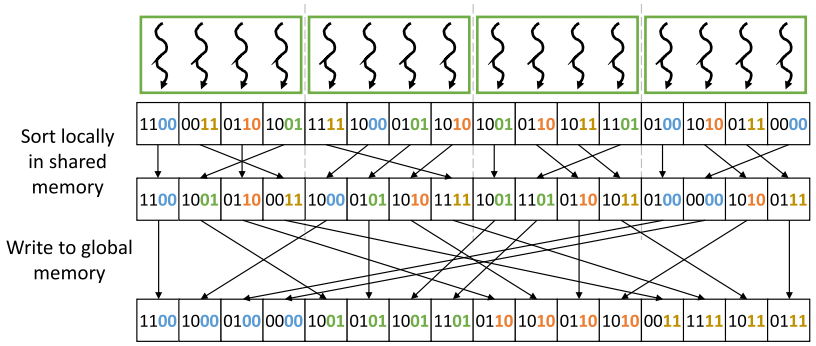


FIGURE 14.11

Parallelizing a radix sort iteration and optimizing it for memory coalescing using the shared memory for a 2-bit radix.

destination of each local bucket can be found for the 2-bit radix example. The procedure is similar to the 1-bit example in Fig. 14.9. Each thread block stores the number of keys in each local bucket to a table that is then scanned to obtain the global position of each of the local buckets. The main distinction from the 1-bit radix example is that each thread block has four local buckets instead of two, so the exclusive scan operation is performed on a table with four rows instead of two. In general, for an r-bit radix, the exclusive scan operation is performed on a table with 2^r rows.

We have seen that the advantage of using a larger radix is that it reduces the number of iterations needed to fully sort the keys. Fewer iterations means fewer grid launches, global memory accesses, and global exclusive scan operations. However, using a larger radix also has disadvantages. The first disadvantage is that each thread block has more local buckets where each bucket has fewer keys. As a result, each thread block has more distinct global memory bucket sections that it needs to write to and less data that it needs to write to each section. For this reason, the opportunities for memory coalescing decrease as the radix gets larger. The second disadvantage is

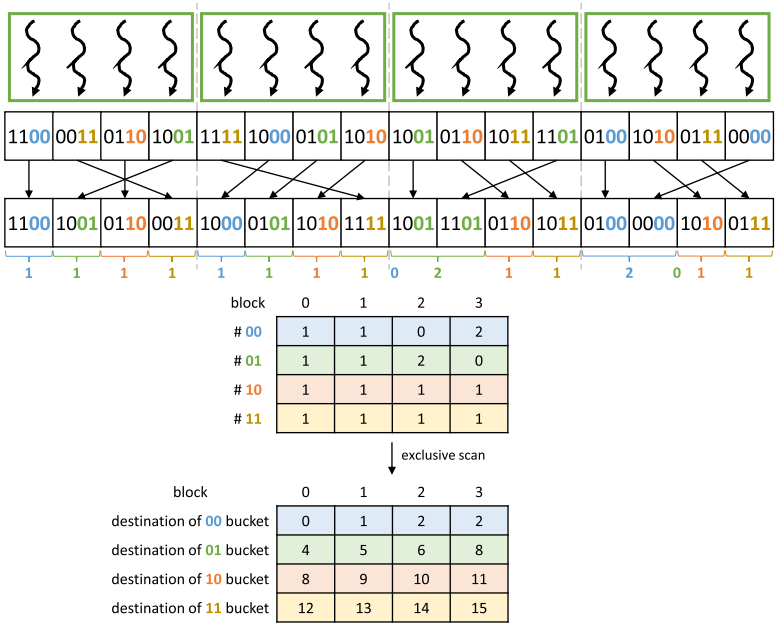


FIGURE 14.12

Finding the destination of each block's local buckets for 2-bit radix.

that the table on which the global exclusive scan is applied gets larger with a larger radix. For this reason, the overhead of the global exclusive scan increases as the radix increases. Therefore, the radix cannot be made arbitrarily large. The choice of radix value must strike a balance between the number of iterations on one hand, and the memory coalescing behavior as well as the overhead of the global exclusive scan on the other hand. We leave the implementation of radix sort with a multi-bit radix as an exercise for the reader.

14.8 Thread coarsening to improve coalescing

The overhead of parallelizing radix sort across many thread blocks is poor coalescing of writes to global memory. Each thread block has its own local buckets that it writes to global memory. Having more thread blocks means having fewer keys per thread block, which means that the local buckets are going to be smaller, exposing fewer opportunities for coalescing when they are written to global memory. If these thread blocks were to be executed in parallel, the price of poor coalescing might be worth paying. However, if these thread blocks were to be serialized by the hardware, the price is paid unnecessarily.

To address this issue, thread coarsening can be applied whereby each thread is assigned to multiple keys in the input list instead of just one. Fig. 14.13 illustrates

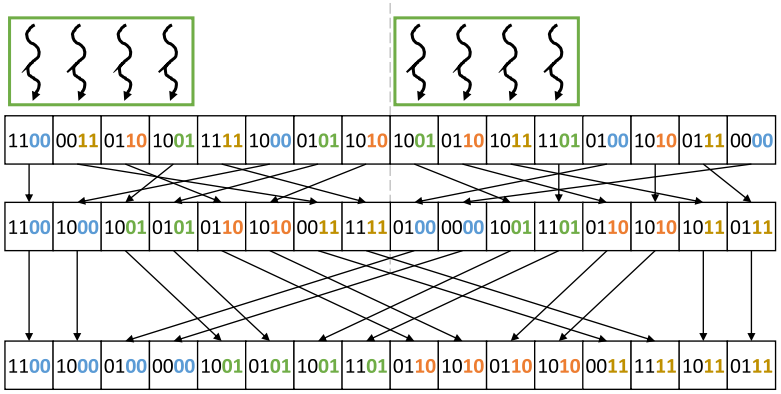


FIGURE 14.13

Radix sort for 2-bit radix with thread coarsening to improve memory coalescing.

how thread coarsening can be applied to a radix sort iteration for the 2-bit radix example. In this case, each thread block is responsible for more keys than the example in Fig. 14.11. Consequently, the local buckets of each thread block are larger exposing more opportunities for coalescing. When comparing Fig. 14.11 and Fig. 14.13, it is clear that in Fig. 14.13, it is more likely the case that consecutive threads write to consecutive memory locations.

Another overhead of parallelizing radix sort across many thread blocks is the overhead of performing the global exclusive scan to identify the destination of each thread block's local buckets. Recall from Fig. 14.12 that the size of the table on which the exclusive scan is performed is proportional to the number of buckets as well as the number of blocks. By applying thread coarsening, the number of blocks is reduced, thereby reducing the size of the table and the overhead of the exclusive scan operation. We leave the application of thread coarsening to radix sort as an exercise for the reader.

14.9 Other parallel sort methods

The algorithms outlined in this chapter are only some of the many possible ways to sort data in parallel. In this section, we briefly outline some of the other methods that may be of interest to readers.

The odd-even transposition sort covered in Section 14.2 uses a fixed pattern of comparisons and it swaps elements when they are out of order. It is easily parallelized because each step compares pairs of keys that do not overlap. There is an entire category of sorting methods that use fixed patterns of comparison to sort sequences, often in parallel. These methods are usually referred to as sorting networks and the most well-known parallel sorting networks are Batcher's bitonic sort and odd-even merge

sort [4]. Batcher's algorithms operate on sequences of fixed length and are more efficient than odd-even transposition sort, requiring only $O(N \cdot log^2 N)$ comparisons for a sequence of N elements. Even though the cost of these algorithms is asymptotically worse than the $O(N \cdot logN)$ cost of methods like merge sort, in practice, they are often the most efficient methods on small sequences because of their simplicity.

Most comparison-based parallel sorts that do not use the fixed set of comparisons typical of sorting networks can be divided into two broad categories. The first partitions the unsorted input into tiles, sorts each tile, and then performs most of its work in combining these tiles to form the output. The merge sort described in this chapter is an example of such an algorithm; most of the work is performed in the merge tree that combines sorted tiles. The second category focuses most of its work on partitioning the unsorted sequence, such that combining partitions is relatively trivial. Sample sort algorithms [5] are the typical example of this category. Sample sort begins by selecting p-1 keys from the input (e.g., at random), sorts them, and then uses them to partition the input into p buckets such that all keys in bucket k are greater than all keys in any bucket j $<$ k and less than all in any bucket j $>$ k. This step is analogous to a p-way generalization of the 2-way partitioning performed by quicksort. Having partitioned the data in this way, each bucket can be sorted independently, and the sorted output is formed by merely concatenating the buckets in order. Sample sort algorithms are often the most efficient choice for extremely large sequences where data must be distributed across multiple physical memories, including across the memories of multiple GPUs in a single node. In practice, over-sampling the keys is common, since modest over-sampling will result in balanced partitions with high probability [6].

Just as merge sort and sample sort typify bottom-up and top-down strategies to comparison-based sorting, radix sorting algorithms can be designed to follow a bottom-up or top-down strategy. The radix sort described in this chapter is more completely described as a least-significant bit (LSB), or more generally least-significant digit (LSD), radix sort. The successive steps of the algorithm start with the least-significant digit of the key and work towards the most-significant digit. A most-significant digit (MSD) radix sort adopts the opposite strategy. It begins by using the most-significant digit to partition the input into buckets corresponding to the possible values of that digit. This same partitioning is then applied independently in each bucket using the next-most-significant digit. Upon reaching the least-significant digit, the entire sequence will have been sorted. Like sample sort, MSD radix sort is often a better choice for very large sequences. Whereas the LSD radix sort requires global shuffling of data in each step, each step of MSD radix sort operates on progressively more localized regions of the data.

14.10 Summary

In this chapter, we have seen how to sort keys (and their associated values) on GPUs in parallel. We started with two comparison-based parallel sorting algorithms: parallel odd-even sort and parallel merge sort. Parallel odd-even sort iteratively swaps

pairs of adjacent elements in the list until the list is sorted, alternating between odd pairs and even pairs to avoid race conditions. Parallel merge sort performs independent merge operations of different input segments in parallel, as well as parallelizing within each merge operation by leveraging the parallel merge operation discussed in Chapter 13.

We then focused on a non-comparison-based sorting algorithm, radix sort, which has the advantage of having work complexity that is lower than $O(N \cdot logN)$. Radix sort works by iteratively distributing keys across buckets. The distribution process is repeated for each digit in the key, while preserving the order from the previous digit's iteration, to ensure that the keys are sorted according to all the digits at the end. Each iteration is parallelized by assigning a thread to each key in the input list and having that thread find the destination of the key in the output list, which involves collaborating with other threads to perform an exclusive scan operation.

One of the key challenges with optimizing radix sort is achieving coalesced memory accesses when writing the keys to the output list. An important optimization to enhance coalescing is to have each thread block perform a local sort to local buckets in shared memory, then write each local bucket to global memory in a coalesced manner. Another optimization is to increase the size of the radix to reduce the number of iterations needed, and thus the number of grids launched. However, the radix size should not be increased too much because it would result in poorer coalescing of memory accesses and more overhead from the global exclusive scan operation. Finally, applying thread coarsening is effective at improving memory coalescing as well as reducing the overhead of the global exclusive scan.

Implementing and optimizing parallel sorting algorithms on GPUs is complex and programmers are encouraged to use parallel sorting libraries for GPUs such as Thrust [7,8] or CUB [9] rather than to implement their own sorting kernels from scratch. Nevertheless, parallel sorting remains an interesting case study of the trade-offs that go into optimizing parallel patterns.

14.11 Exercises

1. Extend the kernel in Fig. 14.7 by using shared memory to improve memory coalescing.
2. Extend the kernel in Fig. 14.7 to work for a multi-bit radix.
3. Extend the kernel in Fig. 14.7 by applying thread coarsening to improve memory coalescing.
4. Implement parallel merge sort using the parallel merge implementation from Chapter 13.

References

[1] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore gpus, in: 2009 IEEE International Symposium on Parallel & Distributed Processing, IEEE, 2009, pp. 1–10.

[2] R. Cole, Parallel merge sort, SIAM Journal on Computing 17 (4) (1988) 770–785.
[3] A. Adinets, D. Merrill, Onesweep: a faster least significant digit radix sort for gpus, arXiv preprint, arXiv:2206.01784, 2022.
[4] K.E. Batcher, Sorting networks and their applications, in: Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, 1968, pp. 307–314.
[5] W.D. Frazer, A.C. McKellar, Samplesort: a sampling approach to minimal storage tree sorting, Journal of the ACM 17 (3) (1970) 496–507.
[6] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, M. Zagha, A comparison of sorting algorithms for the connection machine cm-2, in: Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, 1991, pp. 3–16.
[7] N. Bell, J. Hoberock, Thrust: a productivity-oriented library for cuda, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 359–371.
[8] NVIDIA, Thrust: the C++ parallel algorithms library, https://nvidia.github.io/cccl/thrust/index.html, 2025.
[9] NVIDIA, CUB, https://nvidia.github.io/cccl/cub/index.html, 2025.

CHAPTER

Advanced optimizations for matrix multiplication

15

CONTENTS

15.1 Background 349
15.2 Data reuse analysis 350
15.3 Using larger tiles with thread coarsening 352
15.4 Register tiling of the input tiles 357
15.5 Coalesced storing of the output tile 360
15.6 Eliminating bank conflicts 361
15.7 Occupancy considerations 363
15.8 Software pipelining 364
15.9 Specialized software and hardware support 368
15.10 Summary 370
15.11 Exercises 370
References 370

Matrix multiplication is an important operation in science, engineering, and graphics applications. Due to its high computation complexity, it exhibits a high degree of data reuse and thus significant optimization opportunities. In Chapter 3, we used matrix multiplication as an example of mapping multi-dimensional grids to data. Subsequently, in Chapter 5, we used matrix multiplication as an example of using shared memory tiling to reduce global memory accesses. In this chapter, we will dive more deeply into matrix multiplication and the advanced optimizations for this important computation on the GPU architecture.

15.1 Background

Matrix-matrix multiplication, or matrix multiplication in short, is widely used in mathematical formulations of science and engineering problems. Conceptually, it is more intuitive to start with vector-matrix multiplication, a special case of matrix multiplication, where the first matrix operand has only one row and thus can be viewed as a vector v. By multiplying a vector v with a matrix B to generate a vector u, i.e., $u = v \times B$, one linearly transforms v into u. That is, each element of u is a linear combination of all elements of v. Column j of B defines the coefficients used for linearly

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00025-1
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

combining all elements of v into element j of u. For example, in graphics applications, one can perform a coordinate rotational transformation by multiplying the 3D vector coordinate of a point in space with a 3×3 matrix, along with a vector addition if shifting is needed and/or multiplying all elements with a scaling factor, to derive its coordinate vector according to a new vantage point along with new orientation and scale.

To perform the same linear transformation on multiple vectors, one can collect these vectors into a matrix A, where each row is one of the vectors, and perform a matrix multiplication $D = A \times B$. After multiplication, each row of D contains one of the transformed vectors. Going back to the coordinate transformation example, one may need to scale the matrix multiplication result and perhaps add a (scaled) matrix C, where each row of C is a vector to be added to the corresponding transformed row vector of A. Thus, we have a general form $D = \alpha A \times B + \beta C$, which is referred to as *General Matrix Multiply (GEMM)* in linear algebra and high-performance computing communities. It should be clear that the matrix multiplication step $A \times B$ is likely the most compute intensive step of GEMM. Thus, we focus on the matrix multiplication step of GEMM in this chapter.

Depending on the context of the computation, the matrices involved in the matrix multiplication operation may each be stored in row major or column major order. The particular storage format may be a result of operating on the original matrix or its transpose, as discussed in Chapter 6. Throughout this chapter, we assume that the two matrices involved in the matrix multiplication are both in row major order. However, the optimization techniques we describe can also be applied to other combinations of storage formats. In fact, libraries that support GEMM operations typically support these different combinations.

The applicability and benefit of optimizations for matrix multiplication depend on the dimensions of the matrices involved. Before the rise of deep learning, matrix multiplications in applications typically involve matrices of modest dimensions. This is because these applications typically apply linear transformations to vectors with modest numbers of elements. As we will see in Chapter 19 and Chapter 20, one can formulate the training of Convolutional Layers of the Convolutional Neural Networks and the Attention Layers of Large Language Models into multiplications using large matrices. These large-scale matrix multiplications motivate optimizations that were less profitable when applied to smaller-scale multiplications. Many optimizations covered in this chapter are designed to benefit large-scale matrix multiplications.

15.2 Data reuse analysis

Fig. 15.1 reviews tiled matrix multiplication which was covered in Chapter 5, using more general tile dimensions. In this example, a thread block is assigned to compute an output tile of matrix C of size $m \times n$. The threads in the thread block collaboratively iterate over the input tiles of matrices A and B, loading a $m \times k$ tile of matrix A and the corresponding $k \times n$ input tile of matrix B from global memory to shared

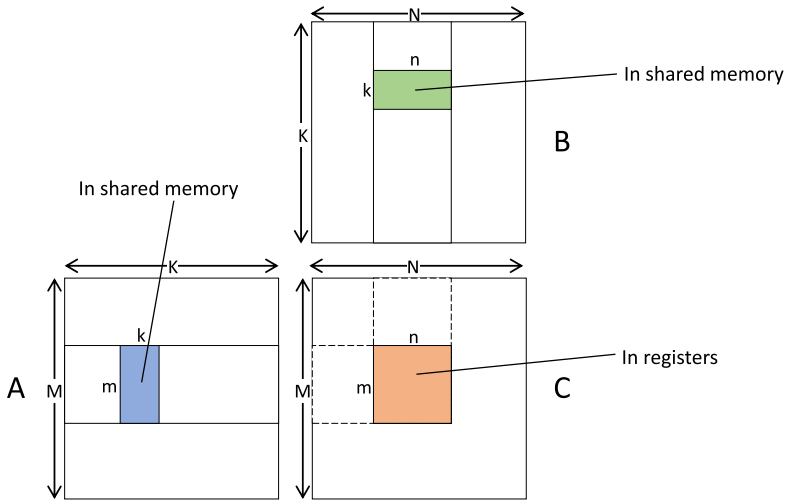


FIGURE 15.1

Review of tiled matrix multiplication.

memory. For each pair of input tiles, the threads compute the partial contribution of the input tiles to each element in the output tile. The output tile is stored collectively in the registers of the threads in the thread block. When the computation is over, the threads in the thread block write the values of the output tile from their registers to the global memory.

In Chapter 5, our data reuse analysis assumed that the input and output tiles were square. Let us now analyze the data reuse achieved by tiled matrix multiplication for non-square tiles. For each pair of input tiles, the threads in the thread block collaboratively compute k multiply and k add operations for each of the $m \times n$ output elements, making the total number of floating point operations $2 \cdot m \cdot n \cdot k$. At the same time, the threads in the thread block will collaboratively load a $m \times k$ tile of A and a $k \times n$ tile of B where each element loaded is 4 B, making the total number of bytes loaded $4 \cdot k \cdot (m + n)$. If we take the ratio of these two numbers, we get an arithmetic intensity of $0.5 \cdot m \cdot n/(m + n)$ operations per byte.

As we saw in Chapter 5, a larger arithmetic intensity would allow the computation to better overcome the memory bandwidth bottleneck and achieve better performance. The ratio $0.5 \cdot m \cdot n/(m + n)$ can be increased by using larger values of m and n, i.e., using larger output tiles. For example, if $m = n = 32$, the arithmetic intensity would be 8 FLOP/B, which would keep the kernel memory-bound on an H100 GPU (see Chapter 5). On the other hand, if we increase $m = n = 128$, then the arithmetic intensity would increase to 32 FLOP/B, which would make the kernel compute-bound on an H100 GPU.

The intuition behind why increasing m and n improves the arithmetic intensity of the computation is that larger output tiles allow the same input data to be reused

across more output values. For example, in Fig. 15.1, if we increase n from 32 to 128, we enlarge the output tile by 4× in the horizontal direction. In other words, each thread block would calculate the equivalent of four original output tiles. Calculating these output tiles uses the same input tile of A, as shown on the left part of Fig. 15.1. In the original configuration, the four output tiles were calculated using four different thread blocks, each redundantly loading the same input tile of `A`. After increasing n by 4×, all four output tiles are calculated by only one thread block and the input tile of `A` needs to be loaded only once by the thread block. One can further reduce the redundant loading of `B` input tiles by enlarging the height `m`. This elimination of redundant loading of `A` and `B` input tiles across thread blocks enhances data reuse and thus increases arithmetic intensity.

We look at how we can increase the tile sizes for tiled matrix multiplication in Section 15.3.

15.3 Using larger tiles with thread coarsening

Using larger tiles in tiled matrix multiplication would necessitate assigning to each thread in the block multiple elements to compute per output tile and multiple elements to load per input tile. Hence, this optimization can be thought of as a thread coarsening optimization, where the overhead of finer-grain parallelization is the need to load the same input value more times by different thread blocks. This optimization is similar to the thread coarsening optimization applied in Chapter 8 on the stencil computation where thread coarsening was also used to increase the output tile size for increased data reuse and arithmetic intensity.

Fig. 15.2 illustrates how tiled matrix multiplication can be performed with large tiles. Each thread block computes an output tile of size `bM`×`bN`. We refer to the output tile of a thread block as a *block-level output tile*. When the size of the block-level output tiles is larger than the number of threads in a block, each thread in the block must be responsible for computing multiple elements of the block-level output tile. For example, if each block-level output tile consists of `bM`×`bN` = 128 × 128 elements and each thread block has 16 × 16 threads, each block-level output tile can be divided across threads such that each thread takes an output tile of size `tM`×`tN` = 8 × 8. We refer to the `tM`×`tN` tile taken by a thread as its `thread-level output tile`.

In general, the input and output matrices for matrix multiplication can be logically divided into a hierarchy of tiles. In our example, we have block-level tiles and thread-level tiles. Each tile in a tile hierarchy can be assigned to a unit of the thread hierarchy. In our example, each block-level tile is assigned to a thread block and each thread-level tile is assigned to a thread. In this section, we focus on the kernel code that uses only the block-level and the thread-level tiles. As the reader has probably guessed, it can also be useful to reason about tiles at other levels such as warp-level tiles, GPU-level tiles, and even compute cluster node-level tiles. In fact, we will use warp-level tiles in Section 15.5.

Fig. 15.3 shows the kernel code for performing tiled matrix multiplication with large tiles. First, all threads in each block identify the starting row index and column

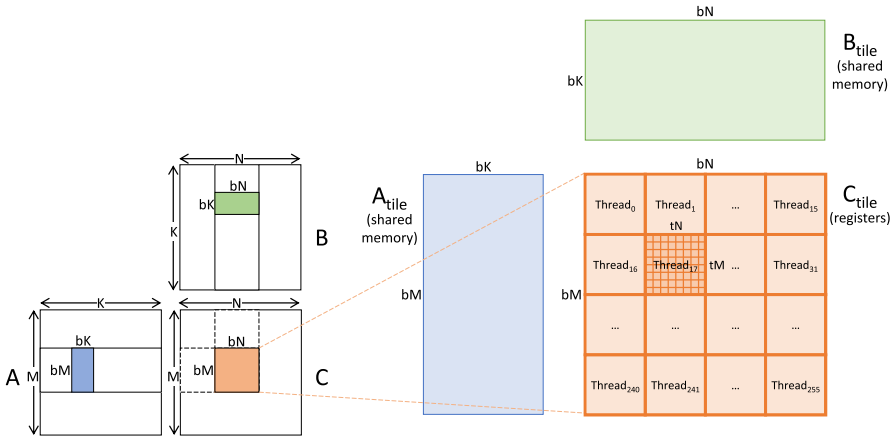


FIGURE 15.2

Tiled matrix multiplication with large tiles.

index of their block-level output tile within the `C` output matrix, `bRow` and `bCol`, by multiplying its block indices by the dimensions of the block output tile (lines 05-06). Next, each thread identifies the starting row and column of its thread-level output tile within the block-level output tile (`tRow` and `tCol`) by multiplying its thread indices by the dimensions of the thread-level output tile (lines 09-13).

Each thread declares a `tM`×`tN` local array `C_r` to hold its output tile (line 16) and calls a device function `clear()` to initialize all its elements to zero. Recall from Chapter 5 and Chapter 6 that a local array is placed in the global memory by default. The compiler can place a local array into registers for faster access if all the accesses to the array have constant indices. To ensure that `C_r` is placed in registers, we must ensure that all accesses to `C_r` have constant indices. To do so, we unroll all the loops that iterate over `C_r`, which we will do in the implementations of the device functions that access `C_r`. As a result, each thread's thread-level output tile is placed in the thread's registers (`C_r` on line 16).

After that, each block of threads proceeds to iteratively load the input tiles and compute their contribution to the output tiles using several device functions along the way. We first explain the main code from Fig. 15.3 and then explain each device function later in this section.

As shown in Fig. 15.2, a horizontal `bM` × `K` panel of `A` and a vertical `K` × `bN` panel of `B` are needed to calculate the output tile of a thread block. These panels are divided into $\frac{K}{bK}$ input tile pairs. All threads in the same block iterate over the input tile pairs to compute their contributions to the block-level output tile (line 20). We first declare the shared memory arrays `A_s` and `B_s` to hold the input tiles (lines 23-24). In each iteration of the `for-loop`, we use the device function `loadTile()` to load a pair of input tiles from `A` and `B` from global memory into `A_s` and `B_s` in shared memory (lines 25-26). After loading both input tiles, a barrier synchronization is executed (line 27) to ensure that no thread begins computing before all the data is available.

```
01  __global__ void mm_tiled_kernel(float* A, float* B, float* C, unsigned int M,
02      unsigned int N, unsigned int K) {
03
04      // Identify the block's tile
05      unsigned int bRow = blockIdx.y*bM;
06      unsigned int bCol = blockIdx.x*bN;
07
08      // Identify the thread's tile within the block
09      unsigned int tilesPerBlockX = bN/tN;
10      unsigned int ty = threadIdx.x/tilesPerBlockX;
11      unsigned int tx = threadIdx.x%tilesPerBlockX;
12      unsigned int tRow = ty*tM;
13      unsigned int tCol = tx*tN;
14
15      // Initialize the output tile
16      float C_r[tM][tN];
17      clear(C_r, tM, tN);
18
19      // Iterate over input tiles
20      for(unsigned int tile = 0; tile < (K + bK - 1)/bK; ++tile) {
21
22         // Load A and B tiles to shared memory
23         __shared__ float A_s[bM*bK];
24         __shared__ float B_s[bK*bN];
25         loadTile(&A[bRow*K + tile*bK], K, M - bRow, K - tile*bK, &A_s[0], bK, bM, bK);
26         loadTile(&B[tile*bK*N + bCol], N, K - tile*bK, N - bCol, &B_s[0], bN, bK, bN);
27         __syncthreads();
28
29         // Compute with shared memory tiles
30         mm(tM, tN, bK, &A_s[tRow*bK], bK, &B_s[tCol], bN, C_r);
31         __syncthreads();
32
33      }
34
35      // Write output tile
36      float* c = &C[(bRow + tRow)*N + bCol + tCol];
37      unsigned int maxRow = (bRow + tRow < M)?(M - (bRow + tRow)):0;
38      unsigned int maxCol = (bCol + tCol < N)?(N - (bCol + tCol)):0;
39      writeTile(c, N, maxRow, maxCol, C_r, tM, tN);
40
41  }
```

FIGURE 15.3

Code for tiled matrix multiplication with large tiles.

After the input tiles are available in shared memory, each thread calls a device function `mm()` (line 30) to multiply its portions of the input tiles and accumulate their contribution to its thread-level output tiles. The threads then perform another barrier synchronization (line 31) to make sure that all threads have finished computing before the next iteration's input tiles are loaded. When all the input tiles have been processed, the threads exit the loop and call a device function `writeTile()` to write their thread-level output tiles from the registers to the global memory (line 39).

Let us now turn our attention to the device functions used in the main kernel code in Fig. 15.3. Keep in mind that all these device functions are inlined into the main kernel. Fig. 15.4 shows the implementation of the `clear()` function that we used to initialize the output tile array `C_r` to zero. The implementation is quite simple. Two nested loops iterate over the rows (line 04) and columns (line 06) of the array and initialize each array element to zero (line 07). Importantly, the loops are annotated with `#pragma unroll` to ensure that they are fully unrolled, allowing the `C_r` array to have constant indices and to be placed in the registers.

```
01    __device__ __forceinline__ void clear(float C_r[][tN], unsigned int m,
02        unsigned int n) {
03        #pragma unroll
04        for(unsigned int row = 0; row < m; ++row) {
05            #pragma unroll
06            for(unsigned int col = 0; col < n; ++col) {
07                C_r[row][col] = 0.0f;
08            }
09        }
10    }
```

FIGURE 15.4

Code for clearing the output tile.

```
01    __device__ __forceinline__ void loadTile(float* T, unsigned int lda,
02        unsigned int maxRow, unsigned int maxCol, float* T_s, unsigned int ldas,
03        unsigned int height, unsigned int width) {
04        unsigned int rowsPerSubTile = NUM_THREADS_PER_BLOCK/width;
05        unsigned int numSubtiles = height/rowsPerSubTile;
06        #pragma unroll
07        for(unsigned int subTile = 0; subTile < numSubtiles; ++subTile) {
08            unsigned int row = subTile*rowsPerSubTile + threadIdx.x/width;
09            unsigned int col = threadIdx.x%width;
10            if(row < maxRow && col < maxCol) {
11                T_s[row*ldas + col] = T[row*lda + col];
12            } else {
13                T_s[row*ldas + col] = 0.0f;
14            }
15        }
16    }
```

FIGURE 15.5

Code for loading the input tiles from global memory to shared memory.

Fig. 15.5 shows the implementation of the `loadTile()` function that we used to load the input tiles from the global memory to the shared memory. The function takes as parameters (lines 01-03) the starting address `T` of the tile in the original matrix in global memory, the leading dimension[1] (`lda`) of the tile, and the boundaries (`maxRow` and `maxCol`) of the tile in the original matrix. It also takes the starting address of the tile to be stored in shared memory (`T_s`), its leading dimension (`ldas`), and its size (`height` and `width`).

When the number of threads in the block is smaller than the number of elements in the input tile, each thread needs to load multiple input elements. For example, if the input tile dimension is 128×8 (1024 elements) and each block has 256 threads, each thread needs to load four elements. For this reason, the input tile is logically divided into four sub-tiles and loaded one sub-tile at a time. The code in Fig. 15.5 calculates the number of sub-tiles (line 04-05) and iterates over the sub-tiles (line 07). For each sub-tile, the thread calculates the index of the element that it should load (lines 08-09), makes sure the element is within bounds (line 10), and loads the element (line

[1] In the row-major layout, the leading dimension of a matrix or tile is the number of elements in memory between successive elements of the same column. In our case, the leading dimension of a tile is the width of the larger matrix containing the tile.

```
01    __device__ __forceinline__ void mm(unsigned int m, unsigned int n, unsigned int k
02        float* a, unsigned int lda, float* b, unsigned int ldb, float c[][tN]) {
03        #pragma unroll
04        for(unsigned int row = 0; row < m; ++row) {
05            #pragma unroll
06            for(unsigned int col = 0; col < n; ++col) {
07                #pragma unroll
08                for(unsigned int i = 0; i < k; ++i) {
09                    c[row][col] += a[row*lda + i]*b[i*ldb + col];
10                }
11            }
12        }
13    }
```

FIGURE 15.6

Code for performing a matrix multiplication with the input tiles.

11) or sets the shared memory value to zero if the element is out of bounds (line 13). One possible optimization to this function is to use vector loads so that each thread would load four elements in a single instruction, as described in Chapter 6. We leave this optimization as an exercise for the reader.

Fig. 15.6 shows the implementation of the `mm()` function that each thread calls to compute the contribution of the current input tiles to its thread-level output tile. As is shown on the right side of Fig. 15.2, the calculation of each thread-level output tile, e.g. the one highlighted for Thread 17, requires contributions from only a `tM` × `bK` sub-tile of the block-level tile of `A` and a `bK` × `tN` sub-tile of the block-level tile of `B`. We will refer to these sub-tiles as the *thread-level input tile* for the thread calculating the corresponding thread-level output tile. Furthermore, we will refer to the common dimension between the block-level tile of `A` and the block-level tile of `B`, i.e., the horizontal dimension of the block-level tile of `A` and the vertical dimension of the block-level tile of `B`, as the inner dimension of the block-level matrix multiplication. Since the thread-level subtiles consists of entire rows of the block-level tile of `A` and entire columns of the block-level tiles of `B`, the inner dimension of the block-level matrix multiplication is also the inner dimension (`bK`) of the thread-level multiplication in this design.

The function takes as parameters (lines 01-02) the dimensions of the thread-level output tile (`m` and `n`), the inner dimension of the thread-level matrix multiplication (`k`), the starting addresses of the two thread-level input tiles (`a` and `b`), their leading dimensions (`lda` and `ldb`), and the starting address of the thread-level output tile (`c`). The leading dimensions are needed to calculate the linearized address of the thread-level input tile elements. Although we pass an address to the thread-level output tile, the compiler eventually inlines the function and places the array in registers since it will be accessed with constant indices.

The code for the `mm` device function is quite simple. The outer two loops iterate through the thread-level output tile elements (lines 04 and 06). Importantly, the loops are unrolled (lines 03 and 05) to allow the output tile to be placed in registers. For each element of the output tile, the inner loop iterates through the corresponding row and column of the thread-level input tiles (line 08) and performs the multiply and

```
01    __device__ __forceinline__ void writeTile(float*c, unsigned int ldc, unsigned int
02       maxRow, unsigned int maxCol, float C_r[][tN], unsigned int m, unsigned int n)
03        #pragma unroll
04        for(unsigned int row = 0; row < m; ++row) {
05            #pragma unroll
06            for(unsigned int col = 0; col < n; ++col) {
07                if(row < maxRow && col < maxCol) {
08                    c[row*ldc + col] = C_r[row][col];
09                }
10            }
11        }
12    }
```

FIGURE 15.7

Code for writing the output tiles to global memory.

add operation (line 09). Note that although the row and column indices are in terms of the thread-level input tiles, the elements are actually fetched from the block-level input tile arrays in the shared memory. Since `a` and `b` are set to the starting address of the thread-level input tiles within the block-level input tile arrays (`&A_s[tRow*bK]` and `&B_s[tCol]` on line 30 of Fig. 15.3), we simply need to multiply the row indices (`row` for `a` and `i` for `b`) for the thread-level input tile elements by the leading dimensions `lda` and `ldb` (which are equivalent to the width of the block-level input tiles) to skip to the correct place in the block-level input tiles where the thread-level tile elements needed for the current iteration reside.

Fig. 15.7 shows the implementation of the `writeTile()` function that each thread uses to write its thread-level output tile from its registers to global memory. The function takes as parameters (lines 01-02) the starting address of the tile to be stored to global memory (`c`), its leading dimension (`ldc`), and its boundaries (`maxRow` and `maxCol`). It also takes the starting address of the tile to be accessed from the registers (`C_r`) and its size (`m` and `n`). Note that the starting address for writing the tile into global memory is calculated by the main kernel in line 36 of Fig. 15.3 based on the block-level and thread-level output tile design shown in Fig. 15.2.

The two loops in the `writeTile` function iterate over the output tile and are unrolled to ensure that output tile is placed in registers (lines 03-06). After a boundary check (line 07), elements that are in bounds are stored to global memory. One possible optimization to this function is to use vector stores so that each thread would store four elements in a single instruction, as described in Chapter 6. We leave this optimization as an exercise to the reader.

The implementation of tiled matrix multiplication in this section using large block-level and thread-level tiles improves the arithmetic intensity and overall performance of the computation. However, there are still additional optimizations that can be applied to further improve performance. We discuss these optimizations in the rest of this chapter.

15.4 Register tiling of the input tiles

So far we have used shared memory tiling to overcome the global memory bandwidth bottleneck by loading reused data once to shared memory and then accessing

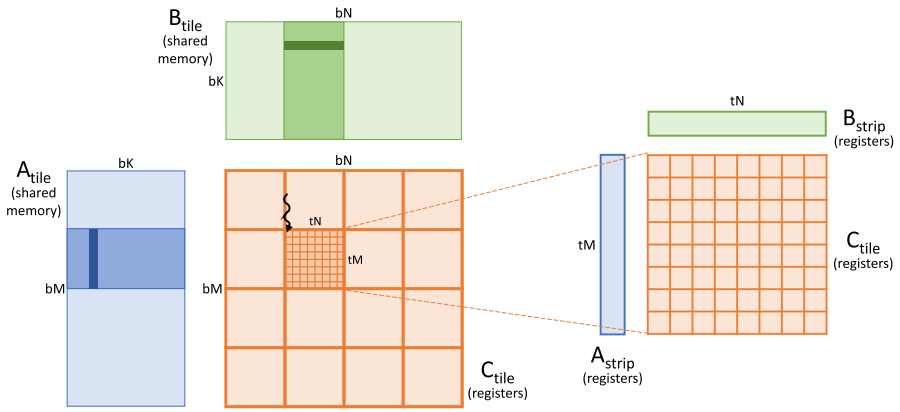


FIGURE 15.8

Matrix multiplication with register tiling of input tiles.

it multiple times from shared memory. Indeed, in the example of tiled matrix multiplication, an element in the input tile is loaded by one thread from global memory to shared memory, and then loaded by multiple threads from shared memory to their registers to perform the computation. While accessing data from shared memory is much faster than accessing it from global memory, it is still slower than accessing it from a register. Hence, after applying shared memory tiling, the shared memory access latency becomes a new bottleneck in the computation.

To overcome the shared memory access latency bottleneck, we observe from Fig. 15.2 that a thread uses the same input value to compute multiple output elements. In our implementation in Section 15.3, the same input value is loaded from shared memory multiple times by the same thread, once for each output element that it contributes to. That is, although each thread defines its own logical thread-level input tile, each time a thread uses a thread-level input tile element, the thread still loads the element from the block-level tile array in shared memory. We can avoid these redundant accesses to shared memory by applying register tiling to the input tiles. In other words, a thread can load each input element it needs from shared memory only once and place it in a register, then reuse that value from the register to compute all the corresponding output values.

Fig. 15.8 illustrates how register tiling can be applied to the input tiles. In this figure, a tile of matrix A has already been loaded into shared memory along with the corresponding tile of matrix B. Each thread iterates over its sections of the shared memory input tiles one strip at a time. The thread loads the pair of strips from the shared memory to its registers. The thread then computes the contributions of the pair of input strips to all its output tile elements before proceeding to load the next pair.

Fig. 15.9 shows the revised `mm()` device function for applying register tiling to the input tiles. Compared to the previous implementation in Fig. 15.6, this implementation moves the loop over the inner dimensions of the input tiles to be the outermost

```
01    __device__ __forceinline__ void mm(unsigned int m, unsigned int n, unsigned int k,
02        float* a, unsigned int lda, float* b, unsigned int ldb, float c[][tN]) {
03
04        #pragma unroll
05        for(unsigned int i = 0; i < k; ++i) {
06
07            // Load A strip to registers
08            float a_r[tM];
09            #pragma unroll
10            for(unsigned int row = 0; row < m; ++row) {
11                a_r[row] = a[row*lda + i];
12            }
13
14            // Load B strip to registers
15            float b_r[tN];
16            #pragma unroll
17            for(unsigned int col = 0; col < n; ++col) {
18                b_r[col] = b[i*ldb + col];
19            }
20
21            // Compute with strips
22            #pragma unroll
23            for(unsigned int row = 0; row < m; ++row) {
24                #pragma unroll
25                for(unsigned int col = 0; col < n; ++col) {
26                    c[row][col] += a_r[row]*b_r[col];
27                }
28            }
29
30        }
31
32    }
```

FIGURE 15.9

Code for register tiling of input tiles.

loop (line 05). This loop interchange effectively causes the thread to iterate over the input strips one pair of strips at a time. It ensures that the strips are used to contribute to all the output tile elements before the next strips are loaded. With this design, we refine the thread-level input tiles into these strips of `A` and `B`. By restricting the thread-level input tiles to these strips, we keep the thread-level input tiles small enough to fit into the registers of each thread.

For each iteration of the new outermost loop (line 05), the thread declares local arrays `a_r` (line 08) and `b_r` (line 15) to place the strips (thread-level input tiles). The subsequent loops (lines 10-12 and lines 17-19) load the strips from shared memory to registers. These loops are fully unrolled (lines 09 and 16) to ensure that the arrays `a_r` and `b_r` are accessed with constant indices and promoted to registers.

With the input data in the registers, the final loop nest iterates through the thread-level output tile and computes the contribution of the input strips to each element in the thread-level output tile (lines 22-28). These loops are also unrolled to ensure that the input and output tiles are all placed in registers.

By reducing the number of times input values are loaded from shared memory to registers, register tiling mitigates the shared memory access latency and instruction execution bottleneck. In fact, shared memory tiling and register tiling are not the only forms of tiling. Tiling is a general optimization can be done at any level in the memory

hierarchy where there is a difference in access latency and throughput between two memory structures. The *hierarchical technique*, as illustrated in Fig. 15.8, properly integrates block-level tiles in shared memory and the thread-level tiles in registers. For very large matrices, such hierarchical tiling techniques may also be extended to integrate tiles across multiple GPUs or even across multiple nodes in a compute cluster when a large-scale matrix multiplication is performed across multiple GPUs and even multiple nodes.

15.5 Coalesced storing of the output tile

In our implementation so far, each thread has been responsible for a large (e.g. 8×8 in Fig. 15.2) thread-level output tile. Using large thread-level output tiles has the advantage of enabling significant data reuse when shared memory tiling and register tiling are applied. However, the large 8×8 thread-level output tile is problematic when we need to store the output tile from the registers to the global memory. In particular, when each thread iterates through its 8×8 thread-level output tile to store it to global memory, threads in the same warp storing adjacent thread-level output tiles issue stores to memory locations that are eight elements apart. The resulting memory access pattern is not coalesced.

Using vector store instructions, as mentioned in Section 15.3, partially alleviates this problem because it allows each thread to store four consecutive elements simultaneously. Still, even with vector stores, the four consecutive elements cover only half the width of the thread's 8×8 output tile. Threads in the same warp storing adjacent output tiles would issue vector store instructions that are four elements apart. The resulting memory access pattern is still not completely coalesced.

To address this problem, we would like threads in the same warp to be responsible for adjacent output tiles that are only 4×4 in size. This size would allow each of these threads to issue vector stores of four-element chunks that are all adjacent to each other, resulting in a fully coalesced store transaction. To achieve this goal, we replace each thread's 8×8 output tile with four different 4×4 output tiles.

Fig. 15.10 shows how the thread-level output tiles can be rearranged such that threads in the same warp process adjacent 4×4 output tiles. On the left side of the figure, the encompassing block-level output tile is logically partitioned across the different warps in the block. One can think of each partition as a warp-level output tile. For example, assume that we have a 128×128 block-level output tile and a block of 256 threads or eight warps. The warps of the block are organized into a 2×4 arrangement, and each warp takes a 64×32 sub-tile of the block-level output tile. We will refer to each one of these 64×32 sub-tiles as a warp-level output tile.

In the middle of Fig. 15.10, the warp-level output tile is logically partitioned into four quadrants, and each thread in the warp takes a 4 × 4 sub-tile from each quadrant. For example, if the warp-level output tile's size is 64×32, then each quadrant's size is 32×16. If we organize the 32 threads in the warp into an 8×4 arrangement, then each thread receives a 4×4 output tile from each quadrant. The right side of Fig. 15.10

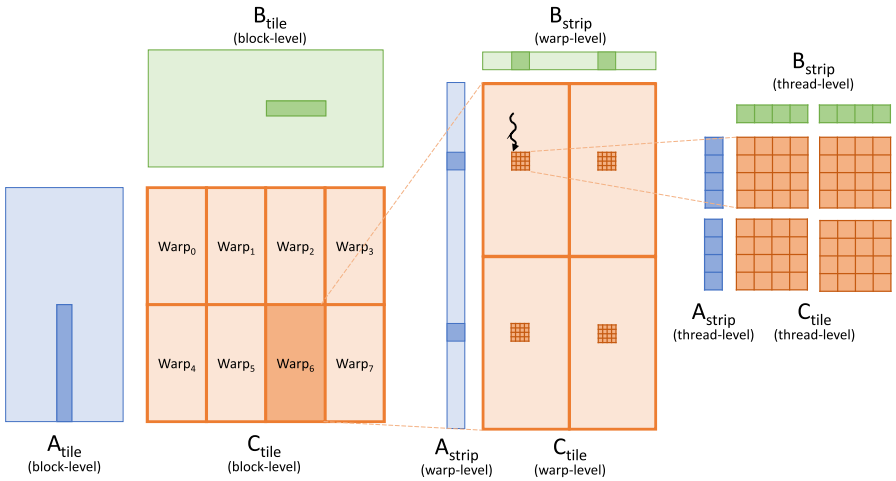


FIGURE 15.10

Rearranging thread output tiles for coalesced storing.

shows how the four 4×4 output tiles from each quadrant collectively form a logical 8×8 thread-level output tile.

Having divided the thread's 8×8 logical output tile into four 4×4 physical output sub-tiles, each thread can now store an entire row of its 4×4 output tile in a single vector store instruction. Accordingly, when threads in the same warp store adjacent physical output sub-tiles to global memory, these vector stores will be coalesced. The `mm()` and `writeTile()` device functions need to be revised to iterate through these 4×4 sub-tiles of the thread-level output tile and load the small input strips in each step. We leave the implementation of this rearrangement of the thread-level output tiles as an exercise for the reader.

15.6 Eliminating bank conflicts

Another challenge with making threads responsible for computing large output tiles is that it causes them to access shared memory in a strided manner. Recall from Chapter 6 that threads accessing shared memory in a strided manner can cause bank conflicts. Let us examine how threads in the same warp access shared memory and how bank conflicts can occur in our running example in Fig. 15.10. Recall that each warp is organized into an 8×4 configuration so the first 4 threads will be loading from identical locations of the block-level tile of `A` in the shared memory.

Fig. 15.11(a) shows how the threads in warp 0 access shared memory when loading the first input strip from the tile of the matrix `A`. Each thread in the warp takes a 4×4 output tile. The first element loaded by threads 0 to 3 is the first element of row 0. This element has a linear index of 0, which we will assume is located in bank 0.

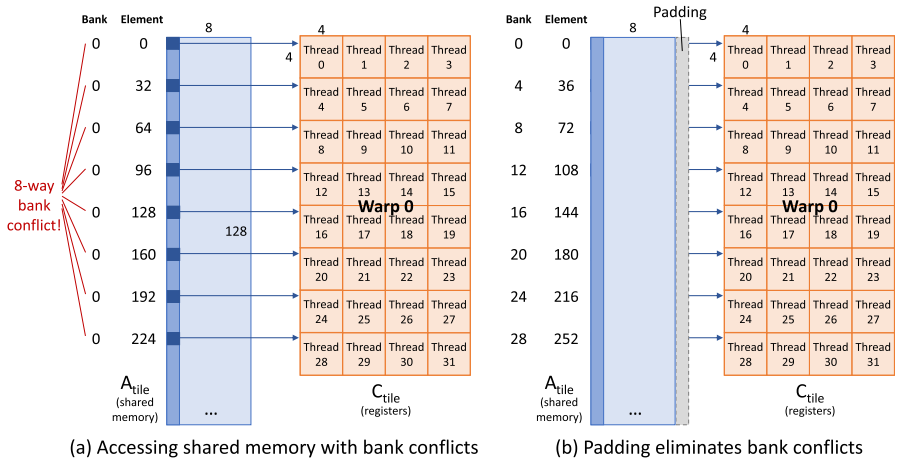


FIGURE 15.11

Adding padding to eliminate shared memory bank conflicts.

The first element loaded by threads 4 to 7 is the first element of row 4. Assuming the block-level input tile in shared memory has a width of 8, the linear index of this element is $4 \times 8 + 0 = 32$, which is also located in bank 0 (32 mod 32). Similarly, the first element loaded by threads 8 to 11 is the first element of row 8, which has a linear index of $8 \times 8 + 0 = 64$ and is also located in bank 0 (64 mod 32). Although we have focused only on the first elements loaded by the four-thread groups, the second elements loaded by the four-thread groups in each warp all fall into bank 1. In fact, all the corresponding elements loaded by all the 8 four-thread groups are located in the same bank, causing 8-way bank conflicts for all accesses.

As we discussed in Chapter 6, one way to eliminate bank conflicts is to rearrange the layout of the data in shared memory, padding a matrix with additional columns. Fig. 15.11(b) shows what happens when a column of padding elements is added to the shared memory input tile. Although the logical width of the tile is still eight elements, its leading dimension used to calculate the linear index is nine elements. In this case, the first element of row 0 still has a linear index of 0 which is located in bank 0. However, the first element of row 4 has a linear index of $4 \times 9 + 0 = 36$ which is located in bank 4 (36 mod 32). Similarly, the first element of row 8 has a linear index of $8 \times 9 + 0 = 72$ which is located in bank 8 (72 mod 32). The same analysis can be applied to the second elements accessed by the four-thread groups, which fall in bank 1, bank 5, bank 9, ..., and bank 29. In other words, the 8-way bank conflicts are eliminated. The reader can complete a general analysis of all the 16 elements accessed by each 4-thread group and verify that with 32 banks, all 8-way bank conflicts are indeed eliminated.

The code changes to Fig. 15.3 required to apply this optimization are quite simple. To add a column of padding to the tile `A_s`, the declaration of `A_s` should be changed to:

```
__shared__ float A_s[bM*(bK + 1)];
```

In particular, `bk` is replaced with `bK + 1`. To account for this additional column of padding when indexing `A_s`, the leading dimension of `A_s` passed to the `loadTile()` and `mm()` functions should be changed from `bk` to `bk + 1`.

Note that the accesses to the second shared memory tile `B_s` do not exhibit any bank conflicts. The entire strip of `B_s` loaded by the threads in the same warp consists of 32 elements that are consecutive in shared memory, and therefore fall in 32 distinct memory banks. For this reason, we do not need to make any modifications to how `B_s` is declared and accessed.

15.7 Occupancy considerations

We have seen that using large tiles increases the arithmetic intensity of the matrix multiplication kernel which improves its performance. However, using large tiles puts pressure on the SM resources, particularly the shared memory used to hold the block-level input tiles and the register file used to hold the thread-level input strips and thread-level output tiles. The pressure on these resources has a significant impact on the occupancy of threads on the SM.

The pressure on the shared memory resources can easily be alleviated. Recall from Section 15.2 that when the first input tile has dimensions $m \times k$, the second input tile has dimensions $k \times n$, and the output tile has dimensions $m \times n$, the arithmetic intensity is $0.5 \cdot m \cdot n/(m+n)$. Interestingly, the arithmetic intensity is independent of k. Hence, the pressure on shared memory can be mitigated by using a smaller value of k without affecting the arithmetic intensity. For example, if $m = n = 128$, we can set k to 8. In this case, `A_s` and `B_s` would have sizes of 128×8 elements and 8×128 elements, respectively, which amounts to 4 KB each. This amount of shared memory usage does not limit occupancy on a modern GPU.

On the other hand, the pressure on the register file remains high. For example, if each thread is responsible for an 8×8 thread-level output tile, it needs 64 registers to store the output tile alone. If register tiling is applied to the thread-level input tiles, then each thread needs an additional eight registers for each of the input strips, bringing the total to 80 registers. This number does not even include all the other registers needed to store local variables and temporary values used throughout the computation. In fact, loop unrolling, which we have applied aggressively, further exacerbates the register usage because it increases the number of temporary values that need to be stored throughout the computation.

This high demand for registers makes the number of registers in the SM the main limiting factor for occupancy. In practice, such matrix multiplication kernels use the maximum number of registers per thread that they are allowed to use, which is 255 registers per thread on a modern GPU. With a typical SM's register file having a capacity of 64k registers, the number of threads per SM is limited to 256 threads. This limit explains why we chose our thread block size to be 256 threads per block in Section 15.3. Accordingly, each SM executes only a single thread block with 256 threads, which is only 12.5% of the SM's maximum occupancy of 2048 threads.

Although executing with such low occupancy is not usually desired, the extreme benefit of register tiling to increase arithmetic intensity makes it worthwhile. However, executing with low occupancy has two major drawbacks that need to be overcome. The first drawback is that the GPU executes fewer threads which are capable of issuing fewer memory access instructions to fully utilize the memory bandwidth. We overcome this drawback by having each thread issue larger memory accesses using vector loads and stores, as we saw in Section 15.3.

The second drawback of executing with low occupancy is that the SM has fewer threads available to tolerate long-latency instructions. We overcome this drawback with aggressive loop unrolling, as we saw in Section 15.3. In particular, as mentioned in Chapter 6, loop unrolling eliminates long-latency branch instructions and exposes more independent instructions that the compiler can schedule to increase the distance between long-latency instructions and their users. With proper loop unrolling and instruction scheduling, much of the compiled kernel code becomes a sequence of independent fused-multiply-add instructions that fully-utilize the SM's cores with little need to stall.

The remaining long-latency operations that we need to be concerned with are the memory accesses and barrier synchronizations. Normally, for a compute-bound kernel that has high arithmetic intensity, we do not need to be concerned with the latency of memory accesses because they can be overlapped with the compute instructions. For example, while one thread block on the SM waits for its input tiles to be loaded from global memory to shared memory, another thread block could be fully utilizing the SM cores with its unrolled sequence of independent fused-multiply-add instructions. However, since the occupancy in our case is limited to a single thread block per SM, this overlap does not happen. Instead, a thread block executes on an SM in two phases: a memory-bound phase where the input tiles are loaded while the compute cores are idle, and a compute-bound phase where the computation is performed while the memory access hardware is idle. In the absence of multiple thread blocks to hide the memory access latency, we must resort to software optimizations for overlapping these two phases. One such optimization is software pipelining, which is discussed in Section 15.8.

15.8 Software pipelining

We saw in Section 15.7 that in the matrix multiplication kernel we have implemented so far, a single thread block executes on each SM in two phases: a memory-bound phase and a compute-bound phase. In each phase, either the compute resources or the memory resources will be underutilized, so the speed of execution will not reach its full potential. To better utilize the GPU resources, we would like to overlap the memory-bound and compute-bound phases to hide the long latency of the memory instructions behind the execution of the compute instructions. This overlap can be achieved with software pipelining.

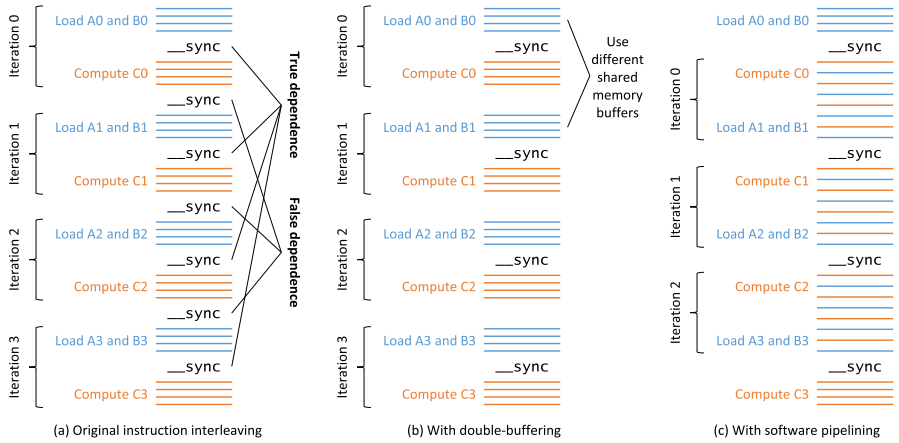


FIGURE 15.12

Instruction interleaving before and after double-buffering and software pipelining.

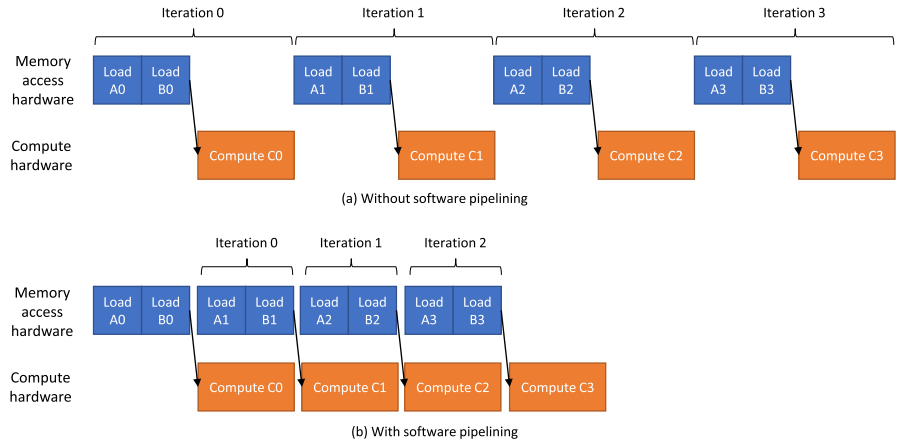


FIGURE 15.13

Timing diagram before and after software pipelining.

Fig. 15.12(a) illustrates the interleaving of instructions when performing tiled matrix multiplication without software pipelining, and Fig. 15.13(a) shows the corresponding timing diagram. First, instructions are executed to load the input tiles A0 and B0 from shared memory. Next, a `__syncthreads()` is executed to ensure that all threads finish loading their part of the data before any computation is performed. After that, the contribution of A0 and B0 to the output tile, referred to as C0, is computed. Next, another `__syncthreads()` is executed to ensure that all threads finish using A0 and B0 before loading new input tiles in their place. Following the sec-

ond synchronization, the process repeats with the next pair of input tiles, A1 and B1, loaded to shared memory.

An important observation from Fig. 15.12(a) and Fig. 15.13(a) is that while the input tiles are being loaded from global memory, there are no floating point operations being performed. In other words, the memory access hardware is busy while the floating point ALUs remain idle. On the other hand, when the contribution to the output tile is being computed, there is no data being loaded from global memory. In other words, the floating point ALUs are busy while the memory access hardware remains idle.

Ideally, we would like to interleave the instructions that load the data from global memory with those that perform floating point computations. This interleaving would place independent fused-multiply-add instructions after long-latency memory access instructions, making sure that the memory access hardware and the floating point ALUs remain busy all the time. However, the compiler is prevented from interleaving the memory access instructions and the floating point arithmetic instructions together because of the presence of a barrier synchronization operation that separates the data loading and computation stages.

Looking back at Fig. 15.12(a), we observe that the barrier synchronizations that separate loading a pair of input tiles from computing those tiles' contributions enforce a true dependence. That is, we must wait for data to be loaded before we can start using it. On the other hand, the barrier synchronizations that separate computing the contribution of a pair of tiles from loading the next pair of tiles enforces a false dependence. That is, these two activities can execute independently, but they are prevented from doing so because the same memory location that holds the old pair while computing is being used to write the new pair.

Recall from Chapter 6 that we can eliminate a barrier synchronization that enforces a false dependence by using the double-buffering optimization. That is, we can use a different memory buffer for loading the next pair of input tiles from the memory buffer holding the current pair of input tiles being used for computation. By doing so, we eliminate the false dependence and no longer need the barrier synchronization, as shown in Fig. 15.12(b). Rather than allocating a different buffer for each iteration, we can reuse buffers such that only two buffers are needed: one for the even iterations and one for the odd iterations.

While eliminating the barrier synchronizations paves the way for interleaving the memory access instructions with the floating point arithmetic instructions, an obstacle still exists. The instructions to be interleaved occur in different loop iterations which prevents the compiler from interleaving them. To overcome this obstacle, we apply *software pipelining* where the code of the loop is rearranged such that the ith loop iteration computes the contribution of the ith pair of tiles while loading the $(i+1)^{th}$ pair of tiles. Upon rearranging the loop in this way, the compiler is then able to interleave the memory access and floating point arithmetic instructions, as shown in Fig. 15.12(c) and Fig. 15.13(b), thereby increasing the independent instructions scheduled consecutively and reducing the warp's need to stall.

Fig. 15.14 shows the code transformation for performing double-buffering and software pipelining on the loop from Fig. 15.3. The code starts by loading the first

```
01    // Pre-fetch first iteration tiles to shared memory
02    loadTile(Acurr_s, A, 0, ...);
03    loadTile(Bcurr_s, B, 0, ...);
04
05    __syncthreads();
06
07    // Iterate over input tiles
08    for(unsigned int tile = 0; tile < numTiles - 1; ++tile) {
09
10        // Compute with current iteration shared memory tiles
11        mm(Acurr_s, Bcurr_s, C_r, ...);
12
13        // Pre-fetch next iteration tiles to shared memory
14        loadTile(Anext_s, A, tile + 1, ...);
15        loadTile(Bnext_s, B, tile + 1, ...);
16
17       __syncthreads();
18
19       // Swap double buffers
20       float (*Atmp_s)[bK] = Acurr_s;
21       Acurr_s = Anext_s;
22       Anext_s = Atmp_s;
23       float (*Btmp_s)[bN] = Bcurr_s;
24       Bcurr_s = Bnext_s;
25       Bnext_s = Btmp_s;
26
27    }
28
29    // Compute with last iteration shared memory tiles
30    mm(Acurr_s, Bcurr_s, C_r, ...);
```

FIGURE 15.14

Code with double-buffering and software pipelining.

pair of input tiles from global memory into the shared memory buffers `Acurr_s` and `Bcurr_s` (lines 02-03) and waiting for the loading to complete (line 05). It then iterates over the input tiles (line 08), and for each pair of input tiles, it computes with the pair in `Acurr_s` and `Bcurr_s` (line 11) while loading the next pair from the global memory to the shared memory buffers `Anext_s` and `Bnext_s` (lines 14-15). Next, a barrier synchronization is performed (line 17) to ensure that all threads finish loading data in the next pair before any of them use that data for computation in the next iteration. After the barrier, the buffers are swapped (line 20-25) so that `Anext_s` and `Bnext_s` become `Acurr_s` and `Bcurr_s`, respectively, on the next iteration, and `Acurr_s` and `Bcurr_s` are recycled for the following iteration. When the loop finishes, the computation for the final pair of tiles is performed (line 30).

An important observation in this code is that since different shared memory buffers are used for holding the current pair used for computation (`Acurr_s` and `Bcurr_s`) and for loading the next pair (`Anext_s` and `Bnext_s`), no barrier synchronization is needed on line 12 between the two operations. The absence of this barrier synchronization allows the compiler to interleave the instructions in the `mm()` and `loadTile()` device functions after inlining these functions. This interleaving ensures that both compute and global memory load instructions are overlapped to keep both the floating point ALUs and the memory access hardware busy simultaneously.

While software pipelining is effective at interleaving memory access instructions with compute instructions to overlap their execution, it relies on the success

of the compiler at estimating the latency of the instructions when applying instruction scheduling. However, the compiler has limited knowledge of the conditions that the code will encounter as it executes. Alternatively, we could delegate the interleaving of memory access instructions with compute instructions to the hardware. One technique for doing so is to program some warps to perform the memory accesses while other warps to perform the computation. In this case, the hardware will automatically interleave these operations as it schedules the warps to execute in a multithreaded manner. This technique is referred to as *warp specialization*. Yet another way to delegate the interleaving to the hardware is to use specialized hardware support for transferring data between global memory and shared memory in the background. This kind of support is discussed in Section 15.9.

15.9 Specialized software and hardware support

Due to the prevalence and importance of matrix multiplication in GPU computing, it is common for hardware vendors to provide specialized libraries with highly optimized matrix multiplication routines that embody the optimizations introduced in this chapter. For example, NVIDIA provides multiple CUDA libraries such as cuBLAS [1], cuDNN [2], and CUTLASS [3] for this purpose. cuBLAS is a library that provides a GPU implementation of the standard BLAS API for basic linear algebra, including matrix multiplication. cuDNN is a library that provides routines for implementing deep neural networks on GPUs, which perform matrix multiplication underneath. CUTLASS is an open source library for implementing matrix multiplication at different levels and scales, which makes it easier to incorporate matrix multiplication as a building block in a larger kernel. More recently, programming interfaces like cuTile allow developers to write array-based code and use the compiler to perform advanced optimizations by taking advantage of the special properties of tensors. With these libraries and programming interfaces, GPU programmers typically do not implement matrix multiplication from scratch. Nevertheless, learning about the advanced optimizations applied to matrix multiplication is a useful exercise in understanding the underlying techniques, which may be applied in other contexts as well.

GPUs also provide dedicated hardware support for matrix multiplication that can be used by the compiler, programmed directly with a special API, or more commonly, used by calling routines from libraries such as cuBLAS, cuDNN, or CUTLASS. For example, beginning with the Volta architecture, some NVIDIA GPUs come with *tensor cores* which are special-purpose cores that each perform a small matrix multiplication operation in a single instruction. Reducing the number of instructions required to perform matrix multiplication eliminates bottlenecks due to limited instruction decoding and processing throughput. Furthermore, tensor cores achieve extremely high matrix multiplication throughput by taking advantage of advanced warp-level hardware features for exchanging register contents, and by using lower-precision floating-point arithmetic units.

Before the Hopper architecture, tensor cores could be programmed with the Warp Matrix Multiply and Accumulate (WMMA) instructions, which were limited to individual warps. The Hopper architecture introduced a Warp Group Matrix Multiply Accumulate (WGMMA) instruction that enables a group of warps to work together on a larger matrix multiplication using tensor cores. Moreover, WGMMA can take input operands directly from shared memory, alleviating the register pressure. The WGMMA instruction can execute asynchronously with other instructions, reducing the reliance on the compiler to schedule the instructions for overlapping the long latency instructions. Additionally, the Blackwell architecture introduces a new tensor memory (TMEM), which is an on-chip memory for storing the inputs and outputs of tensor core operations, further reducing the register pressure. Interested readers are referred to the literature for details of the design and use of WGMMA and TMEM.

As hardware support such as tensor cores elevates the compute throughput for matrix multiplication, the throughput of matrix multiplication can become limited by the rate at which data can be fetched from and stored to shared memory and global memory. While global memory bandwidth and shared memory bandwidth play fundamental roles in moving data to and from the arithmetic units, instruction processing overhead for data movement and the impact of register usage on occupancy significantly affect the rate of data movement. For example, as we saw in Section 15.5, an effective hardware support for higher data throughput is vector stores. Using an appropriate output tile configuration, each thread can store more data using the same number of instructions, reducing instruction processing overhead and improving memory coalescing. However, moving data between global memory and shared memory using vector loads and stores still requires two steps: loading the data from one memory into registers followed by storing the data from registers to the other memory. This two-step process increases the usage of the register, not to mention the instruction execution overhead.

Recent NVIDIA GPUs support `LDGSTS` instructions that load vector data directly from global memory to shared memory without going through registers. Using one instead of two instructions also reduces instruction processing overhead. LDGSTS instructions allow more effective overlap of memory accesses with computations by allowing the hardware to perform the memory access asynchronously in the background, without the need for the compiler to interleave the memory access instructions with compute instructions. The NVCC compiler can automatically detect a code sequence formed by a vector load from global memory into registers followed by vector store to the shared memory in kernel code and generate one single LDGSTS instruction. Programmers can also call the `cuda::memcpy_async` function in the libcu++ library from their kernel code to use the LDGSTS instruction. Interested readers are referred to the literature for code examples and important details such as data alignment.

Beginning with the Hopper architecture, NVIDIA GPUs also come with a *Tensor Memory Accelerator* (TMA), which supports asynchronous transfer of entire tensors between global memory and shared memory without using intermediate registers. TMA can be viewed as a generalization of the LDGSTS instruction from 1D vectors into multidimensional tenors. TMA can be used via the `cuda::memcpy_async` API, and also by calling routines from libraries such as cuBLAS, cuDNN, or CUTLASS that use it.

15.10 Summary

In this chapter, we have seen how many of the optimizations introduced in Chapter 6 can be applied in the context of matrix multiplication to achieve an advanced and highly optimized matrix multiplication kernel. We observed that using large output tiles increases the arithmetic intensity of the kernel to make it compute bound, and applied thread coarsening to achieve these large output tile sizes. After overcoming the global memory bandwidth bottleneck, we applied register tiling to the input tiles to also overcome the shared memory access latency. We rearranged the mapping of threads to data to ensure coalesced store operations when writing the output tiles from registers to global memory. We also added padding to one of the shared memory input tiles to avoid bank conflicts.

We observed that the excessive use of registers results in very low occupancy, and applied mitigations to overcome the drawbacks of low occupancy. Using vector loads and stores ensures that threads issue enough memory accesses to utilize the memory bandwidth. Aggressive loop unrolling ensures that cores have enough independent compute instructions to utilize the floating point ALU throughput. Software pipelining or warp specialization ensure that the memory access and compute instructions are interleaved such that the latency of the memory access instructions can be overlapped with the execution of the compute instructions.

The optimizations discussed in this chapter are readily available in libraries such as cuBLAS, cuDNN, and CUTLASS. These libraries also use specialized hardware support such as tensor cores, tensor memory, asynchronous memory copies, and the tensor memory accelerator. Programmers typically use these highly optimized libraries and do not implement matrix multiplication from scratch. Still, matrix multiplication is an excellent use case for demonstrating many of the optimizations we have learned in this book.

15.11 Exercises

1. Modify the device function in Fig. 15.5 to use vector loads instead of scalar loads.
2. Modify the device function in Fig. 15.7 to use vector stores instead of scalar stores.
3. Reimplement the tiled matrix multiplication kernel presented in this chapter using the rearrangement of the thread-level output tiles described in Section 15.5.

References

[1] NVIDIA, cuBLAS, http://docs.nvidia.com/cuda/cublas, 2025.
[2] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, cuDNN: efficient primitives for deep learning, arXiv preprint, arXiv:1410.0759, 2014.
[3] NVIDIA, CUTLASS, https://docs.nvidia.com/cutlass, 2025.

PART

Advanced patterns and applications

3

This page intentionally left blank

CHAPTER

Dynamic programming and wavefront parallelism 16

With special contributions from Juan Gómez-Luna

CONTENTS

16.1 Dynamic programming 373
16.2 Implementation approaches 374
16.3 Wavefront patterns 376
16.4 Floyd-Warshall algorithm 377
16.5 Genome sequence alignment and Smith-Waterman algorithm 382
16.6 Wavefront parallelization: block-level tiling 384
16.7 Hyperplane transformation 389
16.8 More optimizations 397
16.9 Summary 399
16.10 Exercises 399
References 400

Wavefront parallelism is an algorithmic technique for exploiting data parallelism in applications with data dependencies along space and/or time dimensions. In this chapter, we study wavefront parallelism in the context of dynamic programming applications. Dynamic programming represents a class of problems that can be solved by dividing a problem into smaller sub-problems. The solution to the original problem can be found by assembling the solutions found for the sub-problems. Parallelizing dynamic programming algorithms on GPUs entails interesting challenges in terms of hardware utilization and synchronization across threads. In this chapter, we discuss these challenges and present software techniques that address them efficiently.

16.1 Dynamic programming

There exist problems, such as all-pair shortest paths in a graph and genomic sequence alignment, whose solutions can be recursively assembled from the solutions to smaller sub-problems. That is, finding optimal solutions to the sub-problems recursively solves the original problem optimally. Dynamic programming is a technique for recursively dividing such problems into smaller sub-problems, deriving and saving the solutions to the sub-problems, and assembling the saved sub-problem solutions into the solution to the original problem. These saved sub-problem solutions are

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00027-5
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

often used multiple times throughout the process, thus reducing redundant computation.

Problems that are amenable to dynamic programming exhibit an *optimal substructure* with *overlapping sub-problems*. That is, solutions to sub-problems can be assembled to form the solution to the original problem. Dynamic programming methods differ from divide-and-conquer methods in that the solutions to sub-problems in divide-and-conquer methods do not overlap. For example, in the merge sort algorithm (Chapter 14), the sub-problems are disjoint and they can be solved independently.

A notable example of dynamic programming is the Fibonacci series, which is generally represented by the following recursive expression:

$$F_i = F_{i-1} + F_{i-2} \tag{16.1}$$

If F_i were calculated with simple recursion, the same sub-problems would be solved over and over. For example, solving both F_i and F_{i-1} would incur the cost of solving F_{i-2}. To avoid such undesirable recomputation, dynamic programming algorithms typically store the solutions to the sub-problems in a table, i.e., *memoization* or *tabulation*, depending on the specific implementation approach, as we will discuss in Section 16.2. The stored solutions are reused throughout the process to eliminate redundant computation.

The execution of dynamic programming algorithms defines *wavefronts*, which are groups of intermediate solutions that can be calculated in parallel because there are no dependencies between them. Therefore, parallel execution within each wavefront is a natural way of parallelizing the execution of dynamic programming algorithms.

Besides the Fibonacci series, other problems that are commonly solved with dynamic programming algorithms encompass shortest path (e.g., Dijkstra algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm), chain matrix multiplication, time series (e.g., dynamic time warping algorithm), Hidden Markov models (e.g., Viterbi algorithm), etc. Bioinformatics is another relevant field where dynamic programming algorithms are of key importance, as they are used to accelerate the solution of problems such as sequence alignment (e.g., Smith-Waterman algorithm, Needleman-Wunsch algorithm), protein folding, RNA structure prediction, and protein-DNA binding [1].

16.2 Implementation approaches

There are two main implementation approaches for dynamic programming algorithms: *top-down* and *bottom-up*. Code snippets in Fig. 16.1 illustrate both approaches when applied to the Fibonacci series.

The top-down approach is based on recursive execution, where each invocation of the function to solve a sub-problem calls the function to solve smaller overlapping sub-problems (i.e., each sub-problem is possibly encountered multiple times when solving a larger sub-problem). Line 7 in the upper code snippet of Fig. 16.1 shows

```
1  // Fibonacci top-down implementation
2  #include <unordered_map>
3  int fibonacci_topdown(int n, std::unordered_map<int, int>&
       hash_table) {
4      if (hash_table.find(n) != hash_table.end()) return
           hash_table[n];
5      int result;
6      if (n != 0 && n != 1) {
7          result = fibonacci_topdown(n-1, hash_table) +
               fibonacci_topdown(n-2, hash_table);
8      }
9      else
10         result = n;
11     hash_table[n] = result;
12     return result;
13 }
```

```
1  // Fibonacci bottom-up implementation
2  #include <vector>
3  int fibonacci_bottomup(int n) {
4      if (n != 0 && n != 1) {
5          std::vector<int> table(n + 1, 0);
6          table[0] = 0;
7          table[1] = 1;
8          for (int i = 2; i <= n; i++) {
9              table[i] = table[i-1] + table[i-2];
10         }
11         return table[n];
12     }
13     else
14         return n;
15 }
```

FIGURE 16.1

Top-down and bottom-up C++ implementations of Fibonacci series.

the recursive calls. The top-down approach employs *memoization* as a way of holding intermediate results that can later be used to avoid redundant computation when the same sub-problems are encountered later. Each intermediate result is stored in an associative array (e.g., in `hash_table` in line 11). In the beginning of each invocation (line 4), the recursive function checks if the result it needs to calculate is already in the associative array. If so, it reuses the result from the hash table. Otherwise, it proceeds to either invoke the next recursion (line 7) or return the value (line 10) if the end of recursion has been reached, i.e., when `n` is either 1 or 0.

For example, the calculation of F_3 recursively invokes F_2 (first recursive call in line 7) and F_1 (second recursive call in line 7). The calculation of F_2 would recursively invoke F_1 and F_0. The next level of recursion would find $F_1 = 1$ and $F_0 = 0$ in line 10 and save them into the hash table. When the execution for F_3 makes the

second recursive call to invoke F_1, the called function would find the F_1 value in the hash table and return with this value in line 4.

Although the top-down approach is conceptually simple and matches with the mathematical formulation, it is unfortunately not efficient on GPUs for two main reasons. First, CUDA-enabled GPUs do not efficiently support deep nesting of function calls in kernel code. Device functions are typically inlined by the compiler for efficient execution. Recursive function calls cannot be inlined indefinitely, so they are typically retained as functions calls and cause significant slowdown. Second, accesses to the associative array, i.e., the `hash_table`, tend to result in uncoalesced memory accesses across threads in the same warp, which could also cause significant slowdown.

The bottom-up approach starts by solving the smaller sub-problems and tabulating their solutions, then progressively solving larger sub-problems until the desired problem is solved. The bottom-up approach is iterative, as the `for` loop in lines 8-10 of the lower code snippet of Fig. 16.1 shows. Each iteration computes a sub-problem, starting from the smallest sub-problems. The solution to every sub-problem is stored in a table (`table` in line 5). Since the loop iterates from the smaller sub-problems to the larger problems, each iteration is guaranteed to find the solutions of its sub-problems in the table.

The set of sub-problems that are solved (also, table cells that are filled) in each iteration is called a *wavefront*. For the Fibonacci series, wavefront `i` is composed of a single sub-problem F_i (stored in table cell `table[i]`). Unfortunately, having only one sub-problem in the wavefront results in sequential execution for this particular problem. Fortunately, many dynamic programming algorithms solve more complex problems where wavefronts consist of multiple sub-problems (often, the tables have multiple dimensions). Sub-problems in the same wavefront can be computed in parallel, since they do not depend on each other. Due to its parallel nature, wavefront parallelism is suitable for GPU computing, but it requires a careful pre-planning of the execution based on a precise understanding of its dynamic behavior (e.g., the wavefront size may grow or shrink) and the dependencies between sub-problems in different wavefronts. We focus on bottom-up approaches for the remainder of this chapter.

16.3 Wavefront patterns

As we have seen in the previous sections, in each iteration of a dynamic programming algorithm, the wavefront is composed of the sub-problems that can be solved in parallel, because they are independent of each other and all their smaller sub-problems have been solved. The execution of consecutive iterations (or *waves*) is serialized by enforcing data dependencies via synchronization of the parallel compute units (e.g., barriers).

Fig. 16.2 depicts four example wavefront patterns. The arrows show the variety of dependence patterns. The number on each cell indicates the wavefront number. The

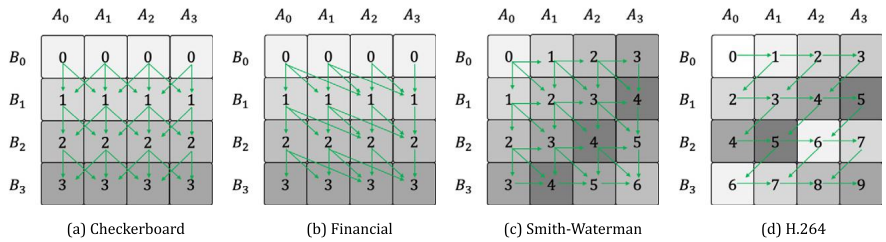


FIGURE 16.2

Example wavefront patterns.

wavefront size and the amount of parallelism can be constant (as in examples (a) and (b)) or can change (as in examples (c) and (d)) across iterations.

In this chapter, we cover examples of both types of wavefronts. Section 16.4 introduces the Floyd-Warshall algorithm and its bottom-up implementation. Floyd-Warshall is an example of dynamic programming algorithm where the wavefront has a constant size. It is similar to examples (a) and (b) of Fig. 16.2 but in each iteration, instead of computing one row of a two-dimensional space, it computes one plane of a three-dimensional space. Section 16.5 introduces the Smith-Waterman algorithm, which is widely used in genome sequence alignment. Smith-Waterman's wavefront pattern is in Fig. 16.2(c). It is an example of wavefront parallelism that grows and shrinks across iterations. Section 16.5 to Section 16.8 cover the CUDA implementation of Smith-Waterman and several optimizations that can be applied.

Another important distinction between algorithms is whether or not the entire table of sub-problem solutions needs to be retained. In some algorithms, only a certain number of wavefronts need to be retained during the execution. For example, in the Fibonacci series, when computing F_i, we only need to retain the values of F_i and F_{i-1}. The older values can be discarded (they are not discarded in the code in Fig. 16.1, but they can be). Similarly, in the Floyd-Warshall algorithm in Section 16.4, we will see that we only need to retain a single 2D plane of the 3D problem space. Older planes can be discarded because they are usually not needed in the final solution. In contrast, in the Smith-Waterman algorithm introduced in Section 16.5, the results of all iterations (all wavefronts) are saved until the end of the algorithm. The reason is that they are used to back-trace the final alignment.

16.4 Floyd-Warshall algorithm

The Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices of a directed weighted graph.[1] To understand how the problem is divided into subproblems, consider the function $d(i, j, k)$, which represents the shortest distance

[1] A graph is a data structure that represents relations between entities. For a detailed introduction to graphs, we refer the reader to Chapter 18.

between vertices i and j in a graph using only the vertices from 0 to k as intermediate vertices. Let us assume that we know $d(i, j, k-1)$ for all i and j, and for some vertex k. In this case, we can find $d(i, j, k)$ using the following recurrence relation:

$$d(i, j, k) = min(d(i, j, k-1), d(i, k, k-1) + d(k, j, k-1))$$

In other words, it is the shorter distance between going from i to j using only vertices 0 to $k-1$ without passing through k, and going from i to k then from k to j. In this way, we have decomposed the problem of finding $d(i, j, k)$ into three smaller sub-problems, namely $d(i, j, k-1)$, $d(i, k, k-1)$, and $d(k, j, k-1)$.

In the bottom up implementation of the Floyd-Warshall algorithm, we can start from the smallest sub-problems having the smallest value of k and work our way up by increasing k until we have included all vertices in the graph among the intermediate vertices. We already know the values of $d(i, j, -1)$, i.e., the distance between any two vertices i and j using no intermediate vertices. It is simply the weight of the edges between vertices i and j, or infinity if there is no edge. Using the recurrence relation, we can solve $d(i, j, 0)$ for all i and j. We can then solve for $d(i, j, 1)$ for all i and j. We can keep iterating until we have solved $d(i, j, N-1)$ for all i and j, where N is the number of vertices in the graph. The solution to $d(i, j, N-1)$ represents the shortest distance between a vertex i and a vertex j using any vertex in the graph as an intermediate vertex, which is the ultimate shortest distance we are looking for.

An important observation is that once we have found the solutions of $d(i, j, k)$ for all i and j and for some k, we no longer need the solutions for $d(i, j, k-1)$ and we can discard them. Hence, although the space of sub-problems defined by $d(i, j, k)$ is $O(N^3)$ in size, the space of sub-problem solutions that we need to store at once is only $O(N^2)$ in size. For very large graphs, the difference between $O(N^2)$ and $O(N^3)$ is quite substantial!

As the pseudo-code in Fig. 16.3 shows, the algorithm is formulated as three nested `for` loops with indices `k`, `i`, and `j`. The outer `for` loop with index `k` defines the iterations over the sub-problems of different sizes. The order of visiting the vertices `k` is irrelevant, as long as we visit all vertices `k` before we exit the loop. The `for` loops with indices `i` and `j` solve the wavefront of sub-problems and update all cells of the `dist` table. At the beginning of iteration k, the `dist` table contains the solutions for $d(i, j, k-1)$ for all i and j. At the end of the iteration, it contains the solutions for $d(i, j, k)$ for all i and j. All table cells can be computed in parallel, which makes Floyd-Warshall suitable for GPU computing.

We assume a straightforward bottom-up GPU parallelization of Floyd-Warshall algorithm that assigns one pair of vertices `(i, j)` per thread. In each iteration `k`, if the distance from vertex `i` to vertex `k` plus the distance from vertex `k` to vertex `j` is smaller than the currently observed shortest distance from vertex `i` to vertex `j`, a new shortest path from `i` to `j` has been discovered and the corresponding thread updates the value of `dist[i][j]`. At the end of each iteration, threads synchronize via kernel termination and grid relaunch. This global synchronization across all threads of the grid ensures that the possible new distances due to including a vertex `k` have been calculated and stored in the table `dist` before the next iteration begins. This way,

```
1  # Initialize nearest neighbors to actual distance, all others =
       infinity
2  initialize(dist);
3  # Order of visiting k values not important, must visit each value
4  for k in range(V):
5      # Pick all vertices as source in parallel
6      Parallel for_each i in range(V):
7          # Pick all vertices as destinations for the above picked
               source
8          Parallel for_each j in range(V):
9              # If vertex k is on the shortest path from i to j, then
                     update the value of dist[i][j]
10             dist[i][j] = min (dist[i][j], dist[i][k] + dist[k][j])
11             # dist[i][j] calculation can be parallel within each k
12             # All dist[i][j] for a single k must be computed before
                     moving to the next k
13     Synchronize
```

FIGURE 16.3

Pseudo-code of bottom-up implementation of Floyd-Warshall algorithm.

when the algorithm evaluates the distances for the next vertex `k`, all shortest distances observed so far are up to date. Thus, global synchronization is required to guarantee the correct execution of the algorithm.

Fig. 16.4 shows the CUDA kernel based on our bottom-up parallelization and a host code snippet that iteratively calls the kernel. An example of directed weighted graph and distance table is in Fig. 16.5. The distance table is initialized with the distance between adjacent vertices (all other cells equal `INFINITY`).

The host launches the kernel as many times as vertices in the graph (lines 4-5 in the bottom code snippet of Fig. 16.4). In each iteration `k`, all cells of the distance table are potentially updated by considering vertex `k` as an additional intermediate vertex. Each thread block covers a row section of the distance table. The thread grid has two dimensions of thread blocks (line 3 of the host code) and each thread block is one-dimensional (line 2). We assume that the variable `threads` in the host code specifies the number of threads in each thread block. The expression for the horizontal (x) dimension of the grid (`dimGrid`) ensures that there are enough thread blocks in the horizontal dimension to cover all sections of the columns of the distance table. The number of blocks in the vertical (y) dimension of the grid is simply the total number of vertices in the graph (`n_vertices`) because each block covers only one row of the distance table. We need `n_vertices` blocks in the vertical direction to cover all rows of the distance table.

In iteration `k`, each thread reads its assigned cell `dist[i][j]`, a cell in row `k` (`dist[k][j]`), and a cell in column `k` (`dist[i][k]`) before updating its assigned cell. In principle, in each iteration `k`, all distance table cells can be potentially updated. However, for graphs where all distances are non-negative and the distance of each vertex to itself is always 0 (i.e., `dist[k][k] = 0`), threads assigned

```
1  // Floyd-Warshall bottom-up CUDA kernel
2  __global__ void FW_bottomup(int k, int *dist, int n_vertices) {
3      // Distance table column and row
4      int col = blockIdx.x * blockDim.x + threadIdx.x;
5      if(col >= n_vertices) return;
6      int row = blockIdx.y;
7
8      // Index in dist matrix
9      int distIndex = n_vertices * row + col;
10
11     __shared__ int dist_k_col;
12     // Distance to intermediate vertex with column k
13     if(threadIdx.x == 0)
14         dist_k_col = dist[n_vertices * row + k];
15     __syncthreads();
16     if(dist_k_col == INFINITY) {
17         return;
18     }
19
20     // Distance to intermediate vertex with row k
21     int dist_k_row = dist[k * n_vertices + col];
22     if(dist_k_row == INFINITY) {
23         return;
24     }
25
26     // Update if vertex k is on the shortest path
27     int new_distance = dist_k_col + dist_k_row;
28     if(new_distance < dist[distIndex])
29         dist[distIndex] = new_distance;
30 }
```

```
1  // Host code: One kernel call per iteration k
2  dim3 dimBlock(threads);
3  dim3 dimGrid((n_vertices + threads - 1) / threads, n_vertices);
4  for(int k = 0; k < n_vertices; k++) {
5      FW_bottomup<<<dimGrid, dimBlock>>>(k, dist, n_vertices);
6      cudaDeviceSynchronize();
7  }
```

FIGURE 16.4

Bottom-up CUDA implementation of Floyd-Warshall algorithm.

to row `k` or column `k` do not modify their cells. The reason is that threads assigned to update the `dist[i][k]` and `dist[k][j]` values compute their new values as, respectively, `min(dist[i][k], dist[i][k] + dist[k][k])` and `min(dist[k][j], dist[k][k] + dist[k][j])`. Since `dist[k][k] = 0`, the expressions are simplified to `min(dist[i][k], dist[i][k])` and `min(dist[k][j], dist[k][j])`. Therefore, the values of `dist[i][k]` and `dist[k][j]` are the same before and after the execution of an iteration. As a result, there is no possible hazard and we can use a single distance table `dist`. Recall that, in other problems where there is a potential race condition cause by threads updating values that are read by other threads, we would need to use the double buffering approach introduced in Chapter 6.

The kernel (upper code of Fig. 16.4) assigns one cell of `dist` to a thread. Lines 4 and 6 of the kernel obtain the row and column values of the table cell assigned to each

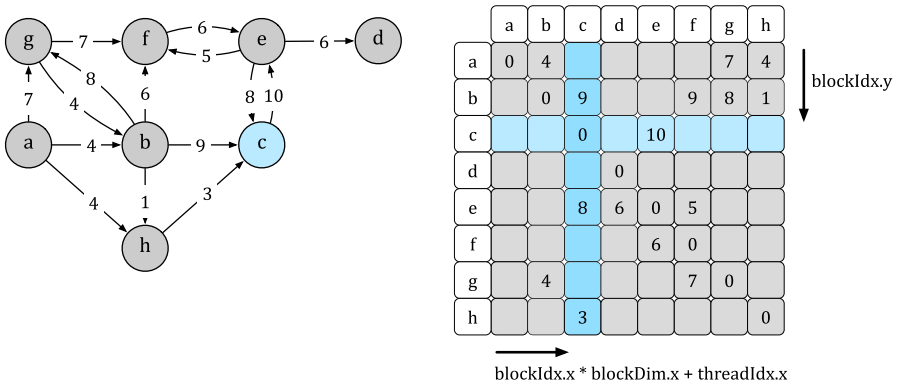


FIGURE 16.5

Directed weighted graph (left) and distance table after initialization (right).

thread. Recall that each thread block covers a row section of the distance table. Thus, all threads in a block use the y component of their common block index (`blockIdx.y`) as the row index of their assigned elements. As for the column index, each thread uses the familiar linearized index `blockIdx.x * blockDim.x + threadIdx.x` as the column index of its assigned element, which ensures that every cell in any given row is assigned to a thread. Line 9 computes the one-dimensional index into the table of the cell assigned to each thread, assuming row-major layout.

All threads in the same block need to read the same `dist` cell located in column `k` and their assigned row, i.e., `row = blockIdx.y`. In the example of Fig. 16.5, if `k` corresponds to vertex *c*, all threads of a block need the value of the cell in their row and column *c*. Thus, thread 0 reads the value from global memory and writes it in `dist_k_col` in shared memory (lines 11-15). If the distance value is infinity (line 16), vertex `k` is not on the shortest path and threads can retire. Otherwise, each thread reads the value of `dist` located in their column `col` of row `k` (line 21), e.g., thread 4 accesses column *e* in row *c* in Fig. 16.5, and stores it in `dist_k_row`. These accesses to global memory are coalesced. If the distance `dist_k_row` for any of the threads is infinity (line 22), vertex `k` is not on the shortest path and the thread returns. Finally, each remaining thread computes the new distance (i.e., the distance from vertex `row` to vertex `k` plus the distance from vertex `k` to vertex `col`) and compares it to the current distance (lines 27-28). If the new distance is shorter than the current distance, the new distance is written to `dist` (line 29).

Floyd-Warshall can be parallelized on GPU in a straightforward way because of its regularity: (1) all vertex pairs are considered in each iteration so the wavefront size remains constant, and (2) dependencies are simple (all cells depend on vertex `k`, i.e., row `k` and column `k` from the previous iteration). However, other dynamic programming problems, when parallelized, may impose more complicated wavefront patterns where the wavefront size changes across iterations. The Smith-Waterman algorithm that we introduce in Section 16.5 is an example of such wavefront patterns.

16.5 Genome sequence alignment and Smith-Waterman algorithm

A genome is the complete set of an organism's genetic instructions. The DNA is a long string of base pairs (characters): A, C, G, T. For example, the human genome contains 3.2 billion base pairs (bp). Genome analysis helps to understand genetic variations, predict the presence and abundance of microbes, monitor disease outbreaks, develop personalized medicine, etc. For these analyses, the DNA needs to be sequenced (obtained from biological samples). But sequencing machines provide only randomized fragments (reads), which need to be mapped to the reference genome. Depending on the sequencing technology, there are short (50-300 bp) and long reads (10 K-100 K bp).

A key step in the read mapping process is sequence alignment, which has been traditionally performed by quadratic-time dynamic programming algorithms such as Smith-Waterman and Needleman-Wunsch. The main data structure of the Smith-Waterman algorithm is a dynamic programming table H, called the *homology score matrix*, or *scoring matrix* for short. The goal of the alignment is to detect if two sequences are homologous or similar in a biological sense (e.g., they have shared ancestors). Assume that the two sequences being aligned, A and B, have L_A and L_B base pairs, respectively. The scoring matrix would be a $(L_A + 1) \times (L_B + 1)$ matrix. Within each sequence, we number the base pairs from 1 to the length of the sequence. All cells in row 0 and column 0 of the scoring matrix store the boundary value 0. The 0's in cells in row 0 indicate no desirable alignment has been found between the non-existent base pair 0 of B to any of the A base pairs. The 0's in cells in column 0 indicate no desirable alignment has been found between the non-existent base pair 0 of A to any of the B base pairs. Any other cell of the scoring matrix $H_{i,j}$ represents the homology score of the alignment between a segment of sequence A (from its first base pair to its i-th base pair) and a segment of sequence B (from its first base pair to its j-th base pair). The higher the homology score, the more similar the two sequence segments are. Note that, as i and j grow, computing $H_{i,j}$ becomes a sub-problem that can be solved by leveraging previously computed sub-problems, i.e., $H_{i,j}$ for smaller i and/or j. Thus, the purpose of the scoring table is to store the solutions to previous sub-problems.

For any two elements of A and B that we compare individually, we can obtain a *similarity score* $S_{i,j}$. The similarity score is positive (e.g., $+3$), if there is a $match$ (i.e., the two elements A_i and B_j are equal), or negative (e.g., -3), if there is a $mismatch$ (i.e., the two elements A_i and B_j are not equal). When there is a deletion or an insertion, we apply a *gap penalty* (e.g., 2), which we call $deletion\_penalty$ or $insertion\_penalty$, respectively. With these parameters, we can calculate every cell of the scoring matrix H as follows:

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - deletion\_penalty \\ H_{i,j-1} - insertion\_penalty \\ 0 \end{cases} \tag{16.2}$$

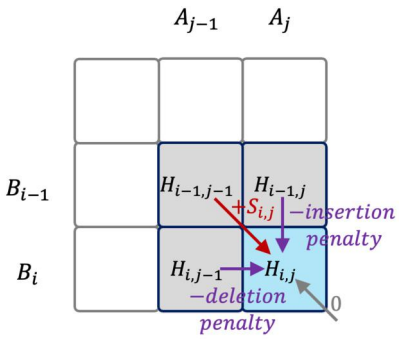


FIGURE 16.6

Dependencies in the scoring matrix of the Smith-Waterman algorithm.

Fig. 16.6 illustrates the dependencies of a cell $H_{i,j}$ of the scoring matrix. The equation assumes that the mapping process starts at the beginning of both sequences. Cell $H_{i,j}$ indicates the homology score for base pair B_i to map to A_j. Intuitively, cell $H_{i,j}$ represents the similarity of the segment of B from its beginning to its ith base pair (B_i) and the segment of A from its beginning to its jth base pair (A_j). The homology score for base pair B_i to map to A_j is evaluated for three scenarios. The first scenario is that B_{i-1} is mapped to A_{j-1}. The homology score for the scenario is previously calculated and stored into $H_{i-1,j-1}$. In this case, the homology score for B_i to map to A_j is higher if the two base pairs are the same ($S_{i,j} = +3$) and lower if the two base pairs are different ($S_{i,j} = -3$). Thus, the homology score for mapping B_i to A_j is estimated as $H_{i-1,j-1} + S_{i,j}$ in this scenario.

The second scenario is that B_{i-1} is mapped to A_j. The homology score for this scenario is assumed to have been previously calculated and stored in $H_{i-1,j}$. In this scenario, for B_i to also map to A_j, there must be a insertion in sequence B. That is, the B_j was incorrectly inserted by the sequencing process. Since an insertion implies that there is an error in the sequencing process for B, the homology score should be lowered by subtracting an insertion penalty from $H_{i-1,j}$. Thus, the homology score for mapping B_i to A_j is $H_{i,j} = H_{i-1,j} - insertion\_penalty$.

The third scenario is that B_i is mapped to A_{j-1}. The homology score for this scenario has been previously calculated and stored in $H_{i,j-1}$. In this scenario, for B_i to also map to A_j, there must be a base pair missing in sequence B. That is, a base pair was incorrectly deleted from sequence B by the sequencing process. Since a deletion implies that there is an error in the sequencing process for B, the homology score should be lowered by subtracting a deletion penalty from $H_{i,j-1}$. Thus, the homology score for mapping B_i to A_j is $H_{i,j} = H_{i,j-1} - deletion\_penalty$.

If all the three scenarios lead to negative values for $H_{i,j}$, we set $H_{i,j}$ to 0 so that all cells of the scoring matrix remain non-negative. As the figure shows, the computation of each matrix cell is only dependent on the three neighbor cells in the previous two anti-diagonals. All cells in the same anti-diagonal are independent of

each other and can be computed in parallel. Thus, each anti-diagonal of the scoring matrix is a wavefront. For example, for the first wavefront, we can only calculate $H_{1,1}$. It will use $H_{0,1}$, $H_{0,0}$ and $H_{1,0}$ that are all boundary condition cells initialized to 0. For the second wavefront, we can calculate $H_{2,1}$ and $H_{1,2}$ based on $H_{1,1}$ as well as boundary cells $H_{0,1}$, $H_{0,2}$, $H_{1,0}$, and $H_{2,0}$. For the third wavefront, we can calculate $H_{3,1}$, $H_{2,2}$, and $H_{1,3}$. Each wavefront forms an anti-diagonal of the matrix. One can observe that the size of the wavefront first grows and then shrinks from the first to the last anti-diagonal of the scoring matrix.

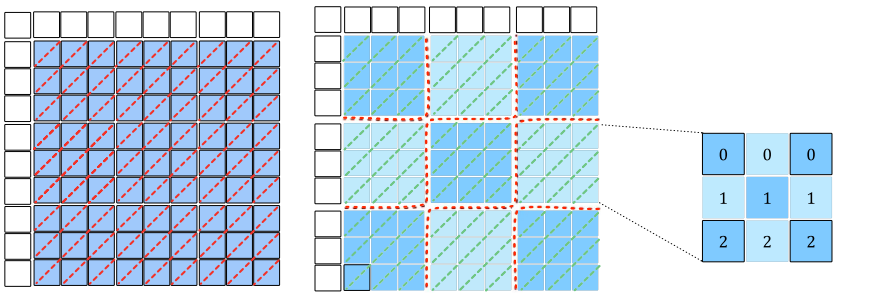
The entire scoring matrix of the Smith-Waterman algorithm is kept until all wavefronts are calculated. The reason is that the entire matrix is needed for the final *traceback* process, which starts from the cell with the highest homology score and follows the path of predecessor cells (top, top-left, or left) until reaching a score 0 (in the first row or column). The complete path reconstructs the optimal alignment between the two sequences. The traceback is a sequential process, so in this chapter we focus on the computation of wavefronts, which offers interesting parallelization opportunities.

16.6 Wavefront parallelization: block-level tiling

A basic GPU parallelization of the Smith-Waterman algorithm launches one kernel for each anti-diagonal wavefront, and each GPU thread computes one anti-diagonal cell. This approach computes all elements of each wavefront in parallel and guarantees that an entire wavefront is completely computed before the next wavefront starts. Fig. 16.7(a) shows the scoring matrix with dashed diagonal lines over the anti-diagonals representing successive kernel launches. We leave the implementation of this basic parallelization as an exercise. This implementation does not exploit data locality across adjacent anti-diagonals, wastes reuse opportunities, and has to pay the cost of launching a new kernel per anti-diagonal.

A more efficient parallelization strategy uses block-level tiling, as shown in Fig. 16.7(b). We divide the dynamic programming matrix into tiles and assign each tile to a thread block. Inside a tile, threads compute anti-diagonal entries and synchronize locally (i.e., `__syncthreads()`) before moving to the next anti-diagonal (dashed diagonal lines). As a result, global synchronization (i.e., kernel termination) is only necessary between "anti-diagonals of tiles" (dashed horizontal and vertical lines). An additional advantage of tiling is that we can use the shared memory of each streaming multiprocessor to store the tiles. After a tile is computed, threads can store the tile into global memory in a coalesced manner.

Fig. 16.8 shows the host code of a block-level tiling CUDA implementation. We make the simplifying assumption that the input sequences are of the same length (`L_seq`). Thus, each dimension of the scoring matrix is of length `L = L_seq + 1` (line 2), and the scoring matrix is an `L` × `L` square matrix (the first row and the first column of the table contain zeros). Each thread block is organized as a one-dimensional array of threads. The host code assumes that the number of threads is specified in the



(a) Basic wavefront parallelization (b) Thread block-level tiling (c) Thread assignment inside a tile

FIGURE 16.7

Basic wavefront parallelization (a), thread-block-level (tile-based) parallelization (b), and thread assignment inside a tile (c). The dashed horizontal and vertical lines represent global synchronization points (i.e., kernel termination). The dashed diagonal lines represent local synchronization points (i.e., `__syncthreads()`). The first row and the first column of the tables are boundary cells, and they are all zeros.

```
1  // Length of scoring matrix side
2  int L = L_seq + 1;
3  // Number of tiles in x dimension
4  int numTiles_x = (L_seq + threads - 1) / threads;
5  // Max blocks per antidiagonal
6  int numBlocks = numTiles_x;
7  // Loop over anti-diagonals of tiles
8  for(unsigned int d = 0; d < 2 * numTiles_x - 1; d++){
9    // Kernel call
10   sw_kernel_square<<<numBlocks, threads,
11                   threads * threads * sizeof(int)>>>
12                   (sw, rea, ref, L, d);
13 }
```

FIGURE 16.8

Thread block-level tiling implementation (host code).

`threads` variable. The scoring matrix (excluding the first row and first column of zeros) is partitioned into `threads` × `threads` tiles, each of which will be computed by a thread block. The number of tiles in the horizontal i.e., (`x`) dimension is calculated in line 4 and assigned to variable `numTiles_x`. Since the number of cells in the horizontal dimension that need to be computed is `L_seq` (column 0 cells are boundary cells that do not need to be computed) which may not be a multiple of the thread block size, the familiar expression `(L_seq + threads - 1) / threads` in line 4 ensures that there are enough blocks to cover all the tiles. As illustrated in Fig. 16.7(b), the number of tiles in the vertical dimension and the maximal number of blocks in the anti-diagonals are the same as in the horizontal dimension and are therefore also specified by the variable `numTiles_x`.

The tiles are further organized as tile anti-diagonals. For example, in Fig. 16.7(b), the first tile anti-diagonal (tile anti-diagonal 0) is the tile at the upper left corner of the matrix. The second tile anti-diagonal (tile anti-diagonal 1) consists of the two light-colored tiles next to the first anti-diagonal. There are a total of 5 tile anti-diagonals in our example and we can number them from 0 to 4. The kernel is designed to have each thread block to process one tile. The host code loops over tile anti-diagonals (line 8) with successive kernel calls (line 10). For simplicity, we use the maximal number of blocks needed (i.e., `numTiles_x`) for all kernel calls (line 6). However, not all thread blocks will be active during the execution of the kernel, as we shall see below. The reader is encouraged to explore, as an exercise, how to set `numBlocks` to the exact number of blocks needed to calculate each tile anti-diagonal.

The kernel receives the scoring matrix `sw`, two input sequences `rea` and `ref`, the length of the scoring table side `L`, and the tile anti-diagonal number `d` as arguments. A total of `2*numTiles_x-1` kernel calls will be made to process all the tile anti-diagonals. The tile anti-diagonal number used in the kernel calls will range from 0 to `2*numTiles_x-2`, e.g., 0 to 4 in Fig. 16.7(b) where `numTiles_x` is equal to 3.

The CUDA kernel is presented in Fig. 16.9. In this kernel, each thread block of `tile_width` threads (line 3) computes a square tile of `tile_width` × `tile_width` cells of the scoring matrix `sw`. In the beginning of the kernel, each thread computes the total number of tiles in each dimension of the scoring matrix (line 4). We call it `numTiles_x` because, with equally long sequences, the number of tiles in both horizontal and vertical dimensions as well as the maximal tile anti-diagonal are all the same. This number determines the number of active blocks needed. For example, in Fig. 16.7(b), the total number of tiles is 3 in horizontal, vertical, and anti-diagonal dimensions. Note that, for the calculation of the number of tiles, we only consider `(L-1)` × `(L-1)` cells of the table, because row 0 and column 0 of the table are all zeros, which we do not need to compute.

In lines 6-7, each thread determines the row and column indices of the anti-diagonal tile assigned to its thread block. Note that the thread block index `blockIdx.x` determines the row of tiles where each thread block works (line 6). That is, the assignment of tiles to thread blocks in a tile anti-diagonal starts with the top tile. We will use the processing of tile anti-diagonal 1 (`d=1`) in Fig. 16.7(b) as a running example. In this example, tile (0, 1) in tile row 0 is assigned to block 0 (`blockIdx.x=0`) and tile (1, 0) in tile row 1 is assigned to block 1 (`blockIdx.x=1`). The row index of the tile is the same as the `blockIdx.x`.

The tile column index is a function of the tile anti-diagonal number `d` (line 7). Note that the beginning tile of each tile anti-diagonal has a tile column index d. In our running example, the beginning tiles of tile anti-diagonals 0, 1, and 2 are tiles (0, 0), (0, 1), and (0, 2). Within each tile anti-diagonal, the tile column index for each tile decreases as the `blockIdx.x` of its assigned block increases. That is, for block anti-diagonal 1, block 0 is responsible for tile (0, 1) and block 1 for tile (1, 0). The tile column number for each block is `d - blockIdx.x`. If the column index for a block is negative or larger than or equal to `numTiles_x`, the thread block will not be active in the corresponding `d` iteration (line 8). In this case, the block has been assigned to

```
1  __global__ void sw_kernel_square(int* sw, T* rea, T* ref, unsigned
       int L, unsigned int d) {
2    extern __shared__ int swTile[];
3    const int tile_width = blockDim.x;
4    const int numTiles_x = (L-1+tile_width-1)/tile_width;
5    // Tile indices
6    const int tile_row = blockIdx.x;
7    const int tile_col = d - blockIdx.x;
8    if(tile_col >= 0 && tile_col < numTiles_x){
9      // Iterate over anti-diagonals of the tile
10     for(int d_tile=0; d_tile<2*tile_width-1; d_tile++){
11       // Row indices in tile and global memory
12       int r_tile = threadIdx.x;
13       int r = tile_width * tile_row + r_tile + 1;
14       // Column indices in tile and global memory
15       int q_tile = d_tile - threadIdx.x;
16       int q = tile_width * tile_col + q_tile + 1;
17       // Bound checking
18       if(q_tile>=0 && q_tile<tile_width && r<L && q<L){
19         // Load from the previous two anti-diagonals
20         int n = load_n(sw, r, q, L, swTile, r_tile, q_tile,
               tile_width);
21         int w = load_w(sw, r, q, L, swTile, r_tile, q_tile,
               tile_width);
22         int nw = load_nw(sw, r, q, L, swTile, r_tile, q_tile,
               tile_width);
23         // Similarity score
24         int subs_val = (rea[r-1] == ref[q-1]) ? MATCH : MISMATCH;
25         // Obtain maximum and store in shared memory
26         swTile[r_tile * tile_width + q_tile] = max4(0, nw +
               subs_val, w + DELETION, n + INSERTION);
27       }
28       __syncthreads(); // Thread block synchronization
29     }
30     // Store the tile in global memory
31     store_tile(sw, swTile, L, tile_width, tile_row, tile_col);
32   }
33 }
```

FIGURE 16.9

Thread block-level tiling implementation (kernel code).

a non-existent tile because the tile anti-diagonal has fewer than `numTiles_x` tiles and does not need all the launched thread blocks. For example, in Fig. 16.7, tile anti-diagonal 1 (`d=1`) has only two tiles. Thus, block 2 will have a tile column number −1, which means that it is not needed. For another example, tile anti-diagonal 3 (`d=3`) also has only 2 tiles. In this case, block 0 will have a column index value 3, which is equal to `numTiles_x` and thus out of bound. This means that block 0 will not be active; only block 1 and block 2 will be active for tile anti-diagonal 3.

The main loop of the kernel (lines 10-29) computes the anti-diagonals of the tile in `2*tile_width-1` iterations. In each iteration, each thread computes one cell of the shared memory tile with row and column indices `r_tile` (line 12) and `q_tile` (line 15), which correspond to row and column indices `r` (line 13) and `q` in global memory

```
1  __device__ inline int load_n(int* sw, int r, int q, unsigned int L,
       int* swTile, int r_tile, int q_tile, int tile_width){
2   return (r_tile == 0) ? sw[(r - 1) * L + q] : swTile[(r_tile - 1) *
        tile_width + q_tile];
3  }

1  __device__ inline int load_w(int* sw, int r, int q, unsigned int L,
       int* swTile, int r_tile, int q_tile, int tile_width){
2   return (q_tile == 0) ? sw[r * L + (q - 1)] : swTile[r_tile *
       tile_width + (q_tile - 1)];
3  }

1  __device__ inline int load_nw(int* sw, int r, int q, unsigned int L
      , int* swTile, int r_tile, int q_tile, int tile_width){
2   return (r_tile == 0 || q_tile == 0) ? sw[(r - 1) * L + (q - 1)] :
       swTile[(r_tile - 1) * tile_width + (q_tile - 1)];
3  }
```

FIGURE 16.10

Thread block-level tiling implementation: device functions to load input values.

(line 16). Note that in the calculation of `r` and `q` we add 1 to these global memory indices, because we do not need to compute the cells in row `r=0` and column `q=0`, which are all zeros.

Thread assignment to tile cells, which is represented in Fig. 16.7(c), follows the same pattern as thread block assignment to tiles. For example, in anti-diagonal 1 of the tile in Fig. 16.7(c), thread 0 is mapped to cell (0, 1) and thread 1 is mapped to cell (1, 0). Since not all threads are assigned a cell in an iteration, they need to check that they are within the tile bounds `q_tile >= 0 && q_tile < tile_width`. For example, for anti-diagonal 1, thread 2 is inactive since it is out of the tile. They also need to check if they are within the non-boundary part of the scoring matrix `r < L && q < L` (line 18). They do not need to check `r >= 0 && q >= 0`, because we add 1 when we calculate `r` and `q`.

Then, each active thread loads the values of three cells in the two previous anti-diagonals on which its assigned cell depends (lines 20-22). We use three device functions, `load_n`, `load_w`, and `load_nw`, which Fig. 16.10 shows. With these three device functions, a thread loads the cell values from shared memory, unless its assigned cell is at the top (i.e., `r_tile = 0`) or the left border of the tile (i.e., `q_tile = 0`). In such case, they are read from global memory, because they were produced in the previous kernel call (i.e., previous `d` iteration).

`n` is the value above (i.e., north of) the assigned cell, i.e., in row `r_tile-1` (and `r-1`). `w` is the value to the left (i.e., west) of the assigned cell, i.e., in column `q_tile-1` (and `q-1`). `nw` is the value above and to the left (i.e., northwest) of the assigned cell, i.e., in row `r_tile-1` (and `r-1`) and column `q_tile-1` (and `q-1`). With `n`, `w`, `nw`, and the similarity score `subs_val` (line 24), a thread can compute the value of its assigned cell (line 26). `max4()` is a device function that returns the maximum of four input values,

```
1  __device__ inline int max4(int a, int b, int c, int d){
2    int m = a;
3    if(m < b) m = b;
4    if(m < c) m = c;
5    if(m < d) m = d;
6    return m;
7  }
```

FIGURE 16.11

An example device function that computes the maximum of four integers.

```
1  __device__ inline void store_tile(int* sw, int* swTile, unsigned
       int L, int tile_width, int tile_row, int tile_col){
2    for(int row = 0; row < tile_width; row++){
3      int r = tile_width * tile_row + row + 1;
4      int q = tile_width * tile_col + threadIdx.x + 1;
5      if(r < L && q < L)
6        sw[r * L + q] = swTile[row*tile_width+threadIdx.x];
7    }
8  }
```

FIGURE 16.12

Thread block-level tiling implementation: device function to store a tile in global memory.

as Fig. 16.11 shows. At the end of each anti-diagonal iteration (line 28), all threads of a block synchronize locally to guarantee that the entire anti-diagonal is in shared memory before starting the next anti-diagonal of the tile.

After all anti-diagonals of a tile are computed, the threads call the `store_tile` device function (line 31) to write the tile into global memory row-by-row. As Fig. 16.12 shows, all threads collaborate on this and perform coalesced memory accesses. This is an example of turning uncoalesced memory accessed into coalesced memory accesses thanks to the use of shared memory. It is similar to the corner-turning technique (Chapter 6) or the packing technique (Chapter 12) that we have seen in previous chapters.

16.7 Hyperplane transformation

The thread block-level tiling technique in Fig. 16.7(b) improves the performance of wavefront parallelization by reducing kernel calling overhead and leveraging the fast shared memory. However, using square (or rectangular) tiles has two important drawbacks, as Fig. 16.13 illustrates with two adjacent tiles.

First, inside a tile, anti-diagonal lengths are non-uniform. As a result, the number of active threads is different for each anti-diagonal, as the left tile in Fig. 16.13 shows.

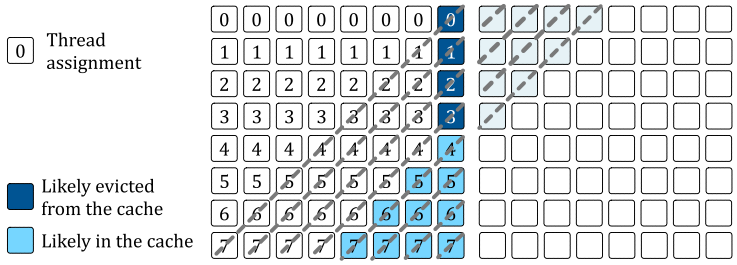


FIGURE 16.13

Drawbacks of thread block-level tiling with square tiles.

In most iterations, the observed parallelism is lower than the maximum, which implies that many threads remain idle (i.e., warps are underutilized). As a result, for a tile of $n \times n$ elements, $2 \times n - 1$ iterations need to be performed.

Second, locality between neighboring tiles that are in two different tile anti-diagonals is likely lost because only the last anti-diagonals of a tile are likely to remain in the cache (e.g., L2 in a GPU). In Fig. 16.13, the darkly shaded cells at the top right of the left tile are necessary to compute the initial anti-diagonals of the right tile (lightly shaded cells). However, it is more likely that the moderately shaded cells, which belong to the last anti-diagonals of the left tile, are in the cache since they were computed later, while the darkly shaded cells may have been evicted from the cache. Unfortunately, by the time the moderately shaded cells are used by the corresponding anti-diagonals of the right tile, they will likely have been evicted since these anti-diagonals of the right tile are computed later.

To alleviate these two drawbacks, we can transform the square (or rectangular) tiles into *hypertiles* [2–4]. This is done via an affine transformation, called *hyperplane partitioning*, which converts rectangular tiles into quadrilaterals. We will limit the transformed tiles to parallelograms that can be formed by shifting the rows of a rectangular tile horizontally so that the cells that correspond to a column in the original tile are aligned with anti-diagonals after the transformation. Such shifting is called *shear transformation*. For example, in Fig. 16.14, we transform each 3×3 square tile in (a) into a 3×3 parallelogram hypertile in (b) by shifting its middle row to the left by 1 cell and the bottom row to the left by 2 cells. In these parallelogram hypertiles, each column of the original rectangular tiles is aligned with an anti-diagonal after the transformation. For example, in Fig. 16.14, the three cells from each column of the original tiles in (a) are aligned with an anti-diagonal in the transformed tile in (b).

Parallel implementations of the Smith-Waterman algorithm using hypertiles have three advantages over those using rectangular tiles. First, the computation in each tile consists of a sequence of wavefronts of the same length. Thus, each thread block executing the kernel will have the same number of active threads when iterating through these wavefronts, which reduces warp divergence and hardware under-utilization. Fig. 16.14(c) shows that the length of all intra-tile anti-diagonals is fixed, keeping the number of active threads constant. Second, the number of iterations to compute

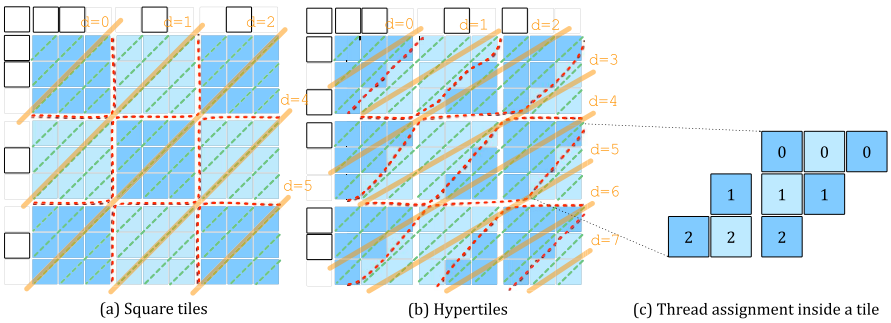


FIGURE 16.14

Hyperplane transformation.

a tile reduces, given that more work is done per iteration. Fig. 16.14 shows that five iterations are needed for a square tile, while only three for a hypertile. Third, the last anti-diagonal of one tile passes directly to the first anti-diagonal of the next tile. This increases data locality across thread blocks and can increase cache hits.

In Fig. 16.14, assume that a cell with tile-level coordinates `(r_tile, q_tile)` is mapped to score-matrix-level coordinates `(r, q)` under square tiling. After the shear transformation, the cell with the same tile-level coordinates `(r_tile, q_tile)` would be mapped to score-matrix-level coordinate `(r, q - r_tile)`. For example, cell (2, 0) of tile (0, 1), i.e., the second tile in the top row, in Fig. 16.14(a) has score-matrix level coordinate (2, 3). After shear transformation, cell (1, 0) in tile (0, 1) has score-matrix-level coordinate (2, 3-2) = (2, 1).

In general, with shear transformation, if a cell with tile-level coordinates `(r_tile, q_tile)` has score-matrix-level coordinate `(r, q)` under rectangular tiling, the cell with the same tile-level coordinate will have score-matrix-level coordinate $(r, q + m \times r_{tile})$ after shear transformation, where m is called the *shear factor*. The example in Fig. 16.14 assumes $m = -1$, the value that tilts each column of the hypertile sufficiently to align with the antidiagonals, as Fig. 16.14(b) shows.

A disadvantage of tilting the tile columns is that each "wavefront of tiles" (i.e., the tiles that can be computed simultaneously in a `d` iteration, as marked by thick diagonal lines in Fig. 16.14), differ from those in the square tile implementation. For example, in the second iteration (`d=1`), only tile (1, 1) is in the wavefront. This is because the tilted columns of tile (0, 0) computed in the first wavefront left cells (1, 2), (2, 1), and (2, 2) not covered. Since these cells are needed by tile (1, 0), tile (1, 0) has to wait for the completion of tile (0, 1) in the second iteration to finish and then be computed in the third iteration. The number of `d` iterations increases with respect to the implementation of the square tile, as the host code in Fig. 16.15 shows, due to incomplete tiles to the left and right of the scoring matrix. In Fig. 16.14, the number of d iterations increase from 5 under square tiling to 8 under hypertiling. A natural consequence of the increase in the number of iterations is that the number of tiles per iteration decreases.

```
1  // Length of scoring matrix side
2  int L = L_seq + 1;
3  // Number of tiles in x dimension
4  int numTiles_x = (L_seq + threads - 1) / threads;
5  // Blocks per antidiagonal
6  unsigned int numBlocks = numTiles_x;
7  // Loop over anti-diagonals of tiles
8  for(unsigned int d = 0; d < 3 * numTiles_x - 1; d++){
9    // Kernel call
10   sw_kernel_hyper<<<numBlocks, threads,
11                     threads * threads * sizeof(int)>>>
12                     (sw, rea, ref, L, d);
13 }
```

FIGURE 16.15

Hypertile-based implementation (host code).

Let us reason what the number of wavefronts (iterations) for the hypertile approach is, given `numTiles_x`. It takes `numTiles_x` wavefronts for the square tile approach to reach the top-right corner of the table and another `numTiles_x - 1` wavefronts to reach the bottom-right corner of the table. If we think about the thread blocks computing the tiles, each thread block is one tile column ahead of the next thread block. Hence, the thread block that computes the first row of tiles needs `numTiles_x` iterations to finish, and then we need one more iteration for each of the remaining `numTiles_x - 1` rows of tiles. Thus, the total number of wavefronts is `2 * numTiles_x - 1`. In the case of the hypertiles, it also takes `numTiles_x` wavefronts to reach the top-right corner of the table. However, the effect of the shear transformation is that the tiles tilt. As a result, the cells of the right-most column of the table in the hypertile case belong to twice the number of tiles than in the square tile case. In other words, the tilting causes each thread block to be two columns of tiles ahead of the next thread block. Hence, the thread block that computes the first row of tiles needs `numTiles_x + 1` iterations (we add one due to the last incomplete tile), and we need two more iterations for each of the remaining `numTiles_x - 1` tile rows. Thus, the total number of wavefronts is `3 * numTiles_x - 1`. In Fig. 16.14(a), we can see that the cells of the right-most column belong to three tiles. Hence, the total number of wavefronts is 5. However, in Fig. 16.14(b), we can see that they belong to six tiles. Hence, the total number of iterations is 8.

Fig. 16.16 shows the CUDA kernel with hypertiles. It follows the same structure as the square tile implementation. Thus, we focus on explaining the key differences. The tile column index `tile_col` (line 10) differs from that of the square tile implementation due to the tilted shape of tiles. We take the first row of a tile as the reference for the column index. As we can observe in Fig. 16.14, the shape of tile anti-diagonals, e.g., the third anti-diagonal (`d=2`), is different in Fig. 16.14(a) and Fig. 16.14(b). For `blockIdx.x=0`, which works on the first row of tiles (`tile_row=0`) in both implementations, the tile column index is 2 in both cases (i.e., `tile_col=d`). However, for `blockIdx.x=1` (i.e., `tile_row=1`), the tile column index is 1 in the square

```
1  #define _m (-1) // Shear factor
2  #define _shear(x, y) (x + _m * y) // Affine transformation
3
4  __global__ void sw_kernel_hyper(int* sw, T* rea, T* ref, unsigned
       int L, unsigned int d) {
5    extern __shared__ int swTile[];
6    const int tile_width = blockDim.x;
7    const int numTiles_x = (L-1+tile_width-1)/tile_width;
8    // Tile indices
9    const int tile_row = blockIdx.x;
10   const int tile_col = d - blockIdx.x * 2;
11   if(tile_col >= 0 && tile_col <= numTiles_x){
12     initialize_tile(swTile, tile_width);
13     // Iterate over anti-diagonals of the tile
14     for(int d_tile = 0; d_tile < tile_width; d_tile++){
15       // Row indices in tile and global memory
16       int r_tile = threadIdx.x;
17       int r = tile_width * tile_row + r_tile + 1;
18       // Column indices in tile and global memory
19       int q_tile = d_tile;
20       int q=tile_width*tile_col+_shear(q_tile, r_tile)+1;
21       // Bound checking
22       if(r < L && q >= 1 && q < L){
23         // Load from the previous two anti-diagonals
24         int n = load_n(sw, r, q, L, swTile, r_tile, q_tile,
               tile_width);
25         int w = load_w(sw, r, q, L, swTile, r_tile, q_tile,
               tile_width);
26         int nw = load_nw(sw, r, q, L, swTile, r_tile, q_tile,
               tile_width);
27         // Similarity score
28         int subs_val = (rea[r-1] == ref[q-1]) ? MATCH : MISMATCH;
29         // Obtain maximum and store in shared memory
30         swTile[r_tile * tile_width + q_tile] = max4(0, nw +
               subs_val, w + DELETION, n + INSERTION);
31       }
32       __syncthreads(); // Thread block synchronization
33     }
34     // Store the tile in global memory
35     store_tile(sw, swTile, L, tile_width, tile_row, tile_col);
36   }
37 }
```

FIGURE 16.16

Hypertile-based implementation (kernel code).

tile implementation (i.e., `tile_col=d-blockIdx.x`) but it is 0 in the hypertile implementation (i.e., `tile_col=d-2*blockIdx.x`).

The `for` loop to iterate over the anti-diagonals of the tile (line 14) has a bound of `tile_width` instead of `2*tile_width-1`. The number of anti-diagonals is the same as the width of the tile because all threads are active during all iterations in the hypertile implementation.

The calculation of the column indices (lines 19-20) has to take into account the shear transformation and how cell values are stored in shared memory. In our hyper-

```
1  __device__ inline int load_n(int* sw, int r, int q, unsigned int L,
       int* swTile, int r_tile, int q_tile, int tile_width){
2    return (r_tile == 0 || q_tile == 0) ? sw[(r - 1) * L + q] :
       swTile[(r_tile - 1) * tile_width + q_tile - 1];
3  }

1  __device__ inline int load_w(int* sw, int r, int q, unsigned int L,
       int* swTile, int r_tile, int q_tile, int tile_width){
2    return (q_tile == 0) ? sw[r * L + (q - 1)] : swTile[r_tile *
       tile_width + (q_tile - 1)];
3  }

1  __device__ inline int load_nw(int* sw, int r, int q, unsigned int L
     , int* swTile, int r_tile, int q_tile, int tile_width){
2    return (r_tile == 0 || q_tile == 0 || q_tile == 1) ? sw[(r - 1) *
       L + (q - 1)] : swTile[(r_tile - 1) * tile_width + (q_tile -
       2)];
3  }
```

FIGURE 16.17

Hypertile-based implementation: device functions to load input values.

tile implementation, for easier indexing, we assume that the shared memory tile is a square. Thus, each thread keeps its assigned value of anti-diagonal `d_tile` in column `q_tile=d_tile` of that square. For the column index in global memory, we apply the shear transformation $(r, q + m \times r_{tile})$, i.e., we shift the column index by as much as the row index value. In the code, the transformation is done with the `_shear()` macro (line 2).

The bound checking (line 22) only needs to check the indices in global memory `r` and `q` (for incomplete tiles), since all threads that work on a tile are active in all iterations.

The rest of the kernel code looks the same as that of the square tile implementation but the body of the device functions changes. Fig. 16.17 shows `load_n`, `load_w`, and `load_nw` functions. As is the case in the square tile implementation, if one tile cell is at the top or the left border of the tile, the thread needs to read the previous anti-diagonal values from the global memory matrix. If these values fall in the shared memory tile, we need to adjust the index calculation by taking into account how cell values are stored in shared memory. For example, `load_n` needs to read a tile cell that corresponds to a cell (`n`) directly above its assigned cell in the scoring matrix. If `n` is in the shared memory tile, its row index in the tile is `r_tile-1`, as is the case with square tiling. However, because of the sheer transformation, the cell is actually stored in the previous column (`q_tile-1`) in the tile because each hypertile is stored as a square in shared memory. We encourage the reader to reason about `load_w` and `load_nw` in a similar way.

Finally, the threads store the tile to global memory by calling `store_tile`, whose code is in Fig. 16.18. The only difference with respect to the code in Fig. 16.12 is that we apply the shear transformation to compute the column index `q` (line 4).

```
1  __device__ inline void store_tile(int* sw, int* swTile, unsigned
       int L, int tile_width, int tile_row, int tile_col){
2    for(unsigned int row = 0; row < tile_width; row++){
3      int r = tile_width * tile_row + row + 1;
4      int q=tile_width*tile_col+_shear(threadIdx.x, row)+1;
5      if(r < L && q < L)
6        sw[r * L + q] = swTile[row*tile_width+threadIdx.x];
7    }
8  }
```

FIGURE 16.18

Hypertile-based implementation: device function to store a tile in global memory.

```
1  __device__ inline void initialize_tile(int* swTile, int tile_width)
       {
2    for(unsigned int row = 0; row < tile_width; row++){
3      swTile[row * tile_width + threadIdx.x] = 0;
4    }
5    __syncthreads();
6  }
```

FIGURE 16.19

Hypertile-based implementation: device function to initialize shared memory.

We have skipped one line (line 12) in our explanation of the code in Fig. 16.16, which is absent from the square tile implementation. This `initialize_tile` device function initializes the shared memory tile cells to 0, as Fig. 16.19 shows. The reason for this initialization function is that the bound checking (line 22) will make sure that some threads do not write to their shared memory positions because they fall outside of the actual scoring matrix. However, all threads remain active during the kernel execution because their assigned cells fall in the range of the tile so some of the threads would calculate out-of-bound cells in the thread block. They will not be allowed to write to the score matrix at the end of the kernel execution. As a result, these threads could read garbage values via `load_n`, `load_w`, and `load_nw` in later iterations. Initializing all shared memory tile cells to zero prevents these threads from generating exceptions when performing using the garbage values.

There is still one challenge to solve in the hypertile implementation due to the way we store the tile in shared memory. As Fig. 16.20 illustrates, the hypertile is stored as a square in shared memory, using a row-major layout. However, if the hypertile width is a multiple or a divisor of the number of shared memory banks, bank conflicts will occur. To avoid the bank conflicts, we can use *padding* (i.e., one empty memory location after each row of the hypertile) in order to shift the addresses accessed by consecutive threads, thus avoiding the bank conflicts. In our code, we can define a macro `pad(x)` equal to `(x + (x » LOG_NUM_BANKS))`, where `x` is the shared memory index and `LOG_NUM_BANKS` is the binary logarithm of the number of banks. Note that

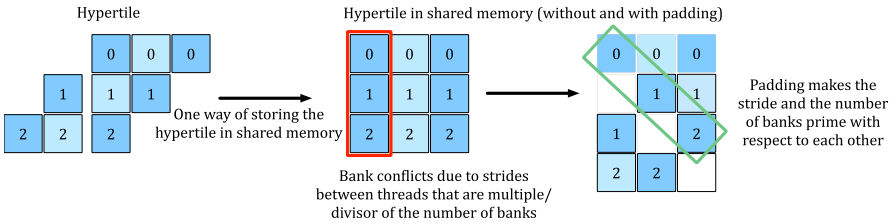


FIGURE 16.20

Example of padding a hypertile in shared memory.

we also need to allocate some extra shared memory space for the tile (i.e., `(threads * threads + ((threads * threads) » LOG_NUM_BANKS)) * sizeof(int)`) in line 11 of Fig. 16.15.

To complete our comparison of the two tiling approaches, we can do a work analysis to estimate the benefit of the hypertiles over the square tiles. A thread block performing the work of an entire square tile requires $2n - 1$ iterations, with n being the size of one tile dimension. In iteration i, the number of computed cells is i before the main anti-diagonal and $2n - i$ after the main anti-diagonal. Eq. (16.3) shows the total amount of work.

$$Work_{square\ tile} = \sum_{i=1}^{n} i + \sum_{i=n+1}^{2n-1} (2n - i) = n^2 \tag{16.3}$$

The computation of one hypertile requires only n iterations and the amount of work per iteration is n. Eq. (16.4) shows the total amount of work to compute a hypertile.

$$Work_{hypertile} = \sum_{i=1}^{n} n = n^2 \tag{16.4}$$

We can observe that the total amount of work is the same for both types of tiles. However, the square tile requires $2n - 1$ iterations, while the hypertile requires n iterations. As a result, the average work per iteration for the square tile is approximately $\frac{n}{2}$, while it is n for the hypertile. That is, the hypertile is twice as efficient as the square tile.

We also need to account for the additional iterations in the host code. The host code needs to iterate through up to $1.5\times$ more wavefronts under hypertiling. However, if the scoring matrix is large enough, the iterations may still be able to achieve good occupancy. In this case, the negative impact on performance will be from the kernel call overhead. Kernel calling overhead can be eliminated with the synchronization technique introduced in Section 16.8.

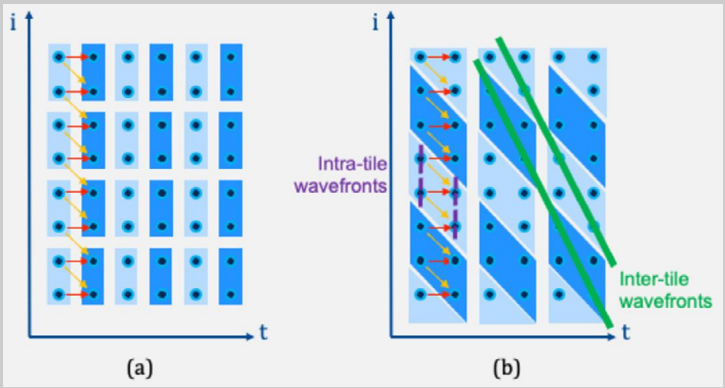
The hyperplane transformation can also apply to other parallel patterns. Stencil computation (Chapter 8) is a prominent example, as the following sidebar explains.

Hyperplane Partitioning for Stencil Computation

Stencil computation can benefit from the hypertile transformation introduced in this section. An example is Jacobi stencils, where there is an outer sequential loop that represents iterations in the time domain. The inner loop/s update the stencil cells in the spatial domain. As a result, there are intra-tile wavefronts and inter-tile wavefronts in the spatial domain and in the time domain, and it is possible to leverage the parallelism in both types of wavefronts efficiently by using hypertiles.

The following code and figures show an example of a toy one-dimensional stencil. As we can see in the code, each element of the stencil `A[i]` is dependent on its previous value and the value of the upper element (i.e., `A[i]` and `A[i+1]` calculated in the previous time iteration). Figure (a) represents the dependencies with arrows. With a straightforward work partitioning, only the elements in the same column (same t) can be calculated in parallel, assuming that all values of the previous t are available. After each t, we need to perform a global synchronization. Figure (b) shows a hypertile partitioning of the time-space domain. The hypertiles define intra-tile wavefronts and inter-tile wavefront, thus exposing more parallelism and reducing the number of global synchronizations.

```
1  for (t=1; t<=T; t++)
2    for (i=1; i<=I; i++)
3      A[i] = 0.5 * (A[i] + A[i+1]);
```



16.8 More optimizations

Synchronization across anti-diagonals of tiles. In the previous sections, we synchronize anti-diagonals of tiles by terminating the kernel and launching a new kernel.

This ensures global synchronization. However, this approach has the disadvantage that each thread block has to wait for all the thread blocks that compute the two previous anti-diagonals, although it only depends on three of these thread blocks. This may cause load imbalance and, thus, entail a potential waste of the computing slots available in the streaming multiprocessors.

Recently, CUDA provides *cooperative groups*, which can synchronize all threads in a warp, a block, several blocks, or an entire grid. Chapter 18 provides a detailed introduction to cooperative groups. An example use of cooperative groups is to allow synchronizing all thread blocks within the grid without relaunching the kernel. For grid-level synchronization, the kernel with cooperative groups needs to use persistent thread blocks. The grid size must be equal to (or less than) the maximum number of concurrent thread blocks that can run on the streaming multiprocessors. The grid-level synchronization provided by cooperative groups can be applied to the computation of tile anti-diagonals in our implementation of the Smith-Waterman algorithm. However, such an implementation would still force each thread block to wait for all thread blocks of the previous two anti-diagonals.

A more efficient synchronization mechanism can use an array of flags to signal the completion of tiles. This way, one thread block only needs to check two flags (i.e., the ones corresponding to the tile on top and the tile at the left) and can start computing as soon as they are set. Furthermore, if we use persistent thread blocks and assign one per row of tiles, each thread block will only need to check one flag. Fig. 16.21 illustrates this synchronization mechanism, which resembles the unidirectional synchronization that we covered in Chapter 11 (Scan). The left part of the figure shows an array of flags and the dependencies between tiles. The right part of the figure illustrates the waste of computing slots that the global synchronization may cause, and the potential for saving execution cycles of the inter-block synchronization mechanism.

The inter-block synchronization mechanism can also be applied to the hypertiles. In their case, the width of the hypertiles determines the amount of synchronization operations (i.e., checking and updating flags) and the waiting time for a block to start computing a tile. A programmer could tune the width in order to find a performance sweet spot.

Small dynamic programming problems. In this chapter, we have used the Smith-Waterman algorithm as a running example. Note that we assume that multiple thread blocks are needed for the computation of the scoring matrix. This would be useful for alignment of long sequences (thousands of base pairs). However, for short sequences (hundreds of base pairs), one thread block or even one warp per alignment may be enough to compute the scoring matrix. Such implementations can greatly benefit from storing tiles in registers and exchanging intermediate results using shuffle instructions (introduced in Chapter 10).

DPX instructions. Since NVIDIA Hopper architecture, CUDA features a new type of SIMD instructions, called DPX, that are intended to accelerate dynamic programming applications. For example, we can replace the `max4()` device function in the kernel of Fig. 16.10 with the DPX instruction `__vimax3_s32_relu()`, which takes advantage of special hardware to calculate the maximum of three input values and 0.

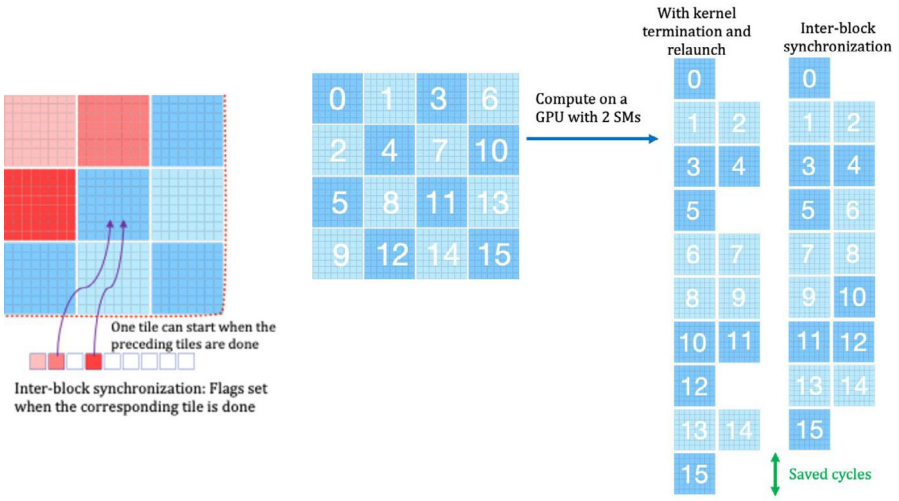


FIGURE 16.21

Inter-block synchronization (left) and its benefit in terms of utilization of the streaming multiprocessors (right).

16.9 Summary

Dynamic programming algorithms divide complex problems into simple subproblems recursively. Bottom-up implementations can memoize intermediate results in tables. The dependencies between table cells define wavefronts, which are sets of sub-problems that can be solved in parallel. This chapter has explained how wavefront parallelism can be efficiently exploited on GPUs. Block-level tiling and hypertile transformations provide ways of optimizing the utilization of compute resources. More advanced optimizations that can provide further performance are inter-block synchronization mechanism and the use of DPX instructions.

16.10 Exercises

1. Implement a version of the Floyd-Warshall kernel that uses square thread blocks, instead of linear blocks as in the implementation of Section 16.4. In the square block implementation, the distance table is divided into square tiles that are assigned to the thread blocks. Threads of each block store the relevant parts of the `k` column and the `k` row in shared memory for reuse. Compare this implementation and the implementation presented in Section 16.4 in terms of memory accesses, data reuse, and performance.
2. Modify the Smith-Waterman kernel with thread block-level tiling to use rectangular tiles.

3. Implement a version of the Smith-Waterman kernel with thread block-level tiling that uses cooperative groups for synchronization across blocks.
4. Implement a version of the Smith-Waterman kernel with thread block-level tiling that uses the unidirectional synchronization. The single lookback synchronization presented in Chapter 11 can be a reference.
5. Implement a version of the Smith-Waterman kernel with hypertiles that uses the unidirectional synchronization. The single lookback synchronization presented in Chapter 11 can be a reference.
6. Replace the `max4()` device function that we use in the kernels in this chapter with a DPX instruction with the same functionality.

References

[1] M. Alser, J. Lindegger, C. Firtina, N. Almadhoun, H. Mao, G. Singh, J. Gomez-Luna, O. Mutlu, From molecules to genomic variations: accelerating genome analysis via intelligent algorithms and architectures, Computational and Structural Biotechnology Journal 20 (2022) 4579–4599.

[2] F. Irigoin, R. Triolet, Supernode partitioning, in: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1988, pp. 319–329.

[3] P. Di, D. Ye, Y. Su, Y. Sui, J. Xue, Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs, in: 2012 41st International Conference on Parallel Processing, IEEE, 2012, pp. 350–359.

[4] M.E. Belviranli, P. Deng, L.N. Bhuyan, R. Gupta, Q. Zhu, Peerwave: exploiting wavefront parallelism on GPUs with Peer-SM synchronization, in: Proceedings of the 29th ACM on International Conference on Supercomputing, 2015, pp. 25–35.

CHAPTER

17 Sparse matrix computation

CONTENTS

17.1 Background 402
17.2 A simple SpMV kernel with the COO format 404
17.3 Grouping row non-zeros with the CSR format 407
17.4 Improving memory coalescing with the ELL format 410
17.5 Regulating padding with the hybrid ELL-COO format 414
17.6 Reducing control divergence with the JDS format 416
17.7 Column-wise accessibility with the CSC format 418
17.8 Summary 421
17.9 Exercises 422
References 423

Our next parallel pattern is sparse matrix computation. In a sparse matrix, the majority of the elements are zeros. Storing and processing these zero elements are wasteful in terms of memory capacity, memory bandwidth, time, and energy. Many important real-world problems involve sparse matrix computation. Due to the importance of these problems, several sparse matrix storage formats and their corresponding processing methods have been developed and are widely used in the field. All these methods employ some type of compaction techniques to avoid storing or processing zero elements at the cost of introducing some level of irregularity into the data representation. Unfortunately, such irregularity can lead to underutilization of memory bandwidth, control flow divergence, and load imbalance in parallel computing. It is therefore important to strike a good balance between compaction and regularization. Some storage formats achieve a high level of compaction at a high level of irregularity. Others achieve a more moderate level of compaction while keeping the representation more regular. The relative performance of a parallel computation using each storage format is known to be heavily dependent on the distribution of non-zero elements in the sparse matrices. Understanding the wealth of work in sparse matrix storage formats and their corresponding parallel algorithms gives a parallel programmer important background for addressing compaction and regularization challenges in solving related problems.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00028-7
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

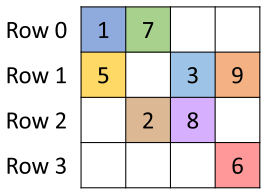


FIGURE 17.1

A simple sparse matrix example.

17.1 Background

A sparse matrix is a matrix where most of the elements are zeros. Sparse matrices arise in many scientific, engineering, and financial modeling problems. For example, matrices can be used to represent the coefficients in a linear system of equations. Each row of the matrix represents one equation of the linear system. In many science and engineering problems, the large number of variables and equations involved are sparsely coupled. That is, each equation only involves a small number of variables. This is illustrated in Fig. 17.1, where each column of the matrix corresponds to the coefficients for a variable: column 0 for x_0, column 1 for x_1, etc. For example, the fact that row 0 has non-zero elements in columns 0 and 1 indicates that only variables x_0 and x_1 are involved in equation 0. It should be clear that variables x_0, x_2, and x_3 are present in equation 1, variables x_1 and x_2 are present in equation 2, and only variable x_3 is present in equation 3. Sparse matrices are typically stored in a format, or representation, that avoids storing the zero elements.

Matrices are often used in solving linear systems of M equations of N variables in the form of $A \times X + Y = 0$, where A is an $M \times N$ matrix, X is a vector of N variables, and Y is a vector of M constant values. The objective is to solve for the X variable values that will satisfy all the equations. An intuitive approach is to invert the matrix so that $X = A^{-1} \times (-Y)$. This can be done for matrices of moderate size through methods such as Gaussian elimination. While it is theoretically possible to use these methods to solve equations represented in sparse matrices, the sheer size of many sparse matrices can simply overwhelm this intuitive approach. Furthermore, an inverted sparse matrix is often much larger than the original due the fact that the inversion process tends to generate many addition non-zero elements called "fill-ins". As a result, it is often impractical to compute and store the inverted matrix when solving real-world problems.

Instead, linear systems of equations represented in sparse matrices can be better solved with an iterative approach. When the sparse matrix A is positive-definite (i.e., $x^T Ax > 0$ for all non-zero vectors x in R^n), one can use Conjugate Gradient methods to iteratively solve the corresponding linear system with guaranteed convergence to a solution [1]. Conjugate Gradient methods guess a solution for X and perform $A \times X + Y$ and see if the result is close to a 0 vector. If not, we can use a gradient vector formula to refine the guessed X and perform another iteration of $A \times X + Y$.

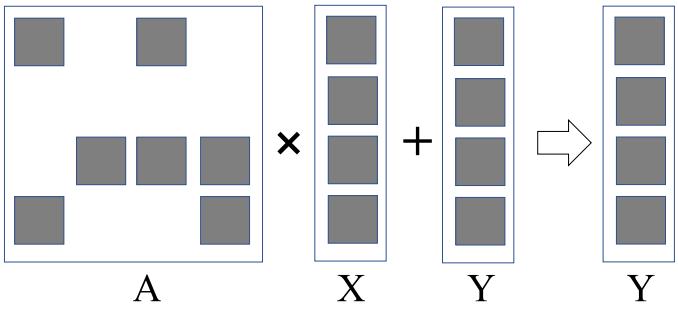


FIGURE 17.2

A small example of matrix-vector multiplication and accumulation.

These iterative solution methods for linear systems are closely related to the iterative solution methods for differential equations that we introduced in Chapter 8.

The most time-consuming part of iterative approaches to solving linear systems of equations is in the evaluation of $A \times X + Y$, which is a sparse matrix-vector multiplication and accumulation. Fig. 17.2 shows a small example of matrix-vector multiplication and accumulation, where A is a sparse matrix. The dark squares in A represent non-zero elements. In contrast, both X and Y are typically dense vectors. That is, most of the elements of X and Y hold non-zero values. Due to its importance, standardized library function interfaces have been created to perform this operation under the name SpMV (Sparse Matrix Vector multiplication and accumulation). We will use SpMV to illustrate the important tradeoffs between different storage formats in parallel sparse matrix computation.

The main objective of the different sparse matrix storage formats is to remove all the zero elements from the matrix representation. Removing all the zero elements not only saves storage, but it also eliminates the need to fetch these zero elements from memory and perform useless multiplication and addition operations with them. This can significantly reduce the consumption of memory bandwidth and computation resources. Moreover, for very large matrices, the reduction in the memory capacity needed to represent the matrix enables storing the entire matrix in the memory of one compute node. Otherwise, out-of-core methods or distributed computation would be needed to process such larger matrices, which suffer from high I/O or network communication latency.

There are various design considerations that go into the structure of a sparse matrix storage formats. The following is a list of some of the key considerations:

- Space efficiency (or compaction): the amount of memory capacity required to represent the matrix using the storage format
- Flexibility: the extent to which the storage format makes it easy to modify the matrix by adding or removing non-zeros
- Accessibility: the kinds of data that the storage format makes easy to access

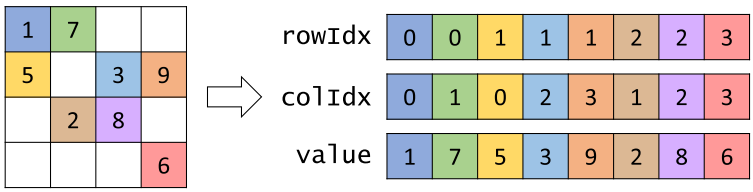


FIGURE 17.3

Example of the Coordinate (COO) format.

- Memory access efficiency: the extent to which the storage format enables an efficient memory access pattern for a particular computation (one facet of regularization)
- Load balance: the extent to which the storage format balances load across different threads for a particular computation (another facet of regularization)

Throughout this chapter, we introduce different storage formats and examine how these storage formats compare in each of the aforementioned design considerations.

17.2 A simple SpMV kernel with the COO format

The first sparse matrix storage format we will discuss is the Coordinate (COO) format. The COO format is illustrated in Fig. 17.3. COO stores the non-zero values in a one-dimensional array shown as the `value` array. Each non-zero element is stored with both its column index and row index. We have both `colIdx` and `rowIdx` arrays to accompany the `value` array. For example `A[0,0]` of our small example is stored at the entry with index 0 in the `value` array (1 in `value[0]`) with both its column index (0 in `colIdx[0]`) and its row index (0 in `rowIdx[0]`) stored at the same position in the other arrays.

COO completely removes all zero elements from the storage. It does incur storage overhead by introducing the `colIdx` and `rowIdx` arrays. In our small example where the number of zero elements is not much larger than the number of non-zero elements, the storage overhead is actually more than the space saved by not storing the zero elements. However, it should be clear that for sparse matrices where the vast majority of elements are zeros, the overhead introduced is far less than the space saved by not storing zeros. For example, in a sparse matrix where only 1% of the elements are non-zero values, the total storage for the COO representation including all the overhead would be around 3% of the space required to store both zero and non-zero elements.

In the COO example in Fig. 17.3, the non-zero elements are sorted by row index then by column index. While COO representations tend to be sorted in practice, sorting is not a requirement. The SpMV kernel presented in the section that uses the COO format works even if the non-zeros are not sorted.

One approach to performing SpMV in parallel using a sparse matrix represented in the COO format is to assign a thread to each non-zero element in the matrix.

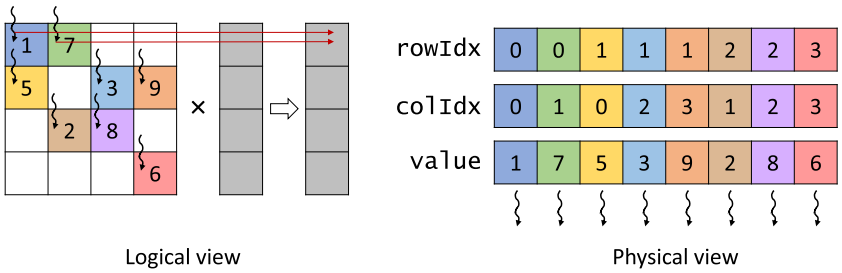


FIGURE 17.4

Example of parallelizing SpMV with the COO format.

```
01    __global__ void spmv_coo_kernel(COOMatrix cooMatrix, float* x, float* y) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        if(i < cooMatrix.numNonzeros) {
04            unsigned int row = cooMatrix.rowIdx[i];
05            unsigned int col = cooMatrix.colIdx[i];
06            float value = cooMatrix.value[i];
07            cuda::atomic_ref<float, cuda::thread_scope_device> y_ref(y[row]);
08            y_ref.fetch_add(x[col]*value, cuda::memory_order_relaxed);
09        }
10    }
```

FIGURE 17.5

A parallel SpMV/COO kernel.

An example of this parallelization approach is illustrated in Fig. 17.4, and the corresponding code is shown in Fig. 17.5. In this approach, each thread identifies the index of the non-zero element it is responsible for (line 02) and ensures that it is within bounds (line 3). Next, the thread identifies the row index (line 04), column index (line 05), and value (line 06) of the non-zero element it is responsible for from the `rowIdx`, `colIdx`, and `value` arrays, respectively. It then looks up the input vector value at the location corresponding to the column index, multiplies it with the non-zero value, then accumulates the result to the output value at the corresponding row index (lines 07-08). An atomic operation is used for the accumulation because multiple threads may update the same output element as is the case with the first two threads mapped to row 0 of the matrix in Fig. 17.4. It should be obvious that any SpMV computation code will reflect the storage format assumed. Therefore, we add the storage format to the name of the kernel to clarify the combination used. We also refer to the SpMV code in Fig. 17.5 as SpMV/COO.

Now we examine the COO format under the design considerations listed in Section 17.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency, we defer the discussion for later when we have introduced other formats. For flexibility, we observe that we can arbitrarily re-order the elements in a COO format without losing any information as long as we re-order the `rowIdx`, `colIdx`, and `value` arrays the same way. This is illustrated using our small example in Fig. 17.6, where we have reordered the elements of `rowIdx`, `colIdx`, and

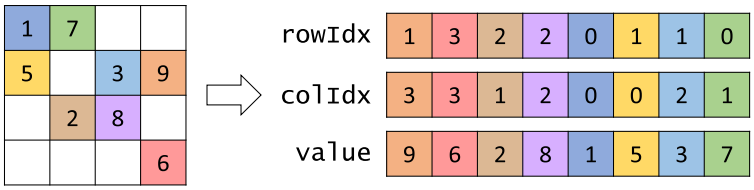


FIGURE 17.6

Re-ordering Coordinate (COO) format.

`value`. Now `value[0]` actually contains an element from row 1 and column 3 of the original sparse matrix. Because we have also shifted the row index and column index values along with the data value, we can correctly identify this element's location in the original sparse matrix.

In the COO format, we can process the elements in any order we want. The correct `y` element identified by `rowIdx[i]` will receive the correct contribution from the product of `value[i]` and `x[colIdx[i]]`. If we make sure that somehow we perform this operation for all elements of `value`, we will calculate the correct final answer regardless of the order in which we process these elements.

The reader may ask why we would want to reorder these elements. One reason is that the data may be read from a file that does not provide the non-zeros in a particular order, and we still need a consistent way of representing the data. For this reason, COO is a popular choice of storage format when the matrix is initially being constructed. Another reason is that not having to provide any ordering guarantees enables non-zeros to be added to the matrix by simply appending entries at the end of each of the three arrays. For this reason, COO is a popular choice of storage format when the matrix is modified throughout the computation. We will also see another benefit of the flexibility of the COO format in Section 17.5.

The next design consideration we look at is accessibility. COO makes it easy to access, for a given non-zero, its corresponding row index and column index. This feature of COO enables parallelization across non-zero elements in SpMV/COO. On the other hand, COO does not make it easy to access, for a given row or column, all the non-zeros in that row or column. For this reason, COO would not be a good choice of format if the computation required a row- or column-wise traversal of the matrix.

For memory access efficiency, we refer to the physical view in Fig. 17.4 for how the threads access the matrix data from memory. The access pattern is such that consecutive threads access consecutive elements in each of the three arrays that form the COO format. Therefore, accesses to the matrix by SpMV/COO are coalesced.

For load balance, we recall that each thread is responsible for a single non-zero value. Hence, all threads are responsible for the same amount of work, which means that we do not expect any control divergence to take place in SpMV/COO except for the threads at the boundary.

The main drawback of SpMV/COO is the need to use atomic operations. The reason for using atomic operations is that multiple threads are assigned to non-zeros

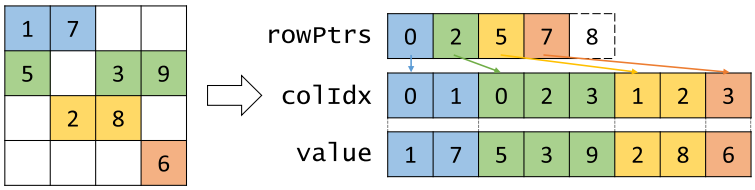


FIGURE 17.7

Example of Compressed Sparse Row (CSR) format.

in the same row, and therefore need to update the same output value. Using atomic operations in SpMV is problematic for two reasons. The first reason is that they incur high latency and have low throughput as we have seen in previous chapters. The second reason is that the order in which the atomic operations are applied to the same output element is non-deterministic. This nondeterminism in the order of the operations is problematic when operating on floating point values because it may cause numerical stability issues (see Appendix A). The atomic operations can be avoided if all the non-zeros in the same row were assigned to the same thread such that the thread will be the only one updating the corresponding output value. However, recall that the COO format does not give this accessibility. In the COO format, it is not easy to access, for a given row, all the non-zeros in that row. In the next section, we will see another storage format that provides this kind of accessibility.

17.3 Grouping row non-zeros with the CSR format

In Section 17.2, we have seen that parallelizing SpMV with the COO format suffers from the use of atomic operations because the same output value is updated by multiple threads. These atomic operations can be avoided if the same thread was responsible for all the non-zeros of a row, which requires the storage format to give us the ability to access, for a given row, all the non-zeros in that row. This kind of accessibility is provided by the Compressed Sparse Row (CSR) storage format.

Fig. 17.7 illustrates how the matrix in Fig. 17.1 can be stored using the CSR format. Like the COO format, CSR stores the non-zero values in a one-dimensional array shown as the `value` array in Fig. 17.7. However, these non-zero values are grouped by row. For example, we store the non-zero elements of Row 0 (1 and 7) first, followed by the non-zero elements of Row 1 (5, 3, and 9), followed by the non-zero elements of Row 2 (2 and 8), and finally the non-zero elements of Row 3 (6).

Also similar to the COO format, CSR stores for each non-zero element in the `value` array its column index at the same position in the `colIdx` array. Naturally, these column indices are grouped by row as the values are. In Fig. 17.7, the non-zeros of each row are sorted by their column indices in increasing order. Sorting the non-zeros in this way results in favorable memory access patterns, but it is not necessary. The non-zeros within each row may not necessarily be sorted by their column index,

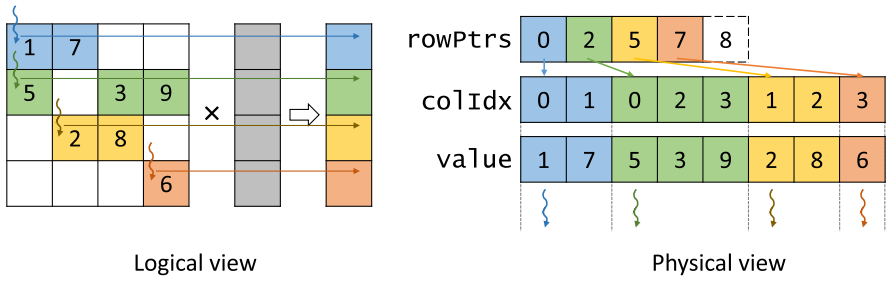


FIGURE 17.8

Example of parallelizing SpMV with the CSR format.

```
01  __global__ void spmv_csr_kernel(CSRMatrix csrMatrix, float* x, float* y) {
02     unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;
03     if(row < csrMatrix.numRows) {
04       float sum = 0.0f;
05       for(unsigned int i=csrMatrix.rowPtrs[row]; i<csrMatrix.rowPtrs[row+1];
06                                                                    ++i) {
07          unsigned int col = csrMatrix.colIdx[i];
08          float value = csrMatrix.value[i];
09          sum += x[col]*value;
10       }
11       y[row] += sum;
12     }
13  }
```

FIGURE 17.9

A parallel SpMV/CSR kernel.

and the kernel presented in this section will still work correctly. When the non-zeros within each row are sorted by their column index, the layout of the `value` array (and the `colIdx` array) for CSR can be viewed as the row-major layout of the matrix after eliminating all the zero elements.

The key distinction between the COO format and the CSR format is that the CSR format replaces the `rowIdx` array with a `rowPtrs` array that stores the starting offset of each row's non-zeros in the `colIdx` and `value` arrays. In Fig. 17.7, we show a `rowPtrs` array whose elements are the indices for the beginning locations of each row. That is, `rowPtrs[0]` indicates that Row 0 starts at location 0 of the `value` array, `rowPtrs[1]` indicates that Row 1 starts at location 2, etc. Note that `rowPtrs[4]` stores the starting location of a non-existing "Row 4". This is for convenience, as some algorithms need to use the starting location of the next row to delineate the end of the current row. This extra marker gives a convenient way to locate the ending location of Row 3.

To perform SpMV in parallel using a sparse matrix represented in the CSR format, one can assign a thread to each row of the matrix. An example of this parallelization approach is illustrated in Fig. 17.8, and the corresponding code is shown in Fig. 17.9. In this approach, each thread identifies the row that it responsible for (line 02) and ensures that it is within bounds (line 03). Next, the thread loops through the non-zero elements of its row to perform the dot product (line 05-06). To find the row's non-zero

elements, the thread looks up their starting index in the `rowPtrs` array (`rowPtrs[row]`). It also finds where they end by looking up the starting index of the next row's non-zeros (`rowPtrs[row + 1]`). For each non-zero element, the thread identifies its column index (line 07) and value (line 08). It then looks up the input value at the location corresponding to the column index, multiplies it with the non-zero value, and accumulates the result to a local variable `sum` (line 09). The `sum` variable is initialized to 0 before the dot product loop begins (line 04) and is accumulated to the output vector after the loop is over (line 11). Notice that the accumulation of `sum` to the output vector does not require atomic operations. The reason is that each row is traversed by a single thread, so each thread will write to a distinct output value as illustrated in Fig. 17.8.

Now we examine the CSR format under the design considerations listed in Section 17.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency, we observe that CSR is more space efficient than COO. COO requires three arrays, `rowIdx`, `colIdx`, and `value`, that each have as many elements as the number of non-zeros. In contrast, CSR only requires two arrays, `colIdx` and `value`, with as many elements as the number of non-zeros. The third array, `rowPtrs`, only requires as many elements as the number of rows plus one, which makes it substantially smaller than the `rowIdx` array in COO. This difference makes CSR more space efficient than COO.

For flexibility, we observe that CSR is less flexible than COO when it comes to adding non-zeros to the matrix. In COO, a non-zero can be added by simply appending it to the ends of the arrays, provided we have over-allocated memory in anticipation of adding non-zeros. On the other hand, in CSR, a non-zero to be added must be added to the specific row that it belongs to. This means that the non-zero elements of the later rows would all need to be shifted, and the row offsets of the later rows in the `rowPtrs` array would all need to be incremented accordingly. For this reason, adding non-zeros to a CSR matrix is substantially more expensive than adding them to a COO matrix.

For accessibility, CSR makes it easy to access, for a given row, the non-zeros in that row. This feature of CSR enables parallelization across rows in SpMV/CSR, which is what allows it to avoid atomic operations in comparison with SpMV/COO. In a real-world sparse matrix application, there are usually thousands to millions of rows, each of which contains tens to hundreds of non-zero elements. This makes parallelization across rows seem very appropriate: there are many threads, and each thread has a substantial amount of work. On the other hand, for some applications, the sparse matrix may not have enough rows to fully utilize all the GPU threads. In these kinds of applications, the COO format can extract more parallelism since there are more non-zeros than rows. Moreover, CSR does not make it easy to access, for a given column, all the non-zeros in that column. If easy access to all elements of a column is needed, an application may need to maintain an additional column-oriented layout of the matrix, such as the CSC format covered in Section 17.7.

For memory access efficiency, we refer to the physical view in Fig. 17.8 for how the threads access the matrix data from memory during the first iteration of the dot

product loop. The access pattern is such that consecutive threads access data elements that are far apart. In particular, threads 0, 1, 2, and 3 access `value[0]`, `value[2]`, `value[5]`, and `value[7]`, respectively, in the first iteration of their dot product loop. They will then access `value[1]`, `value[3]`, `value[6]`, and no data, respectively, in the second iteration, and so on. As a result, the accesses to the matrix by the parallel SpMV/CSR kernel in Fig. 17.9 are not coalesced. The kernel does not make efficient use of memory bandwidth.

For load balance, we observe that the SpMV/CSR kernel can potentially have significant control flow divergence in all warps. The number of iterations taken by a thread in the dot product loop depends on the number of non-zero elements in the row assigned to the thread. Since the distribution of non-zero elements among rows can be random, adjacent rows can have very different number of non-zero elements. As a result, there can be wide-spread control flow divergence in most or even all warps.

In summary, we have seen that the advantages of CSR over COO are that it has better space efficiency, and that it gives us access to all the non-zeros of a row, which allows us to avoid atomic operations by parallelizing the computation across rows in SpMV/CSR. On the other hand, the disadvantages of CSR over COO are that it provides less flexibility with adding non-zero elements to the sparse matrix, it exhibits a memory access pattern that is not amenable to coalescing, and it causes high control divergence. In the following sections, we discuss additional storage formats that sacrifice some space efficiency as compared to CSR in order to improve memory coalescing and reduce control divergence. Note that converting from COO to CSR on the GPU is an excellent exercise for the reader, using multiple fundamental parallel computing primitives, including histogram and prefix sum.

17.4 Improving memory coalescing with the ELL format

The problem of non-coalesced memory accesses can be addressed by applying data padding and transposition on the sparse matrix data. These ideas were used in the ELL storage format, whose name came from the sparse matrix package in ELLPACK a package for solving elliptic boundary value problems [2].

A simple way to understand the ELL format is to start with the CSR format, as is illustrated in Fig. 17.10. From a CSR representation which groups non-zeros by row, we determine the rows with the maximal number of non-zero elements. We then add padding elements to all other rows after the non-zero elements to make them the same length as the maximal rows. This makes the matrix a rectangular matrix. For our small sparse matrix example, we determine that row 1 has the maximal number of elements. We then add one padding element to row 0, one padding element to row 2, and two padding elements to row 3 to make all them the same length. These additional padding elements are shown as squares with an * in Fig. 17.10. Now the matrix has become a rectangular matrix. Note that the `colIdx` array also needs to be padded the same way to preserve its correspondence to the values array.

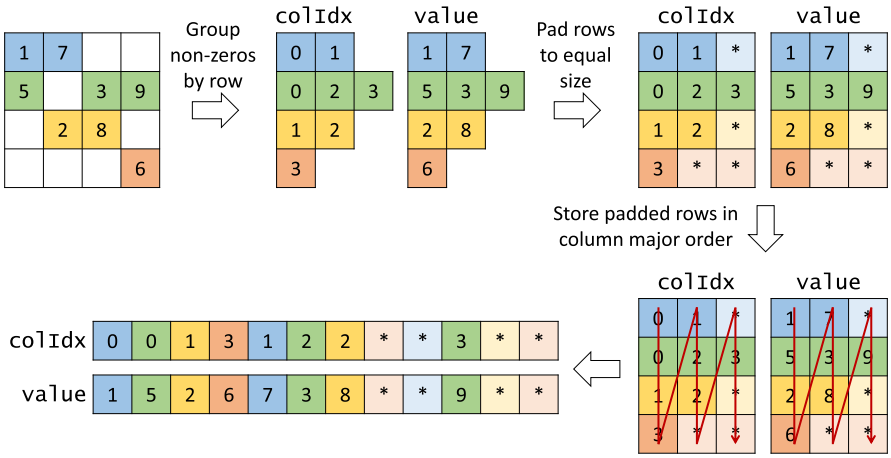


FIGURE 17.10

Example of ELL storage format.

We can now lay the padded matrix out in column major order. That is, we will place all elements of column 0 in consecutive memory locations, followed by all elements of column 1, and so on. This is equivalent to transposing the rectangular matrix in the row major order used by the C language. In terms of our small example, after the transposition, `value[0]` through `value[3]` now contain 1, 5, 2, and 6, the 0th elements of all rows. This is illustrated in the bottom-left portion of Fig. 17.10. Similarly, `colIdx[0]` through `colIdx[3]` contain the column positions of 0th elements of all rows. Note that we no longer need the `rowPtrs` since the beginning of row `r` is now simply `value[r]`. With the padded elements, it is also very easy to move from the current element of row `r` to the next element by simply adding the number of rows in the original matrix to the index. For example, the 0th element of row 2 is in `value[2]` and the next element is in `value[2+4]` which is equivalent to `value[6]`, where 4 is the number of rows in the original matrix in our small example.

We illustrate how to parallelize SpMV using the ELL format in Fig. 17.11, along with a parallel SpMV/ELL kernel in Fig. 17.12. Like CSR, each thread is assigned to a different row of the matrix (line 02) and a boundary check ensures that the row is within bounds (line 03). Next, a dot product loop goes through the non-zero elements of each row (line 05). Note that the SpMV/ELL kernel assumes that the input matrix has a vector `ellMatrix.nnzPerRow` that records the number of non-zeros in each row and allows each thread to only iterate through the non-zeros in its assigned row. If the input matrix does not have this vector, the kernel can simply iterate through all elements including the padding elements and still execute correctly since the padding elements have value zero and will not affect the output values. Next, since the compressed matrix is stored in column major order, the index `i` of the non-zero element in the one-dimensional array can be found by multiplying the iteration number `t` by the number of rows and adding the row index (line 06). Next, the thread loads the column

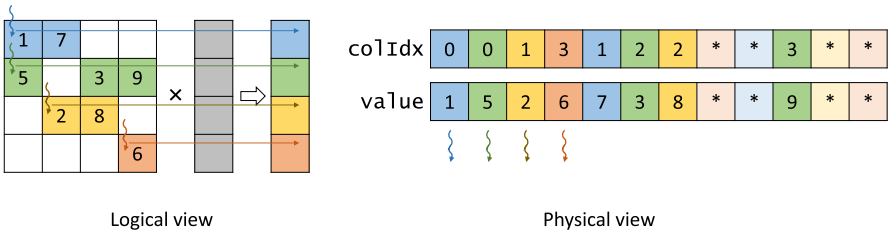


FIGURE 17.11

Example of parallelizing SpMV with the ELL format.

```
01    __global__ void spmv_ell_kernel(ELLMatrix ellMatrix, float* x, float* y) {
02        unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;
03        if(row < ellMatrix.numRows) {
04            float sum = 0.0f;
05            for(unsigned int t = 0; t < ellMatrix.nnzPerRow[row]; ++t) {
06                unsigned int i = t*ellMatrix.numRows + row;
07                unsigned int col = ellMatrix.colIdx[i];
08                float value = ellMatrix.value[i];
09                sum += x[col]*value;
10            }
11            y[row] = sum;
12        }
13    }
```

FIGURE 17.12

A parallel SpMV/ELL kernel.

index (line 07) and non-zero value (line 08) from the ELL matrix arrays. Note that the accesses to these arrays are coalesced because the index `i` is expressed in terms of `row`, which itself is expressed in terms of `threadIdx.x`, meaning that consecutive threads have consecutive array indices. Next, the thread looks up the input value, multiplies it with the non-zero value, and accumulates the result to a local variable `sum` (line 09). The `sum` variable is initialized to 0 before the dot product loop begins (line 04) and is accumulated to the output vector after the loop is over (line 11).

Now we examine the ELL format under the design considerations listed in Section 17.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency, we observe that the ELL format is less space efficient than the CSR format due to the space overhead of the padding elements. The overhead of the padding elements highly depends on the distribution of non-zeros in the matrix. In situations where one or a small number of rows have an exceedingly large number of non-zero elements, the ELL format will result in excessive number of padded elements. Consider our sample matrix, in the ELL format we have replaced a 4x4 matrix with a 4x3 matrix, and with the overhead from the column indices we are storing more data than contained in the original 4x4 matrix. For a more realistic example, if a 1000x1000 sparse matrix has 1% of its elements of non-zero value. This means that on average, each row has 10 non-zero elements. With the overhead, the size of a CSR representation would be about 2% of the uncompressed total size. Assume that one of the rows has 200 non-zero values while all other rows have less than

10. Using the ELL format, we will pad all other rows to 200 elements. This makes the ELL representation about 40% of the uncompressed total size and 20 times larger than the CSR representation. This calls for a method to control the number of padded elements when we convert from the CSR format to the ELL format, which we introduce in Section 17.5.

For flexibility, we observe that ELL is more flexible than CSR when it comes to adding non-zeros to the matrix. In CSR, adding a non-zero to a row would require shifting all the non-zeros of the subsequent rows and incrementing their row offsets. However, in ELL, as long as a row does not have the maximum number of non-zeros in the matrix, a non-zero can be added to the row by simply replacing a padding element with an actual value. Moreover, if we have over-allocated memory for the ELL arrays, we can even add elements to the row with the maximum number of non-zeros by extending the bounds of the arrays with one more column and adding one more padding element to the other rows without shifting any of the existing elements. In this case, the ELL format is as flexible as the COO format.

For accessibility, ELL gives us the accessibility of both CSR and COO. We have seen in Fig. 17.12 how ELL allows to access, given a row index, the non-zeros of that row. However, ELL also allows to access, given the index of a non-zero element, the row and column index of that element. The column index is trivial to find as it can be accessed from the `colIdx` array at the same location `i`. However, the row index can also be accessed due to the regular nature of the padded matrix. Recall that the index `i` of the non-zero element was calculated in Fig. 17.9 as:

```
i = t*ellMatrix.numRows + row
```

Therefore, if instead `i` is given and we would like to find `row`, it can be found as:

```
row = i%ellMatrix.numRows
```

because `row` is always less than `ellMatrix.numRows`, so `row%ellMatrix.numRows` is simply row itself. This accessibility of ELL allows parallelization across both rows as well as non-zero elements. However, parallelization across non-zero elements will have some threads wastefully assigned to padding elements, which is a drawback over COO.

For memory access efficiency, we refer to the physical view in Fig. 17.11 for how the threads access the matrix data from memory during the first iteration of the dot product loop. The access pattern is such that consecutive threads access consecutive data elements. By arranging the elements in column major order, all adjacent threads are now accessing adjacent memory locations, enabling memory coalescing and thus making more efficient use of memory bandwidth. Some GPU architectures, especially in the older generations, have more strict address alignment rules for memory coalescing. One can force each iteration of the SpMV/ELL kernel to be fully aligned to architecturally specified alignment units such as 64-bytes by adding a few rows to the end of the matrix before transposition.

For load balance, we observe that SpMV/ELL still exhibits the same load imbalance as SpMV/CSR because each thread still loops over the number of non-zeros in

the row it is responsible for. Therefore, ELL does not address the problem of control divergence.

In summary, the ELL format improves on the CSR format by allowing more flexibility to add non-zeros elements, better accessibility, and most importantly, more opportunities for memory coalescing in SpMV/ELL. However, ELL has worse space efficiency than CSR, and the control divergence of SpMV/ELL is as bad as that of SpMV/CSR. In Section 17.5, we will see how we can improve on the ELL format to address the problems of space efficiency and control divergence.

17.5 Regulating padding with the hybrid ELL-COO format

The problems of low space efficiency and control divergence in the ELL format are most pronounced when one or a small number of rows have an exceedingly large number of non-zero elements. If we have a mechanism to "take away" some elements from these rows, we can reduce the number of padded elements in ELL and also reduce the control divergence. The answer lies in an important use case for the COO format.

The COO format can be used to curb the length of rows in the ELL format. Before we convert a sparse matrix to ELL, we can take away some of the elements from the rows with exceedingly large number of non-zero elements and place them into a separate COO storage. We can use SpMV/ELL on the remaining elements. With the excess elements removed from the extra-long rows, the number of padded elements for other rows can be significantly reduced. We can then use a SpMV/COO to finish the job. This approach of employing two formats to collaboratively complete a computation is often referred to as a hybrid method.

Fig. 17.13 illustrates how an example matrix can be represented using the hybrid ELL-COO format. We see that in the ELL format alone, rows 1 and 6 have the largest number of non-zero elements, causing the other rows to have excessive padding. To address this issue, we remove the last three non-zero elements of row 2 and the last two non-zero elements of row 6 from the ELL representation and move them into a separate COO representation. By removing these elements, we reduce the maximal number of non-zero elements among all rows in the small sparse matrix from 5 to 2. As shown in Fig. 17.13, we reduced the number of padded elements from 22 to 3. More importantly, all threads now only need to take 2 iterations.

The reader may question whether the additional work done to separate COO elements from an ELL format could incur too much overhead. The answer is: it depends. In situations where a sparse matrix is only used in one SpMV calculation, this extra work can indeed incur significant overhead. However, in many real-world applications, the SpMV operation is performed with the same sparse matrix repeatedly in an iterative solver. In each iteration of the solver, the `x` and `y` vectors vary but the sparse matrix remains the same since its elements correspond to the coefficients of the linear system of equations being solved and these coefficients do not change from iteration

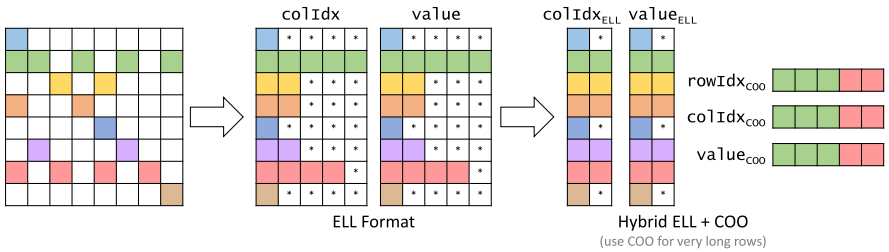


FIGURE 17.13

Hybrid ELL + COO example.

to iteration. So, the work done to produce both the hybrid ELL and COO representation can be amortized across many iterations. We will come back to this point in Section 17.6.

Now we examine the hybrid ELL-COO format under the design considerations listed in Section 17.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency, we observe that the hybrid ELL-COO format has better space efficiency than the ELL format alone because it reduces the amount of padding used.

For flexibility, we observe that the hybrid ELL-COO format is more flexible than the ELL format when it comes to adding non-zeros to the matrix. With both formats, we can add non-zero elements to rows that have padding elements by replacing the padding elements. If the row does not have any padding elements that can be replaced, then we need to append a column in the ELL format, whereas in the hybrid ELL + COO format, we can simply append non-zeros to the COO part of the format.

For accessibility, we observe that the hybrid ELL-COO format sacrifices accessibility compared to the ELL format alone. In particular, it is not always possible to access, given a row index, all the non-zeros in that row. Such an access can be done only for the rows that fit in the ELL part of the format. If the row overflows to the COO part, then finding all the non-zeros of the row would require searching the COO part which is expensive.

For memory access efficiency, both SpMV/ELL and SpMV/COO exhibit coalesced memory accesses to the sparse matrix. Hence, their combination will also result in a coalesced access pattern.

For load balance, removing non-zeros from the long rows in the ELL part of the format reduces control divergence of the SpMV/ELL kernel. These non-zeros are placed in the COO part of the format, which does not impact control divergence since SpMV/COO does not exhibit control divergence as we have seen.

In summary, the hybrid ELL-COO format, in comparison with the ELL format alone, improves space efficiency by reducing padding, retains the flexibility of adding non-zeros to the matrix, retains the coalesced memory access pattern, and reduces control divergence. The price paid is a small limitation in accessibility, where it becomes more difficult to access all the non-zeros of a given row if that row overflows to the COO part of the format.

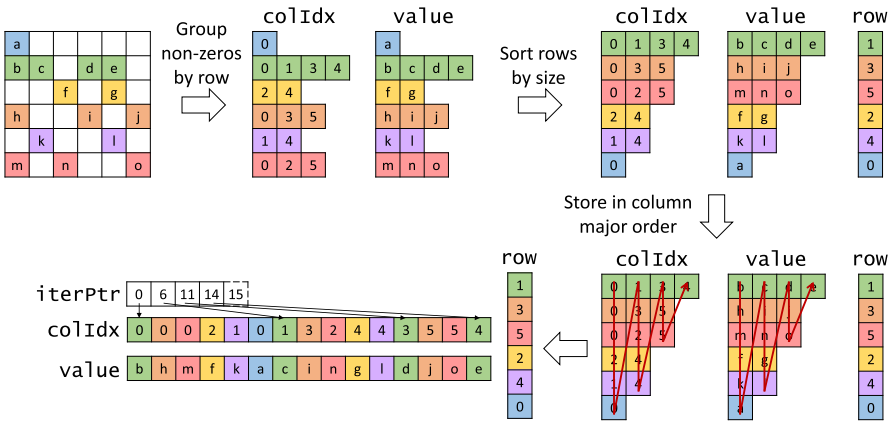


FIGURE 17.14

Example of JDS storage format.

17.6 Reducing control divergence with the JDS format

We have seen that the ELL format can be used to achieve coalesced memory access patterns when accessing the sparse matrix in SpMV, and that the hybrid ELL-COO format can further improve space efficiency by reducing padding and can also reduce control divergence. In this section, we will look at another format that can achieve coalesced memory access patterns in SpMV and also reduce control divergence, without the need to perform any padding. The idea is to sort the rows according to their length, say from the longest to the shortest. Since the sorted matrix looks largely like a triangular matrix, the format is often referred to as the Jagged Diagonal Storage (JDS) format.

Fig. 17.14 illustrates how a matrix can be stored using the JDS format. First, the non-zeros are grouped by row like with CSR and ELL. Next, the rows are sorted by the number of non-zeros in each row in increasing order. As we sort the rows, we typically maintain an additional `row` array that preserves the index of the original row. Whenever we exchange two rows in the sorting process, we also exchange the corresponding elements of the row array. This way, we can always keep track the original position of all rows. After the rows have been sorted, the non-zeros in the `value` array and their corresponding column indices in the `colIdx` array are stored in column major order. A `iterPtr` array is added to track the beginning of the non-zero elements for each iteration.

Fig. 17.15 illustrates how SpMV can be parallelized using the JDS format. Each thread is assigned to a row of the matrix and iterates through the non-zeros of that row, performing the dot product along the way. The threads use the `iterPtr` array to identify where the non-zeros of each iteration begin. It should be clear from the physical view on the right side of Fig. 17.15, which depicts the first iteration for each

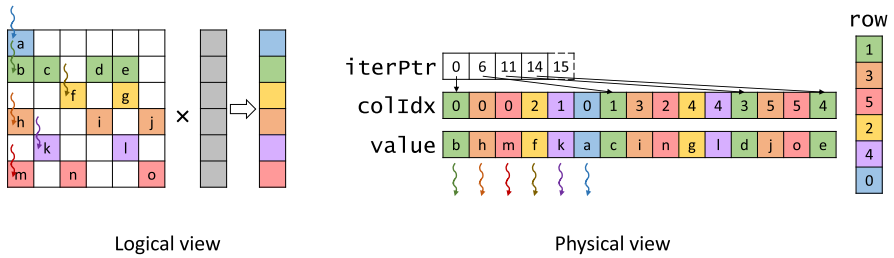


FIGURE 17.15

Example of parallelizing SpMV with the JDS format.

thread, that the threads access the non-zeros and column indices in the JDS arrays in a coalesced manner. The code for implementing SpMV/JDS is left as an exercise.

In another variation of the JDS format, the rows after being sorted can be partitioned into sections of rows. Since the rows have been sorted, all rows in a section will likely have more or less uniform number of non-zero elements. We can then generate the ELL representation for each section. Within each section, we only need to pad the rows to match the row with the maximal number of elements in that section. This would reduce the number of padding elements substantially compared to one ELL representation of the entire matrix. In this variation of JDS, the `iterPtr` array would not be needed. Instead, one would need a section offset array that indicates the beginning of each ELL section only (as opposed to each iteration).

The reader should ask whether sorting rows will result into incorrect solutions to the linear system of equations. Recall that we can freely re-order equations of a linear system without changing the solution. As long as we re-order the y elements along with the rows, we are effectively re-ordering the equations. Therefore, we will end up with the correct solution. The only extra step is to reorder the final solution back to the original order using the `row` array. The other question is whether sorting will incur significant overhead. The answer is similar to what we saw in the hybrid ELL-COO method. As long as the SpMV/JDS kernel is used in an iterative solver, one can afford to perform such sorting as well as the re-ordering of the final solution `x` elements and amortize the cost among many iterations of the solver.

Now we examine the ELL format under the design considerations listed in Section 17.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency, the JDS format is more space efficient than the ELL format because it avoids padding. The variant of JDS that uses ELL for each section has padding, but the amount of padding is less than that with the ELL format.

For flexibility, the JDS format, does not make it easy to add non-zeros to a row of the matrix. It is even less flexible than the CSR format because adding non-zeros changes the sizes of the rows, which may require rows to be resorted.

For accessibility, the JDS format is similar to the CSR format in that it allows us to access, given a row index, the non-zero elements of that row. On the other hand, it

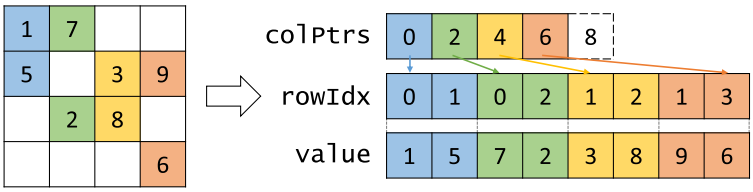


FIGURE 17.16

Example of Compressed Sparse Column (CSC) format.

does not make it easy to access, given a non-zero, the row index and column index of that non-zero, as the COO and ELL formats do.

For memory access efficiency, the JDS format is like the ELL format in that it stores the non-zeros in column major order. Accordingly, the JDS format enables accesses to the sparse matrix to happen in a coalesced manner. Because JDS does not require padding, the starting location of memory accesses in each iteration, as shown in the physical view of Fig. 17.15, can vary in arbitrary ways. As a result, there is no simple, inexpensive way to force all iterations of the SpMV/JDS kernel to start at architecturally specified alignment boundaries. This lack of option to force alignment can make memory accesses for JDS less efficient than those in ELL.

For load balance, the most unique feature of JDS is that it sorts the rows of the matrix such that threads in the same warp are likely to iterate over rows of similar length. Therefore, JDS is effective at reducing control divergence.

17.7 Column-wise accessibility with the CSC format

We have seen that the COO and ELL formats allow us to access the row and column indexes of a given non-zero element, and that the CSR, ELL, and JDS formats allow us to access all the non-zero elements of a given row. For some computations other than SpMV, we may be interested in performing a column-wise traversal, i.e., we may be interested in accessing all the non-zero elements of a given column. For such computations, a useful format is the Compressed Sparse Column (CSC) format.

Fig. 17.16 illustrates how the matrix in Fig. 17.1 can be stored using the CSC format. The CSC format strongly resembles the CSR format, but with the treatment of rows and columns swapped. The CSC format stores the non-zero values in a one-dimensional array shown as the `value` array in Fig. 17.16. These non-zero values are grouped by column: we store the non-zero elements of Column 0 (1 and 5) first, followed by the non-zero elements of Column 1 (7 and 2), followed by the non-zero elements of Column 2 (3 and 8), and finally the non-zero elements of Column 3 (9 and 6).

The CSC format also stores for each non-zero element in the `value` array its row index at the same position in the `rowIdx` array. In Fig. 17.16, the non-zeros of each column are sorted by their row indices in increasing order, but it is not necessary.

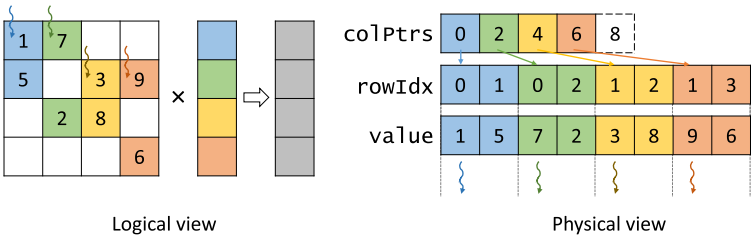


FIGURE 17.17

Example of parallelizing SpMV with the CSC format.

```
01  __global__ void spmv_csc_kernel(CSCMatrix cscMatrix, float* x, float* y) {
02      unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
03      if(col < cscMatrix.numCols) {
04          float inValue = x[col];
05          for(unsigned int i = cscMatrix.colPtrs[col]; i < cscMatrix.colPtrs[col + 1];
06                                                                          ++i) {
07              unsigned int row = cscMatrix.rowIdxs[i];
08              float value = cscMatrix.values[i];
09              cuda::atomic_ref<float, cuda::thread_scope_device> y_ref(y[row]);
10              y_ref.fetch_add(inValue*value, cuda::memory_order_relaxed);
11          }
12      }
13  }
```

FIGURE 17.18

A parallel SpMV/CSR kernel.

Finally, the CSC format stores a `colPtrs` array that stores the starting offset of each column's non-zeros in the `rowIdx` and `value` arrays. In Fig. 17.16, we show a `colPtrs` array whose elements are the indices for the beginning locations of each column. That is, colPtrs[0] indicates that Column 0 starts at location 0 of the `value` array, colPtrs[1] indicates that Column 1 starts at location 2, etc. The last value, colPtrs[4] stores the starting location of a non-existing "Column 4" to delineate the end of Column 3.

The CSC format is not intended to be used for performing SpMV. However, we will show an implementation of SpMV using the CSC format for completeness, and for the interesting exercise of comparing its advantages and disadvantages to other formats. To perform SpMV using the CSC format, one can assign a thread to each column of the matrix. An example of this parallelization approach is illustrated in Fig. 17.17, and the corresponding code is shown in Fig. 17.18. In this approach, each thread identifies the column that it responsible for (line 02) and ensures that it is within bounds (line 03). Next, the thread loads the input vector value that corresponds to that column (line 04). After that, the thread loops through the non-zero elements of its column (line 05-06). To find the column's non-zero elements, the thread looks up their starting index in the `colPtrs` array (colPtrs[col]). It also finds where they end by looking up the starting index of the next column's non-zeros (colPtrs[col + 1]). For each non-zero element, the thread identifies its row index (line 07) and value (line 08). Finally, it multiplies the input vector value with the matrix non-zero value, and

atomically accumulates the result to the corresponding location in the output vector (lines 09-10). The atomic operation is needed because different threads iterating through different columns may simultaneously attempt to update the same output vector element, such as the first and second thread in Fig. 17.17.

Now we examine the CSC format under the design considerations listed in Section 17.1: space efficiency, flexibility, accessibility, memory access efficiency, and load balance. For space efficiency, we observe that the CSC format has the same space efficiency as the CSR format, assuming the matrix has a similar number of rows and columns. It requires two arrays, `rowIdx` and `value`, with as many elements as the number of non-zeros and a third array, `colPtrs`, that requires as many elements as the number of columns plus one.

For flexibility, we observe that CSC is as inflexible as CSR when it comes to adding non-zeros to the matrix. In CSC, a non-zero to be added must be added to the specific column that it belongs to, which means that the non-zero elements of the later columns would all need to be shifted and the column offsets of the later columns in the `colPtrs` array would all need to be incremented accordingly.

For accessibility, CSC makes it easy to access, for a given column, the non-zeros in that column. As mentioned earlier, this kind of accessibility is not favorable for SpMV since SpMV does not need to perform column-wise traversal. However, we will mention later that this kind of accessibility may be useful for other computations.

For memory access efficiency, we refer to the physical view in Fig. 17.17 for how the threads access the matrix data. The access pattern is such that consecutive threads access data elements that are far apart. In particular, threads 0, 1, 2, and 3 access `value[0]`, `value[2]`, `value[4]`, and `value[6]`, respectively, in the first iteration of their dot product loop. They will then access `value[1]`, `value[3]`, `value[5]`, and `value[7]`, respectively, in the second iteration. As a result, the accesses to the matrix by the parallel SpMV/CSC kernel in Fig. 17.18 are not coalesced. Furthermore, the access to the output vector requires atomic operations because the same output vector element may be accessed by different threads.

For load balance, we observe that the SpMV/CSC kernel can potentially have significant control flow divergence in all warps. The number of loop iterations performed by a thread depends on the number of non-zero elements in the column assigned to the thread. Since the distribution of non-zero elements among columns can be random, adjacent columns can have a very different number of non-zero elements. Although this is coincidentally not the case in Fig. 17.17 where all columns have the same number of non-zeros, in practice different columns can have a very different number of non-zeros. As a result, there can be wide-spread control flow divergence in most or even all warps.

Overall, the CSC format when used for SpMV seems to combine the worst aspects of the COO and CSR formats together. Like the COO format, it requires atomic operations when accessing the output vector. Moreover, like the CSR format, it incurs accesses to the input matrix that are not coalesced and imposes control divergence. The uncoalesced accesses to the input matrix and the control divergence can be partially mitigated by applying techniques similar to those applied by the ELL and JDS

formats. However, the atomic operations when accessing the output vector would remain.

CSC does have one favorable feature when performing SpMV which is the manner in which the input vector is accessed. While the SpMV kernels implemented for all other formats access the input vector randomly, SpMV/CSC accesses the input vector in a coalesced manner as seen in Fig. 17.18 (line 04). In particular, adjacent threads assigned to adjacent columns load adjacent values from the input vector. Moreover, each value in the input vector is loaded only once and reused for all the non-zeros of the column. Nevertheless, this favorable access to the input vector does not overcome the unfavorable accesses to the input matrix and output vector, not to mention the control divergence, making CSC an unsuitable format for SpMV overall.

CSC is most useful for computations where column-wise traversal is most natural. For example, assume we needed to perform vector-matrix multiplication instead of matrix-vector multiplication, i.e., the input vector is the first operand and the matrix is the second operand. In this case, every element in the output vector would be the dot product of the input vector and the corresponding column in the input matrix. CSC would be a natural fit for such a computation because it involves a column-wise traversal. As another example, assume we are performing matrix-vector multiplication, but the input vector is sparse. This computation is referred to as sparse-matrix-sparse-vector multiplication (SpMSpV). In this case, we can leverage the fact that the input vector is sparse and skip all the non-zero elements in the input matrix that are in a column corresponding to an input vector value that is zero. Skipping the non-zero elements of an entire column is possible with CSC because they are grouped together.

17.8 Summary

In this chapter, we presented sparse matrix computation as an important parallel pattern. Sparse matrices are important in many real-world applications that involve modeling complex phenomena. Furthermore, sparse matrix computation is a simple example of data-dependent performance behavior of many large real-world applications. Due to the large amount of zero elements, compaction techniques are used to reduce the amount of storage, memory accesses, and computation performed on these zero elements. Using this pattern, we introduce the concept of regularization using hybrid methods and sorting/partitioning. These regularization methods are used in many real-world applications. Interestingly, some of the regularization techniques reintroduce zero elements into the compacted representations. We use hybrid methods to mitigate the pathological cases where we could introduce too many zero elements. Readers are referred to [3] and encouraged to experiment with different sparse data sets to gain more insight into the data dependent performance behavior of the various SpMV kernels presented in this chapter.

It should be clear that both the execution efficiency and memory bandwidth efficiency of the parallel SpMV kernels depends on the distribution of the input data

matrix. This is quite different from most of the kernels we have studied so far. However, such data-dependent performance behavior is quite common in real-world applications. This is one of the reasons why parallel SpMV is such an important parallel pattern. It is simple and yet it illustrates an important behavior in many complex parallel applications.

The advantage of sparse matrix representation over dense matrix representation is realized when the matrix is very sparse. Dense matrices have the advantage of having implicit indices and being easy to tile in order to optimize their movement throughout the memory hierarchy. On the other hand, sparse matrix storage formats store explicit indices that need to be accessed, and they are more difficult to apply tiling techniques to. These penalties are worthwhile only when the sparsity is very high and the number of zero elements removed is very large.

In general, the FLOPS rating achieved by either CPUs or GPUs are much lower for sparse matrix computation than for dense matrix computation. People are often surprised by the low FLOPS rating of this type of computation on both CPUs and GPUs. After reading this chapter, one should be no longer be surprised.

Because of the importance of sparse matrix computations, the cuSPARSE library [4] provides GPU-accelerated routines for performing basic linear algebra computations on a number of sparse matrix storage formats. This chapter gives readers the right background needed to effectively use such libraries, or to develop more sophisticated formats and routines that are tailored to their specific sparse computations of interest.

17.9 Exercises

1. Consider the following sparse matrix:

   ```
   1  0  7  0
   0  0  8  0
   0  4  3  0
   2  0  0  1
   ```

   Represent it in each of the following formats: (a) COO, (b) CSR, (c) ELL, and (d) JDS.
2. Given a sparse matrix of integers with m rows, n columns, and z non-zeros. How many integers are needed to represent the matrix in (a) COO, (b) CSR, (c) ELL, and (d) JDS. If the information provided is not enough to answer, indicate what information is missing.
3. Implement the code to convert from COO to CSR using fundamental parallel computing primitives, including histogram and prefix sum.
4. Implement the host code for producing the hybrid ELL-COO format and using it to perform SpMV. Launch the ELL kernel to execute on the device, and compute the contributions of the COO elements on the host.
5. Implement a kernel that performs parallel SpMV using a matrix stored in the JDS format.

References

[1] M.R. Hestenes, E. Stiefel, et al., Methods of Conjugate Gradients for Solving Linear Systems, vol. 49, NBS, Washington, DC, 1952.
[2] J.R. Rice, R.F. Boisvert, Solving Elliptic Problems Using ELLPACK, vol. 2, Springer Science & Business Media, 2012.
[3] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, pp. 1–11.
[4] NVIDIA, cuSPARSE, http://docs.nvidia.com/cuda/cusparse, 2025.

This page intentionally left blank

CHAPTER

18 Graph traversal

With special contributions from John Owens and Juan Gómez-Luna

CONTENTS

18.1 Background 425
18.2 Breadth-first search 429
18.3 Vertex-centric parallelization of BFS 431
18.4 Edge-centric parallelization of BFS 436
18.5 Improving work efficiency with frontiers 438
18.6 Reducing contention with privatization 442
18.7 Reducing launch overhead with cooperative groups 445
18.8 Other optimizations 448
18.9 Summary 449
18.10 Exercises 449
References 451

A graph is a data structure that represents the relations between entities. The entities involved are represented as vertices and the relations are represented as edges. Many important real-world problems are naturally formulated as large-scale graph problems and can benefit from massively parallel computation. Prominent examples include social networks and driving direction map services. There are multiple strategies for parallelizing graph computations, some of which are centered on processing vertices in parallel while others are centered on processing edges in parallel. Graphs are intrinsically related to sparse matrices because graph adjacency matrices tend to be very sparse. Graph computation can be formulated in terms of sparse matrix operations on square matrices. However, one can often improve the efficiency of graph computation by exploiting properties that are specific to the type of graph computation being performed. In this chapter, we will focus on breadth first search, a graph traversal computation that underlies many real-world applications.

18.1 Background

A graph data structure represents the relations between entities. For example, in social media, the entities are users and the relations are connections between users. As another example, in driving direction map services, the entities are locations and the

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00029-9
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

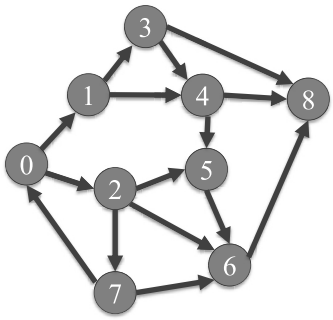
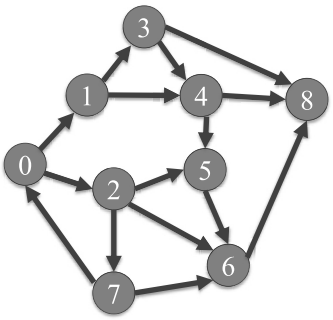


FIGURE 18.1

A simple graph example with nine vertices and fifteen directional edges.



destination vertex

source vertex	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									

FIGURE 18.2

Adjacency matrix representation of the simple graph example.

relations are the roadways between them. Some relations are bidirectional, such as friend connections in a social network. Other relations are directional, such as one-way streets in a road network. In this chapter, we will focus on directional relations. Bidirectional relations can be represented with two directional relations, one for each direction.

Fig. 18.1 shows an example of a simple graph with directional edges. A directional relation is represented as an arrowed edge going from a source vertex to a destination vertex. We assign a unique number to each vertex, also called the vertex id. There is one edge going from vertex 0 to vertex 1 and one going from vertex 0 to vertex 2, etc.

An intuitive representation of a graph is an adjacency matrix. In an adjacency matrix A, if there is an edge going from a source vertex i to a destination vertex j, the value of element $A_{i,j}$ of the adjacency matrix is 1. Otherwise, it is 0. Fig. 18.2 shows the adjacency matrix for the simple graph in Fig. 18.1. We see that $A_{1,3}$ and $A_{4,5}$ are both 1 since there are edges going from vertex 1 to vertex 3 and from vertex

4 to vertex 5. For clarity, we leave the 0 values out of the adjacency matrix. That is, if an element is empty, its value is understood to be 0.

If a graph with N vertices is fully connected, i.e., every vertex is connected with all other vertices, each vertex should have $N - 1$ outgoing edges. There should be a total of $N \cdot (N - 1)$ edges, since there is no edge going from a vertex to itself. For example, if our nine-vertex graph were fully connected, there should be 8 edges going out of each vertex. There should be a total of 72 edges. Obviously, our graph is much less connected; each vertex has three or fewer outgoing edges. Such a graph is referred to as being sparsely connected. That is, the average number of outgoing edges from each vertex is much smaller than $N - 1$.

At this point, the reader has most likely made the correct observation that sparsely connected graphs can probably benefit from a sparse matrix representation. As we have seen in Chapter 17, using a compressed representation of the matrix can drastically reduce the amount of storage required and the number of wasted operations on the zero elements. Indeed, many real-world graphs are sparsely connected. For example, in a social network, the average number of connections for each user is much smaller than the total number of users. This makes the number of non-zero elements in the adjacency matrix much smaller than the total number of elements.

Fig. 18.3 shows three representations of our simple graph example using three different storage formats: CSR, CSC, and COO. We will refer to the row indices and offsets arrays as the `src` and `srcPtrs` arrays, respectively, and the column indices and offsets arrays as the `dst` and `dstPtrs` arrays, respectively. If we take CSR as an example, recall that in a CSR representation of a sparse matrix, each row offset in the `rowPtrs` array gives the starting location for the non-zero elements in a row. Similarly, in a CSR representation of a graph, each source vertex offset in the `srcPtrs` array gives the starting location of the outgoing edges of the vertex. For example, `srcPtrs[3]=7` gives the starting location of the non-zero elements in row 3 of the original adjacency matrix. Also, `srcPtrs[4]=9` gives the starting location of the non-zero elements in row 4 of the original matrix. Thus, we expect to find the non-zero data for row 3 in `data[7]` and `data[8]` and the column indices (destination vertices) for these elements in `dst[7]` and `dst[8]`. These are the data and column indices for the two edges leaving vertex 3. The reason we call the column index array `dst` is that the column index of an element in the adjacency matrix gives the destination vertex of the represented edge. In our example, we see that the destination of the two edges for source vertex 3 are `dst[7]=4` and `dst[8]=8`. We leave it as an exercise to the reader to draw similar analogies for the CSC and COO representations.

Note that the data array in this example is unnecessary. Since the value of all its elements is 1, we do not need to store it. We can make the data implicit, i.e., whenever a non-zero element exists, we can just assume it is 1. For example, the existence of each column index in the destination array of a CSR representation implies that an edge exists. However, in some applications, the adjacency matrix may store additional information about the relationship, such as the distance between two locations or the date when two social network users became connected. In those applications, the data array will need to be explicitly stored.

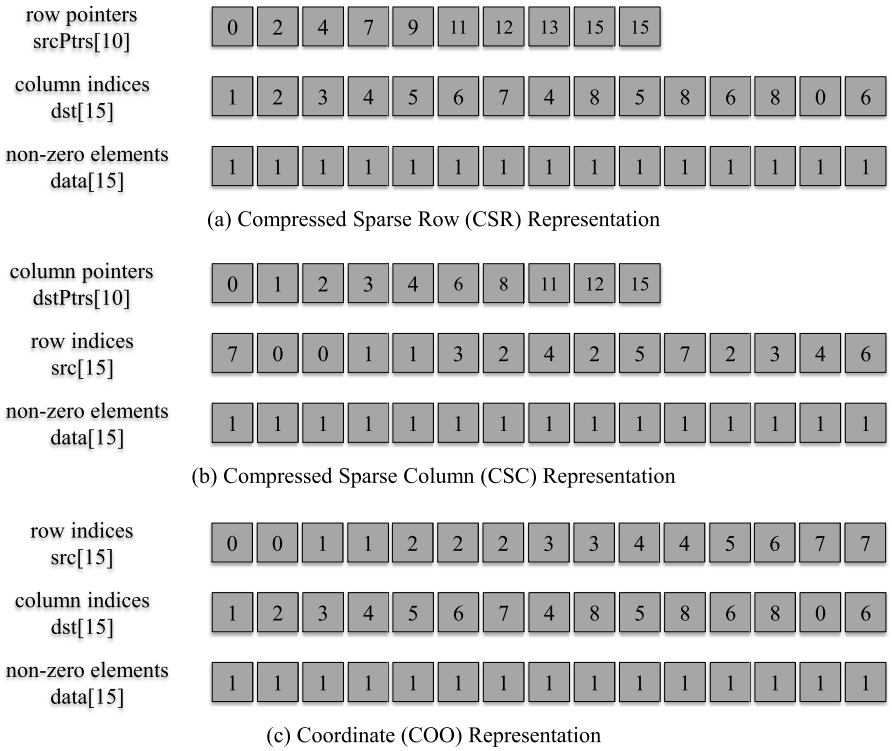


FIGURE 18.3

Three sparse matrix representations of the adjacency matrix: (a) CSR, (b) CSC, and (c) COO.

Sparse representation can lead to significant savings in storing the adjacency matrix. For our example, assuming that the data array can be eliminated, the CSR representation requires storage for 25 locations, in contrast with $9^2 = 81$ locations if we stored the entire adjacency matrix. For real-life problems where a very small fraction of the adjacency matrix elements are non-zero, the savings can be tremendous.

Different graphs may have drastically different structures. One way to characterize these structures is to look at the distribution of the number of edges connected to each vertex (the "vertex degree"). A road network, expressed as a graph, would have a relatively uniform degree distribution with a low average degree per vertex, because each road intersection (vertex) would typically have a low number of roads (edges) connected to it. On the other hand, a graph of social network followers, where each incoming edge represents a "follow", has a much broader distribution of vertex degrees, with a large-degree vertex representing a popular social network user. The structure of the graph may influence the choice of algorithm to implement a particular graph application.

Recall from Chapter 17 that each sparse matrix representation gives different accessibility to the represented data. Hence, the choice of which representation to use

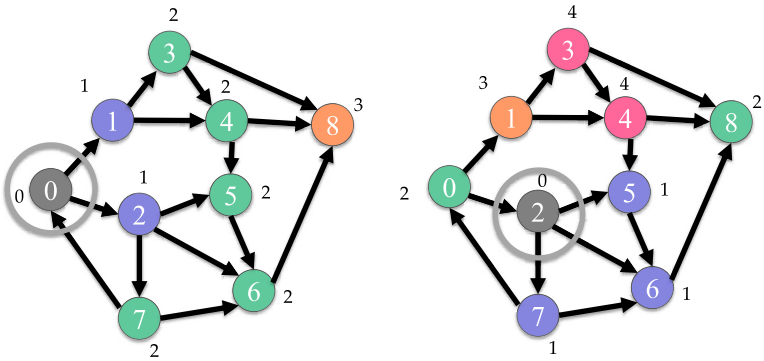


FIGURE 18.4

(a and b) Two examples of breadth-first search results for two different root vertices. The labels adjacent to each vertex indicate the number of hops ("depth") from the root vertex.

for the graph has implications on which information about the graph is made easily accessible to the graph traversal algorithm. The CSR representation gives easy access to the outgoing edges of a given vertex. The CSC representation gives easy access to the incoming edges of a given vertex. The COO representation gives easy access to the source and destination vertices of a given edge. Therefore, the choice of the graph representation comes hand-in-hand with the choice of the graph traversal algorithm. We demonstrate this concept throughout this chapter by examining different parallel implementations of breadth first search, a widely used graph traversal computation.

18.2 Breadth-first search

An important graph traversal computation is breadth-first search (BFS). Breadth-first search is often used to discover the shortest number of edges that one needs to traverse in order to go from one vertex to any other vertex of the graph. In the graph example in Fig. 18.1, we may need to find all the alternative routes that we could take going from the location represented by vertex 0 to that represented by vertex 5. By visual inspection, we see that there are three possible paths: 0→1→3→4→5, 0→1→4→5, and 0→2→5, with 0→2→5 being the shortest. There are different ways of summarizing the outcome of a BFS traversal. One way is, given a vertex referred to as the root, to label each vertex with the smallest number of edges that one needs to traverse to go from the root to that vertex. The outcome of BFS can also be useful for understanding the topology of a graph.

Fig. 18.4(a) shows the desired breadth-first search result with vertex 0 as the root. Through one edge, we can get to vertices 1 and 2. Thus, we label these vertices as belonging to level 1. By traversing another edge, we can get to vertices 3 (through vertex 1), 4 (through vertex 1), 5 (through vertex 2), 6 (through vertex 2), and 7

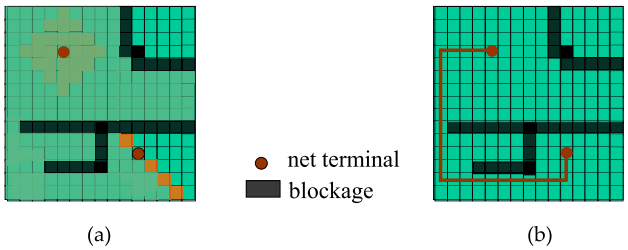


FIGURE 18.5

Maze routing in integrated circuits – an application for breadth-first search: (a) breath-first search, (b) identifying a routing path.

(through vertex 2). Thus, we label these vertices as belonging to level 2. Finally, by traversing one more edge, we can get to vertex 8 (through any of vertices 3, 4 or 6).

The breadth-first search result would be quite different with another vertex as the root. Fig. 18.4(b) shows the desired result of breadth-first search with vertex 2 as the root. The level 1 vertices are 5, 6, and 7. The level 2 vertices are 8 (through vertex 6) and 0 (through vertex 7). Only vertex 1 is at level 3 (through vertex 0). Finally, the level 4 vertices are 3 and 4 (both through vertex 1). It is interesting to note that the outcome is quite different for each vertex even though we moved the root to a vertex that is only one edge away from the original root.

One can view the labeling actions of BFS as constructing a BFS tree that is rooted in the root node of the search. The tree consists of all the labeled vertices and only the edges traversed during the search that go from a vertex at one level to vertices at the next level. Once we have all the vertices labeled with their level, we can easily find a path from the root vertex to any of the vertices where the number of edges traveled is equivalent to the level. For example, in Fig. 18.4(b), we see that vertex 1 is labeled as level 3, so we know that the smallest number of edges between the root (vertex 2) and vertex 1 is 3. If we need to find the path, we can simply start from the destination vertex and trace back to the root. At each step, we select the predecessor whose level is one less than the current vertex. If there are multiple predecessors with the same level, we can randomly pick one. Any vertex thus selected would give a sound solution. The fact that there are multiple predecessors to choose from means that there are multiple equally good solutions to the problem. In our example, we can find a shortest path from vertex 2 to vertex 1 by starting from vertex 1, choosing vertex 0, then vertex 7, and then vertex 2. Therefore, a solution path is $2 \rightarrow 7 \rightarrow 0 \rightarrow 1$. This of course assumes that each vertex has a list of the source vertices of all the incoming edges so that one can find the predecessors of a given vertex.

Fig. 18.5 shows an important application of breadth-first search in computer aided design (CAD). When designing an integrated circuit chip, there are many electronic components that need to be connected to complete the design. The connectors of these components are called net terminals. Fig. 18.5(a) shows two such net terminals as round dots, one belongs to a component in the upper left part and the other belongs to another component in the lower right part of the chip. Assume that the design requires

that these two net terminals be connected. This is done by running, or routing, a wire of a given width from the first net terminal to the second net terminal.

The routing software represents the chip as a grid of wiring blocks where each block can potentially serve as a piece of a wire. A wire can be formed by extending in either the horizontal or the vertical direction. For example, the black J-shaped obstacle in the lower half of the chip consists of 21 wiring blocks and connects three net terminals. Once a wiring block is used as part of a wire, it can no longer be used as part of any other wires. Furthermore, it forms a blockage for wiring blocks around it. No wires can be extended from a used block's lower neighbor to its upper neighbor, or from its left neighbor to its right neighbor, etc. Once a wire is formed, all other wires must the routed around it. Routing blocks can also be occupied by circuit components, which imposes the same blockage constraint as when they are used as part of a wire. This is why the problem is called a maze routing problem. The previously formed circuit components and wires form a maze for the wires that are yet to be formed. The maze routing software finds a route for each additional wire given all the constraints from the previously formed components and wires.

The maze routing application represents the chip as a graph. The routing blocks are vertices. An edge from vertex i to vertex j indicates that one can extend a wire from block i to block j. Once a block is occupied by a wire or a component, it is either marked as a blockage vertex or taken away from the graph, depending on the design of the application. Fig. 18.5 shows that the application solves the maze routing problem with a breadth-first search from the root net terminal to the target net terminal. This is done by starting with the root vertex and labeling the vertices into levels. The immediate vertical or horizontal neighbors (a total of four) that are not blockages are marked as level 1. We see that all four neighbors of the root are reachable and will be marked as level 1. The neighbors of level 1 vertices that are neither blockages nor visited by the current search will be marked as level 2. The reader should verify that there are four level-1 vertices, eight level-2 vertices, and twelve level-3 vertices, etc. in Fig. 18.5(a). As we can see, the breadth-first search essentially forms a wavefront of vertices for each level. These wavefronts start small for level 1 but can grow very large very quickly in a few levels.

Fig. 18.5(b) shows that once the breadth-first search is complete, we can form a wire by finding a shortest path from the root to the target. As explained earlier, this can be done by starting with the target vertex and tracing back to the predecessors whose levels are one lower than the current vertex. Whenever there are multiple predecessors that have equivalent levels, there are multiple routes that are of the same length. One could design heuristics to choose the predecessor in such a way that minimizes the difficulty of constraints for wires that are yet to be formed.

18.3 Vertex-centric parallelization of BFS

A natural way to parallelize graph algorithms is to perform operations on different vertices or edges in parallel. In fact, many parallel implementations of graph algorithms can be classified as vertex-centric or edge-centric. A vertex-centric parallel

implementation assigns threads to vertices and has each thread perform an operation on its vertex, which usually involves iterating over the neighbors of that vertex. Depending on the algorithm, the neighbors of interest may be those that are reachable via the outgoing edges, the incoming edges, or both. In contrast, an edge-centric parallel implementation assigns threads to edges and has each thread perform an operation on its edge, which usually involves looking up the source and destination vertices of that edge. In this section, we look at two different vertex-centric parallel implementations of BFS: one that iterates over outgoing edges and one that iterates over incoming edges. In Section 18.4, we look at an edge-centric parallel implementation of BFS and compare.

The parallel implementations we will look at follow the same strategy when it comes to iterating over levels. In all implementations, we start by labeling the root vertex as belonging to level 0. We then call a kernel to label all the neighbors of the root vertex as belonging to level 1. After that, we call a kernel to label all the unvisited neighbors of the level 1 vertices as belonging to level 2. Then, we call a kernel to label all the unvisited neighbors of the level 2 vertices as belonging to level 3. This process continues until no new vertices are visited and labeled.

The reason a separate kernel is called for each level is that we need to wait until all vertices in a previous level are labeled before proceeding to label vertices in the next level. Otherwise, we risk labeling a vertex incorrectly. In the rest of this section, we focus on implementing the kernel that is called for each level. That is, we will implement a BFS kernel that, given a level, labels all the vertices that belong to that level based on the labels of the vertices from previous levels. We will revise this approach in Section 18.7 where we will iterate over multiple BFS levels with a single kernel.

The first vertex-centric parallel implementation assigns each thread to a vertex to iterate over the vertex's outgoing edges. Each thread first checks if its vertex belongs to the previous level. If so, the thread iterates over the outgoing edges to label all the unvisited neighbors as belonging to the current level. This vertex-centric implementation is often referred to as a *top-down* or *push* implementation.[1] Since this implementation requires accessibility to the outgoing edges of a given source vertex (i.e., non-zero elements of a given row of the adjacency matrix), a CSR representation is needed.

Fig. 18.6 shows the kernel code for the vertex-centric push implementation, while Fig. 18.7 shows an example of how this kernel performs a traversal from level 1 (previous level) to level 2 (current level). The kernel starts by assigning a thread to each vertex (line 03) and each thread ensures that its vertex id is within bounds (line 04). Next, each thread checks if its vertex belongs to the previous level (line 05). In Fig. 18.7, only the threads assigned to vertices 1 and 2 pass this check. The threads

[1] If we are constructing a BFS tree, this implementation can be seen as assigning threads to parent vertices in the BFS tree in search of their children, hence the name top-down. This terminology assumes that the root of the tree is on the top and the leaves of the tree are at the bottom. Push refers to each active vertex's action of pushing its depth via its outgoing edges to all its neighbors.

```
01  __global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
02                              unsigned int* newVertexVisited, unsigned int currLevel) {
03      unsigned int vertex = blockIdx.x*blockDim.x + threadIdx.x;
04      if(vertex < csrGraph.numVertices) {
05          if(level[vertex] == currLevel - 1) {
06              for(unsigned int edge = csrGraph.srcPtrs[vertex];
07                      edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
08                  unsigned int neighbor = csrGraph.dst[edge];
09                  if(level[neighbor] == UNVISITED) {
10                      level[neighbor] = currLevel;
11                      *newVertexVisited = 1;
12                  }
13              }
14          }
15      }
16  }
```

FIGURE 18.6

A vertex-centric push (top-down) BFS kernel.

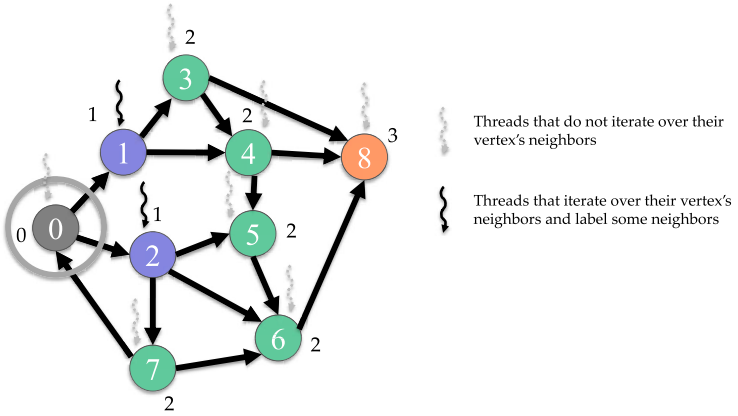


FIGURE 18.7

Example of a vertex-centric push BFS traversal from level 1 to level 2.

that pass this check then use the CSR `srcPtrs` array to locate the outgoing edges of the vertex and iterate over them (lines 06-07). For each outgoing edge, the thread finds the neighbor at the destination of the edge using the CSR `dst` array (line 08). The thread then checks if the neighbor has not been visited (line 09). To enable this check, all vertex levels are initially set to a special value `UNVISITED`. For example, `UNVISITED` could be set to the largest possible unsigned integer `cuda::std::numeric_limits<unsigned>`. Hence, a thread can tell if a neighbor has not been visited if the value of its level is still `UNVISITED`. If the neighbor has not been visited, the thread labels the neighbor as belonging to the current level (line 10). If the thread succeeds in labeling a vertex as visited, it sets a flag indicating that a new vertex has been visited (line 11). This flag is used by the launching code to decide whether a new grid needs to be launched to process a new level or we have reached the end.

Note that multiple threads may label the same vertex as belonging to the current level, and multiple threads may assign 1 to the flag, which means that these two

```
01  __global__ void bfs_kernel(CSCGraph cscGraph, unsigned int* level,
02                  unsigned int* newVertexVisited, unsigned int currLevel) {
03      unsigned int vertex = blockIdx.x*blockDim.x + threadIdx.x;
04      if(vertex < cscGraph.numVertices) {
05          if(level[vertex] == UNVISITED) {
06              for(unsigned int edge = cscGraph.dstPtrs[vertex];
07                      edge < cscGraph.dstPtrs[vertex + 1]; ++edge) {
08                  unsigned int neighbor = cscGraph.src[edge];
09                  if(level[neighbor] == currLevel - 1) {
10                      level[vertex] = currLevel;
11                      *newVertexVisited = 1;
12                      break;
13                  }
14              }
15          }
16      }
17  }
```

FIGURE 18.8

A vertex-centric pull (bottom-up) BFS kernel.

situations are technically race conditions: multiple threads access the same memory location without ordering and at least one of the accesses is a write. As discussed in Chapter 14, such race conditions are benign in practice because the operation is idempotent, having the same effect regardless of how many times it is applied. However, although this code may work in practice, it violates the C++ memory model and is not guaranteed to work. Conservative programmers should use an atomic operation for these two operations.

The second vertex-centric parallel implementation assigns each thread to a vertex to iterate over the vertex's incoming edges. Each thread first checks if its vertex has been visited yet. If not, the thread will iterate over the incoming edges to find if any of the neighbors belong to the previous level. If the thread finds a neighbor that belongs to the previous level, the thread will label its vertex as belonging to the current level. This vertex-centric implementation is often referred to as a *bottom-up* or *pull* implementation.[2] Since this implementation requires accessibility to the incoming edges of a given destination vertex (i.e., non-zero elements of a given column of the adjacency matrix), a CSC representation is needed.

Fig. 18.8 shows the kernel code for the vertex-centric pull implementation, while Fig. 18.9 shows an example of how this kernel performs a traversal from level 1 to level 2. The kernel starts by assigning a thread to each vertex (line 03) and each thread ensures that its vertex id is within bounds (line 04). Next, each thread checks if its vertex has not been visited yet (line 05). In Fig. 18.9, the threads assigned to vertices 3 to 8 all pass this check. The threads that pass this check use the CSC `dstPtrs` array to locate the incoming edges of the vertex and iterate over them (lines 06-07). For each incoming edge, the thread finds the neighbor at the source of the edge using

[2] If we are constructing a BFS tree, this implementation can be seen as assigning threads to potential child vertices in the BFS tree in search of their parents, hence the name bottom-up. Pull refers to each vertex's action of reaching back to its predecessors and pulling active status from them.

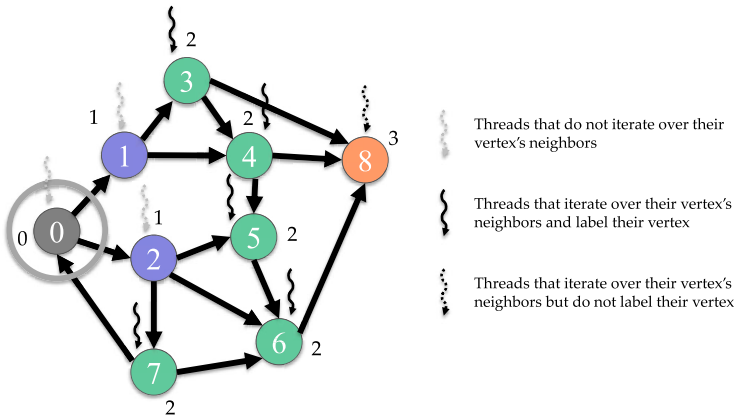


FIGURE 18.9

Example of a vertex-centric pull (bottom-up) traversal from level 1 to level 2.

the CSC `src` array (line 08). The thread then checks if the neighbor belongs to the previous level (line 09). If so, the thread labels its vertex as belonging to the current level (line 10) and sets a flag indicating that a new vertex has been visited (line 11). The thread also breaks out of the loop (line 12).

The justification for breaking out of the loop is as follows. For a thread to establish that its vertex is in the current level, it is sufficient for the thread's vertex to have one neighbor in the previous level. Therefore, it is unnecessary for the thread to check the rest of the neighbors. Only the threads whose vertices do not have any neighbors in the previous level end up looping over the entire neighbors list. In Fig. 18.9, only the thread assigned to vertex 8 loops over the entire neighbor list without breaking.

When comparing the push and pull vertex-centric parallel implementations, there are two key differences to consider that have an important impact on performance. The first difference is that in the push implementation, a thread loops over its vertex's entire neighbor list, whereas in the pull implementation, a thread may break out of the loop early. For graphs with low-degree and low-variance such as road networks or CAD circuit models, this difference may not be important because the neighbor lists are small and similar in size. However, for graphs with high-degree and high-variance such as social networks, the neighbor lists are long and may vary substantially in size resulting in high load imbalance and control divergence across threads. For this reason, breaking out of the loop early can provide substantial performance gains by reducing load imbalance and control divergence.

The second important difference between the two implementations is that in the push implementation, only the threads assigned to vertices in the previous level loop over their neighbor list, whereas in the pull implementation, all the threads assigned to any unvisited vertex loop over their neighbor list. For earlier levels, we expect to have a relatively small number of vertices per level and a large number of unvisited vertices in the graph. For this reason, the push implementation typically performs better for

earlier levels because it iterates over fewer neighbor lists. In contrast, for later levels, we expect to have more vertices per level and fewer unvisited vertices in the graph. Moreover, the chances of finding a visited neighbor in the pull approach and exiting the loop early are higher. For this reason, the pull implementation typically performs better for later levels.

Based on this observation, a common optimization is to use the push implementation for earlier levels, then switch to the pull implementation for later levels. This approach is often referred to as a *direction-optimized* implementation. The choice of when to switch between implementations usually depends on the type of graph. Graphs with low average degree and variance usually have many levels and it takes a while to reach a point where the levels have many vertices and a substantial number of vertices have already been visited. On the other hand, graphs with high average degree and variance usually have few levels, and the levels grow very quickly. Such graphs where it takes only a few levels to get from any vertex to any other vertex are usually referred to as small world graphs. Because of these properties, switching from the push implementation to the pull implementation usually happens much earlier for graphs with high average degree than for those with low average degree.

Recall that the push implementation uses a CSR representation of the graph, whereas the pull implementation uses a CSC representation of the graph. For this reason, if a direction-optimized implementation is to be used, both a CSR and a CSC representation of the graph need to be stored. In many applications, such as social networks or maze routing, the graph is undirected which means that the adjacency matrix is symmetric. In this case, the CSR and CSC representations are equivalent so only one of them needs to be stored and can be used by both implementations.

18.4 Edge-centric parallelization of BFS

In this section, we look at an edge-centric parallel implementation of BFS. In this implementation, each thread is assigned to an edge. It checks if the source vertex of the edge belongs to the previous level and if the destination vertex of the edge is unvisited. If so, it labels the unvisited destination vertex as belonging to the current level. Since this implementation requires accessibility to the source and destination vertices of a given edge (i.e., row and column indices of a given non-zero), a COO data structure is needed.

Fig. 18.10 shows the kernel code for the edge-centric parallel implementation, while Fig. 18.11 shows an example of how this kernel performs a traversal from level 1 to level 2. The kernel starts by assigning a thread to each edge (line 03) and each thread ensures that its edge id is within bounds (line 04). Next, each thread finds the source vertex of its edge using the COO `src` array (line 05) and checks if the vertex belongs to the previous level (line 06). In Fig. 18.11, only the threads assigned to the outgoing edges of vertices 1 and 2 pass this check. The threads that pass this check locate the neighbor at the destination of the edge using the COO `dst` array (line 07) and check if the neighbor has not been visited (line 08). If not, the thread labels the

```
01  __global__ void bfs_kernel(COOGraph cooGraph, unsigned int* level,
02                              unsigned int* newVertexVisited, unsigned int currLevel) {
03      unsigned int edge = blockIdx.x*blockDim.x + threadIdx.x;
04      if(edge < cooGraph.numEdges) {
05          unsigned int vertex = cooGraph.src[edge];
06          if(level[vertex] == currLevel - 1) {
07              unsigned int neighbor = cooGraph.dst[edge];
08              if(level[neighbor] == UNVISITED) {
09                  level[neighbor] = currLevel;
10                  *newVertexVisited = 1;
11              }
12          }
13      }
14  }
```

FIGURE 18.10

An edge-centric BFS kernel.

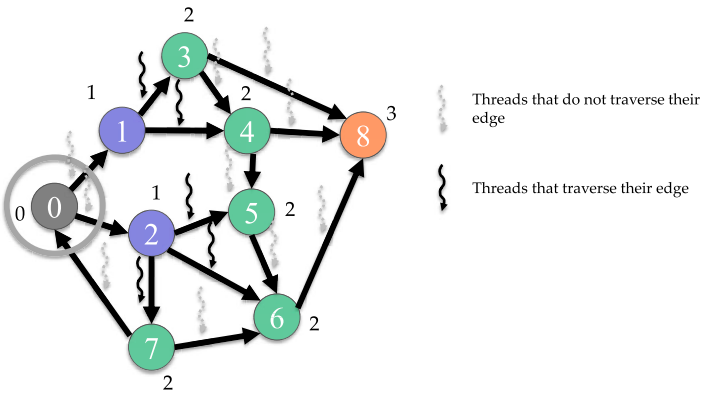


FIGURE 18.11

Example of an edge-centric traversal from level 1 to level 2.

neighbor as belonging to the current level (line 09). Finally, the thread sets a flag indicating that a new vertex has been visited (line 10).

The edge-centric parallel implementation has two main advantages over the vertex-centric parallel implementations. The first advantage is that the edge-centric implementation exposes more parallelism. In the vertex-centric implementations, if the number of vertices is small, we may not launch enough threads to fully occupy the device. Since a graph typically has many more edges than vertices, the edge-centric implementation can launch more threads. Hence, the edge-centric implementation is usually more suitable for small graphs.

The second advantage of the edge-centric implementation over the vertex-centric implementations is that it exhibits less load imbalance and control divergence. In the vertex-centric implementations, each thread iterates over a different number of edges depending on the degree of the vertex it is assigned to. In contrast, in the edge-centric implementation, each thread traverses only one edge. With respect to the vertex-centric implementation, the edge-centric implementation is an example

of rearranging the mapping of threads to work/data to reduce control divergence, as discussed in Chapter 6. The edge-centric implementation is usually more suitable for graphs with high average degree and that have a large variation in the degrees of their vertices.

The disadvantage of the edge-centric implementation is that it checks every edge in the graph. In contrast, the vertex-centric implementations can skip an entire edge-list if they determine that a vertex is not relevant for the level. For example, consider the case where some vertex v has n edges and is not relevant for a particular level. In the edge-centric implementation, our launch includes n threads, one for each edge, and each of these threads independently inspects v and discovers that the edge is irrelevant. In contrast, in the vertex-centric implementations, our launch only includes one thread for v that skips all n edges after inspecting v once to determine that it is irrelevant. Another disadvantage of the edge-centric implementation is that it uses COO which requires more storage space to store the edges compared to CSR and CSC which are used by the vertex-centric implementations.

The reader may have noticed that the code examples in the previous section and this one resemble our implementations of SpMV in Chapter 17. In fact, with a slightly different formulation, we can express a BFS level iteration entirely in terms of SpMV and a few other vector operations, where the SpMV operation is the dominant operation. Beyond BFS, many other graph computations can also be formulated in terms of sparse matrix computations using the adjacency matrix [1]. Such a formulation is often referred to as the *linear algebraic formulation* of graph problems and is the focus of an API specification called GraphBLAS [2]. The advantage of linear algebraic formulations is that they can leverage mature and highly optimized parallel libraries for sparse linear algebra to perform graph computations. The disadvantage of linear algebraic formulations is that they may miss out on optimizations that take advantage of specific properties of the graph algorithm in question.

18.5 Improving work efficiency with frontiers

In the approaches discussed in the previous two sections, we have been checking every vertex or edge in every iteration for their relevance to the level in question. The advantage of this strategy is that the kernels are highly parallel and do not require any synchronization across threads. However, the disadvantage is that many unnecessary threads are launched every iteration and a lot of wasted work is performed making the approaches not work efficient. For example, in the vertex-centric implementations, we launch a thread for every vertex in the graph, many of which simply discover that the vertex is not relevant and do not perform any work. Similarly, in the edge-centric implementation, we launch a thread for every edge in the graph, many of which simply discover that the edge is not relevant and do not perform any useful work.

To quantify the work efficiency of these approaches, let us consider a graph with n vertices and m edges that has a diameter d, i.e., d BFS levels excluding level 0.

```
01  __global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
02              unsigned int* prevFrontier, unsigned int* currFrontier,
03              unsigned int numPrevFrontier, unsigned int* numCurrFrontier,
04              unsigned int currLevel) {
05      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
06      if(i < numPrevFrontier) {
07          unsigned int vertex = prevFrontier[i];
08          for(unsigned int edge = csrGraph.srcPtrs[vertex];
09                  edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
10              unsigned int neighbor = csrGraph.dst[edge];
11              if(visitVertexAtomically(neighbor, level, currLevel)) {
12                  cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
13                      numCurrFrontier_ref(*numCurrFrontier);
14                  unsigned int currFrontierIdx =
15                      numCurrFrontier_ref.fetch_add(1, cuda::memory_order_relaxed);
16                  currFrontier[currFrontierIdx] = neighbor;
17              }
18          }
19      }
20  }
```

FIGURE 18.12

A vertex-centric push (top-down) BFS kernel with frontiers.

Ideally, every vertex and edge is visited only once, making the ideal work complexity of BFS $O(n + m)$. In the vertex-centric push approach, every vertex is checked at every level, but each edge is only traversed once when its source vertex matches the previous level. Hence, the work performed is $O(d \cdot n + m)$. In the vertex-centric pull approach, every vertex is checked at every level, but also, every edge is checked in multiple levels up to the last level as long as its destination vertex is not visited. Hence, the work performed is $O(d \cdot n + d \cdot m))$. In the edge-centric approach, every edge is checked at every level. Hence, the work performed is $O(d \cdot m)$. None of these work complexities matches the ideal work complexity, which shows that all the implementations are not work-efficient.

In this section, we aim to improve the work efficiency of parallel BFS by avoiding launching unnecessary threads and eliminating the redundant checks that they perform in each iteration. We will focus on the vertex-centric push approach that was presented in Section 18.3. Recall that in the vertex-centric push approach, for each level, a thread is launched for each vertex in the graph. The thread checks if its vertex is in the previous level, and if so, it labels all the vertex's unvisited neighbors as belonging to the current level. On the other hand, the threads whose vertices are not in the current level do not do anything. Ideally, these threads should not even be launched. To avoid launching these threads, we can have the threads processing the vertices in the previous level collaborate to construct a frontier of the vertices that they visit. Hence, for the current level, threads only need to be launched for the vertices in that frontier [3]. In this way, each vertex is only checked once and each edge is only traversed once, which results in $O(n + m)$ work complexity.

Fig. 18.12 shows the kernel code for the vertex-centric push implementation that uses frontiers, while Fig. 18.13 shows an example of how this kernel performs a traversal from level 1 to level 2. A key distinction from the previous approach is that the kernel takes additional parameters to represent the frontiers. The additional

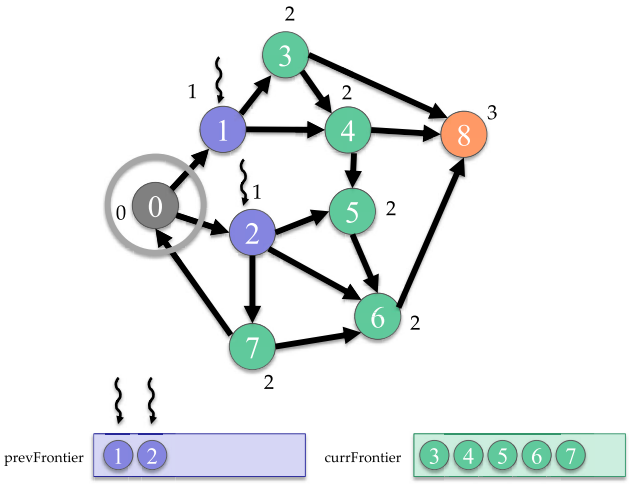


FIGURE 18.13

Example of a vertex-centric push (top-down) BFS traversal from level 1 to level 2 with frontiers.

parameters include the arrays `prevFrontier` and `currFrontier` to store the vertices in the previous and current frontiers, respectively. They also include pointers to the counters `numPrevFrontier` and `numCurrFrontier` that store the number of vertices in each frontier. Note that the flag for indicating that a new vertex has been visited is no longer needed. Instead, the host can tell that the end has been reached when the number of vertices in the current frontier is 0.

We now look at the body of the kernel in Fig. 18.12. The kernel starts by assigning a thread to each element of the previous frontier (line 05) and each thread ensures that its element id is within bounds (line 06). In Fig. 18.13, only vertices 1 and 2 are in the previous frontier, so only two threads are launched. Each thread loads its element from the previous frontier, which contains the index of the vertex that it is processing (line 07). The thread uses the CSR `srcPtrs` array to locate the outgoing edges of the vertex and iterate over them (lines 08-09). For each outgoing edge, the thread finds the neighbor at the destination of the edge using the CSR `dst` array (line 10). The thread then calls a device function `visitVertexAtomically` that checks if the neighbor has not been visited, and if not, labels the neighbor as belonging to the current level (line 11). The implementation of `visitVertexAtomically` will be discussed shortly. If the function returns that it succeeded in labeling the neighbor, the thread must add the neighbor to the current frontier. To do so, the thread atomically increments the size of the current frontier (lines 12-15) and inserts the neighbor at the corresponding location (line 16). The size of the current frontier needs to be incremented atomically because multiple threads may be incrementing it simultaneously, so we need to ensure that no race condition takes place.

We now turn our attention to the `visitVertexAtomically` device function that is called on line 11. As a thread iterates through the neighbors of its vertex, it checks

```
01  __device__ inline unsigned int visitVertexAtomically(unsigned int vertex,
02      unsigned int*level, unsigned int currLevel) {
03      cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
04          level_ref(level[vertex]);
05      unsigned int unvisited = UNVISITED;
06      unsigned int visitSuccess = level_ref.compare_exchange_strong(unvisited,
07          currLevel, cuda::memory_order_relaxed, cuda::memory_order_relaxed);
08      return visitSuccess;
09  }
```

FIGURE 18.14

A device function that visits a vertex atomically using an atomic compare-and-swap operation.

if the neighbor has been visited, and if not, it labels the neighbor as belonging to the current level. In the vertex-centric push kernel without frontiers in Fig. 18.12, this checking and labeling operation is performed without atomic operations (09-10). In that implementation, if multiple threads check the old label of the same unvisited neighbor before any of them can label it, multiple threads may end up labeling the neighbor. Since all threads are labeling the neighbor with the same label (the operation is idempotent), it is okay to allow the threads to label the neighbor redundantly. In contrast, in the frontier-based implementation in Fig. 18.12, each thread not only labels the unvisited neighbor, but also adds it to the frontier. Hence, if multiple threads observe the neighbor as unvisited, they will all add the neighbor to the frontier, causing it to be added multiple times. If the neighbor is added multiple times to the frontier, it will be processed multiple times in the next level, which is redundant and wasteful.

To avoid having multiple threads observe the neighbor as unvisited, the checking and updating of the neighbor's label should be performed atomically. In other words, we must check if the neighbor is not visited, and if not, label it as part of the current level all in one atomic operation. An atomic operation that can perform all of these steps is *compare-and-swap*, which is provided by the C++ function `compare_exchange_strong`. The `visitVertexAtomically` device function uses this C++ function to visit the vertex.

Fig. 18.14 shows the implementation of the `visitVertexAtomically` device function. First, we create an atomic reference to `level[vertex]`, i.e., the level of the vertex that we would like to visit (lines 03-04). Using this reference, we invoke the `compare_exchange_strong` function to atomically update the level value (lines 06-07). The function takes four parameters. The first parameter is the value we want to compare the referenced data to. In our case, we would like to compare `level[vertex]` to `UNVISITED` to check if the vertex is unvisited. However, we cannot pass `UNVISITED` directly because it is a constant and the function expects a reference to an object. For this reason, we create an integer `unvisited` and initialize it to the constant `UNVISITED` (line 05), then pass the integer as the first parameter (line 06). The second parameter is the value we would like to set the referenced data to if the comparison succeeds. In our case, we would like to set `level[vertex]` to the current level if the comparison succeeds, so we pass `currLevel` as the second parameter (line 07).

The third and fourth parameters are the ordering requirements of the memory accesses performed by the atomic operations in the case of success (third parameter) and failure (fourth parameter), respectively. In our case, the atomic operation does not need to be ordered in a particular way with respect to other independent memory accesses performed by the kernel. For this reason, we select `cuda::memory_order_relaxed` in both cases (line 07). The return value of the `compare_exchange_strong` indicates if the swap operation was successful. We take this value and return it as the result of the device function (line 08), indicating if the vertex level was successfully updated.

As mentioned earlier, the advantage of this frontier-based approach over the approach described in the previous section is that it reduces redundant work by only launching threads to process the relevant vertices. On the other hand, the disadvantage of this frontier-based approach is the overhead of the long-latency atomic operations, especially when these operations contend on the same data. For the compare-and-swap operation used to update the vertex levels, we expect the contention to be moderate because only some threads, not all, will visit the same unvisited neighbor. However, for the atomic addition operation used to increment the frontier size, we expect the contention to be high because all threads increment the same counter to add vertices to the same frontier. In the next section, we look at how this contention can be reduced.

18.6 Reducing contention with privatization

The pattern of having different threads insert elements into a frontier should remind the reader of the unstable filter pattern covered in Chapter 12. Two optimizations that we applied to unstable filter to reduce the contention of atomic operations were coalesced atomic operations and privatization. These two optimizations are also applicable to our frontier-based BFS kernel. As mentioned in Chapter 12, we can assume that the coalescing of atomic operations is applied by the compiler. In this section, we will focus on how to leverage privatization to reduce the contention of atomic operations when adding vertices to the current iteration's frontier.

Recall from Chapter 6 that privatization reduces contention of atomic operations by applying partial updates to a private copy of the data, then updating the public copy when done. Privatization can be applied in the context of concurrent frontier updates (increments to `numCurrFrontier`) to reduce the contention on inserting into the frontier. We can have each thread block maintain its own private frontier throughout the computation and update the public frontier when done. Hence, threads will only contend on the same data with other threads in the same block. Moreover, the private frontier and its counter can be stored in shared memory, which enables lower-latency atomic operations on the counter and stores to the private frontier. Furthermore, when the private frontier in shared memory is stored to the public frontier in global memory, the accesses can be coalesced.

```
__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
            unsigned int* prevFrontier, unsigned int* currFrontier,
            unsigned int numPrevFrontier, unsigned int* numCurrFrontier,
            unsigned int currLevel) {

    // Initialize private frontier
    __shared__ unsigned int currFrontier_s[PRIVATE_FRONTIER_CAPACITY];
    __shared__ unsigned int numCurrFrontier_s;
    if(threadIdx.x == 0) {
        numCurrFrontier_s = 0;
    }
    __syncthreads();

    // Perform BFS
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < numPrevFrontier) {
        unsigned int vertex = prevFrontier[i];
        for(unsigned int edge = csrGraph.srcPtrs[vertex];
                edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
            unsigned int neighbor = csrGraph.dst[edge];
            if(visitVertexAtomically(neighbor, level, currLevel)) {
                cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
                    numCurrFrontier_s_ref(numCurrFrontier_s);
                unsigned int currFrontierIdx_s =
                    numCurrFrontier_s_ref.fetch_add(1, cuda::memory_order_relaxed);
                if(currFrontierIdx_s < PRIVATE_FRONTIER_CAPACITY) {
                    currFrontier_s[currFrontierIdx_s] = neighbor;
                } else {
                    numCurrFrontier_s = PRIVATE_FRONTIER_CAPACITY;
                    cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
                       numCurrFrontier_ref(*numCurrFrontier);
                    unsigned int currFrontierIdx =
                       numCurrFrontier_ref.fetch_add(1, cuda::memory_order_relaxed);
                    currFrontier[currFrontierIdx] = neighbor;
                }
            }
        }
    }
    __syncthreads();

    // Allocate in public frontier
    __shared__ unsigned int currFrontierStartIdx;
    if(threadIdx.x == 0) {
        cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
            numCurrFrontier_ref(*numCurrFrontier);
        currFrontierStartIdx = numCurrFrontier_ref.fetch_add(numCurrFrontier_s,
            cuda::memory_order_relaxed);
    }
    __syncthreads();

    // Commit to public frontier
    for(unsigned int currFrontierIdx_s = threadIdx.x;
          currFrontierIdx_s < numCurrFrontier_s; currFrontierIdx_s += blockDim.x) {
        unsigned int currFrontierIdx = currFrontierStartIdx + currFrontierIdx_s;
        currFrontier[currFrontierIdx] = currFrontier_s[currFrontierIdx_s];
    }

}
```

FIGURE 18.15

A vertex-centric push (top-down) BFS kernel with privatization of frontiers.

Fig. 18.15 shows the kernel code for the vertex-centric push implementation that uses privatized frontiers, while Fig. 18.16 illustrates the privatization of the frontiers. The kernel starts by declaring a private frontier for each thread block in shared mem-

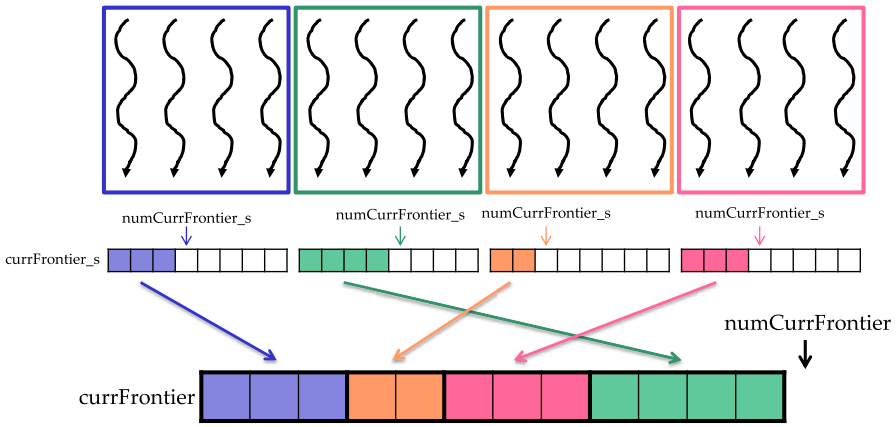


FIGURE 18.16

Privatization of frontiers example.

ory (lines 07-08). One thread in the block initializes the private frontier's counter to 0 (lines 09-11) and all threads in the block wait at the `__syncthreads()` barrier for the initialization to complete before they start using the counter (line 12). The next part of the code is similar to the previous version: each thread loads its vertex from the frontier (line 17), iterates over its outgoing edges (lines 18-19), finds the neighbor at the destination of the edge (line 20), and atomically checks if the neighbor is unvisited and visits it if it is unvisited (line 21).

If the thread succeeds in visiting the neighbor, i.e., the neighbor is unvisited, it adds the neighbor to the private frontier. The thread first atomically increments the private frontier counter (lines 22-25). If the private frontier is not full (line 26), the thread inserts the neighbor into the private frontier (line 27). Otherwise, if the private frontier has overflowed, the thread restores the value of the private frontier counter (line 29) and inserts the neighbor to the public frontier by atomically incrementing the public frontier counter (lines 30-33) and storing the neighbor at the corresponding location (line 34).

After all threads in a block have iterated over their vertices' neighbors, they need to store the private frontier elements to the public frontier. First, the threads wait for each other to complete to ensure that no more neighbors will be added to the private frontier (line 39). Next, one thread in the block acts on behalf of the others to allocate space in the public frontier for all the elements in the private frontier (lines 42-48) while all the other threads wait for it (line 49). Finally, the threads iterate over the vertices in the private frontier (lines 52-53) and store them in the public frontier (line 54-55). Notice that the index into the public frontier `currFrontierIdx` is expressed in terms of `currFrontierIdx_s` which is expressed in terms of `threadIdx.x`. Therefore, threads with consecutive thread index values store to consecutive global memory locations, which means that the stores are coalesced.

18.7 Reducing launch overhead with cooperative groups

The BFS kernels that we have developed so far are designed to be called multiple times from the host CPU, once for every BFS level. When frontiers are large, which is the case when processing graphs with high average degree such as social networks, the overhead of launching a grid is negligible compared to the amount of work done by the grid. However, when frontiers are small, which is the case when processing graphs with low average degree such as road networks, the overhead of launching a grid may dominate the overall performance.

To mitigate the overhead of launching multiple grids, we would like to be able to perform the entire computation with just a single grid launch. However, doing so would require the ability to perform a grid-wide barrier synchronization in between levels to ensure that threads have finished finding the vertices in a level before proceeding to the next level. Performing such a grid-wide barrier synchronization is possible with the *cooperative groups* API in CUDA. However, performing a grid-wide barrier synchronization imposes a constraint on the number of thread blocks that the grid can be launched with.

Recall from Chapter 4 that grids may be launched with a larger number of thread blocks than the number that can be executed simultaneously. In this case, the hardware will schedule and execute as many thread blocks as possible simultaneously and bring in more thread blocks as the older ones complete. The problem with this scheduling approach is that we cannot perform an unrestricted grid-wide barrier synchronization across all thread blocks because it could result in a deadlock. Thread blocks that are scheduled would wait for thread blocks that are not schedule to reach the barrier, while thread blocks that are not scheduled would wait for thread blocks that are scheduled to complete to take their place. To be able to perform a grid-wide barrier synchronization, we need to have the guarantee that all thread blocks participating in the barrier are actively executing on the SMs so they can reach the barrier. To provide this guarantee, we cannot launch a larger number of thread blocks than the maximum number that is allowed to execute simultaneously.

To find the maximum number of thread blocks that can execute simultaneously, we can use the CUDA occupancy calculation functions mentioned in Chapter 4. In particular, we can find the maximum number of thread blocks that can execute per SM as follows:

```
int numBlocksPerSM;
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSM,
                          bfs_kernel, numThreadsPerBlock, 0);
```

Here, the `cudaOccupancyMaxActiveBlocksPerMultiprocessor` function takes the kernel of interest `bfs_kernel`, the number of threads per block, and the amount of dynamic shared memory needed per block (0 in this case) as its second, third, and fourth argument. It calculates the maximum number of thread blocks that can execute per SM and writes the result to `numBlocksPerSM` whose pointer is passed as the first argument to the function.

Knowing the maximum number of thread blocks that can execute per SM, to find the maximum that can execute in total, we need to find the number of SMs in the target GPU. As mentioned in Chapter 4, the number of SMs in a GPU can be found by querying the device properties as follows:

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, 0);
int numSMs = deviceProp.multiProcessorCount;
```

We then find the maximum number of thread blocks that can run simultaneously by multiplying the maximum that can run per SM with the number of SMs as follows:

```
unsigned int numBlocks = numSMs*numBlocksPerSM;
```

If we launch the grid with only this number of thread blocks, we can perform a grid-wide barrier synchronization without fearing deadlock because we know that all the thread blocks will execute simultaneously and can reach the barrier.

To be able to use cooperative groups inside the kernel to perform a grid-wide barrier synchronization, we cannot use the usual interface that we introduced in Chapter 2. Instead, we must call the kernel in a special way as follows:

```
void *kernelArgs[] = { (void*)&csrGraph_d, (void*)&level_d,
        (void*)&prevFrontier_d, (void*)&currFrontier_d,
        (void*)&numPrevFrontier, (void*)&numCurrFrontier_d };
cudaLaunchCooperativeKernel((void*)bfs_kernel, numBlocks,
                            numThreadsPerBlock, kernelArgs);
```

We first store all the kernel arguments to a `void*` array. We then call the kernel via the `cudaLaunchCooperativeKernel` API and pass to it the kernel to be called, which is `bfs_kernel`, the number of thread blocks previously computed, the number of threads per block, and the array of kernel arguments.

Having seen how the BFS kernel must be called to use cooperative groups, we can now turn our attention to implementing the kernel itself. Fig. 18.17 shows a BFS kernel that iterates over multiple BFS levels and uses cooperative groups to perform a grid-wide barrier synchronization between levels. Before the kernel is defined, we include the cooperative groups library (lines 01-02), which is needed to use the cooperative groups API. At the beginning of the kernel, we obtain a handle on the grid (line 08) which is useful for synchronizing the grid later on.

The outermost loop of the kernel iterates over the BFS levels as long as the frontier to be processed is not empty (line 10). Most of the body of this loop (lines 12 to 62) resembles the frontier-based kernel from Fig. 18.15, which reads the vertices in the previous frontier, iterates over their neighbors, and atomically visits the unvisited neighbors while inserting them into the current frontier. The main difference from the code in Fig. 18.15 is that in Fig. 18.15, we assumed that each thread handled a single vertex. On the other hand, in Fig. 18.17, the number of threads is limited by the maximum number that can execute simultaneously, so a single thread may

```
#include <cooperative_groups.h>
using namespace cooperative_groups;

__global__ void bfs_kernel(CSRGraph csrGraph, unsigned int* level,
        unsigned int* prevFrontier, unsigned int* currFrontier,
        unsigned int numPrevFrontier, unsigned int* numCurrFrontier) {

    grid_group grid = this_grid();

    for(unsigned int currLevel = 1; numPrevFrontier > 0; ++currLevel) {

        // Initialize private frontier
        __shared__ unsigned int currFrontier_s[PRIVATE_FRONTIER_CAPACITY];
        __shared__ unsigned int numCurrFrontier_s;
        if(threadIdx.x == 0) {
            numCurrFrontier_s = 0;
        }
        __syncthreads();

        // Perform BFS
        for(unsigned int i = grid.thread_rank();
                i < numPrevFrontier; i += grid.num_threads()) {
            unsigned int vertex = prevFrontier[i];
            for(unsigned int edge = csrGraph.srcPtrs[vertex];
                    edge < csrGraph.srcPtrs[vertex + 1]; ++edge) {
                unsigned int neighbor = csrGraph.dst[edge];
                if(visitVertexAtomically(neighbor, level, currLevel)) {
                    cuda::atomic_ref<unsigned int, cuda::thread_scope_block>
                        numCurrFrontier_s_ref(numCurrFrontier_s);
                    unsigned int currFrontierIdx_s =
                        numCurrFrontier_s_ref.fetch_add(1, cuda::memory_order_relaxed);
                    if(currFrontierIdx_s < PRIVATE_FRONTIER_CAPACITY) {
                        currFrontier_s[currFrontierIdx_s] = neighbor;
                    } else {
                        numCurrFrontier_s = PRIVATE_FRONTIER_CAPACITY;
                        cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
                            numCurrFrontier_ref(*numCurrFrontier);
                        unsigned int currFrontierIdx =
                            numCurrFrontier_ref.fetch_add(1, cuda::memory_order_relaxed);
                        currFrontier[currFrontierIdx] = neighbor;
                    }
                }
            }
        }
        __syncthreads();

        // Allocate in public frontier
        __shared__ unsigned int currFrontierStartIdx;
        if(threadIdx.x == 0) {
            cuda::atomic_ref<unsigned int, cuda::thread_scope_device>
                numCurrFrontier_ref(*numCurrFrontier);
            currFrontierStartIdx = numCurrFrontier_ref.fetch_add(numCurrFrontier_s,
                cuda::memory_order_relaxed);
        }
        __syncthreads();

        // Commit to public frontier
        for(unsigned int currFrontierIdx_s = threadIdx.x;
                currFrontierIdx_s < numCurrFrontier_s; currFrontierIdx_s += blockDim.x) {
            unsigned int currFrontierIdx = currFrontierStartIdx + currFrontierIdx_s;
            currFrontier[currFrontierIdx] = currFrontier_s[currFrontierIdx_s];
        }

        // Swap frontiers
        grid.sync();
        numPrevFrontier = *numCurrFrontier;
        grid.sync();
        if(grid.thread_rank() == 0) {
            *numCurrFrontier = 0;
        }
        grid.sync();
        unsigned int* tmp = prevFrontier;
        prevFrontier = currFrontier;
        currFrontier = tmp;

    }

}
```

FIGURE 18.17

A multi-level BFS kernel that uses cooperative groups.

have to handle multiple vertices. For this reason, each thread iterates over the vertices it is responsible for (lines 21-22) based on its position in the grid (obtained via `grid.thread_rank()`) and the total number of threads in the grid (obtained via `grid.num_threads()`).

After the threads of the grid finish constructing the frontier, they must swap the frontiers for the next iteration (lines 65-74) such that the current frontier becomes the previous frontier and the memory used for the previous frontier is repurposed to construct the new current frontier. To do so, the first step is to perform a grid-wide barrier synchronization to ensure that all threads in the grid have finished adding vertices to the current frontier. This grid-wide barrier synchronization is performed by calling the cooperative groups API function `grid.sync()` (line 65). Next, all the threads read the size of the current frontier to become the size of the new previous frontier (line 66). Another grid-wide barrier synchronization (line 67) ensures that all threads finish reading the size of the current frontier before it is reset to 0 by one of the threads (lines 68-70). Yet another grid-wide barrier synchronization (line 71) ensures that the size of the current frontier is reset to 0 before any thread begins adding to it. Finally, the memory buffers for the frontiers are swapped (lines 72-74) before the threads proceed to the next iteration to construct the frontier for the next level.

Note that although three grid-wide barrier synchronizations are used in this implementation, two of them can be eliminated by using a different counter for each level and resetting them all to 0 before the kernel starts. This optimization resembles the double-buffering optimization, but in a more general form. We leave this optimization as an exercise for the reader.

Using cooperative groups for grid-wide barrier synchronization has multiple benefits besides just mitigating the overhead of grid launches. One benefit is that performing all BFS iterations in a single kernel frees up the CPU to do other useful work rather than having to call the kernel for each iteration. Another benefit that is not observed in this chapter's example, but may occur in other contexts, is the ability to keep data in shared memory or registers across grid-wide barrier synchronizations – data that would have otherwise been written to and re-read from global memory if the kernel was terminated and called again. This benefit of fusing kernels to keep data in shared memory or registers is similar to that achieved in Chapter 11 where multiple stages of the grid-wide prefix sum were fused into a single kernel. The difference is that the kernels in Chapter 11 were fused by using grid-wide unidirectional synchronization, as opposed to grid-wide barrier synchronization.

18.8 Other optimizations

Recall that in the vertex-centric implementations, the amount of work to be done by each thread depends on the connectivity of the vertex assigned to it. In some graphs, such as social network graphs, some vertices (celebrities) may have several orders of magnitude higher degrees than others. When this happens, one or a few of the threads can take excessively long and slow down the execution of the entire grid. We have

seen one way to address this issue, which is by using an edge-centric parallel implementation instead. Another way we can potentially address this issue is by sorting the vertices of a frontier into buckets depending on their degree and processing each bucket in a separate kernel with an appropriately sized group of processors. One notable implementation [4] uses three different buckets for vertices with small, medium, and large degrees. The kernel processing the small buckets assigns each vertex to a single thread; the kernel processing the medium buckets assigns each vertex to a single warp; and the kernel processing the large buckets assigns each vertex to an entire thread block. This technique is particularly useful on graphs with a high variation in vertex degrees.

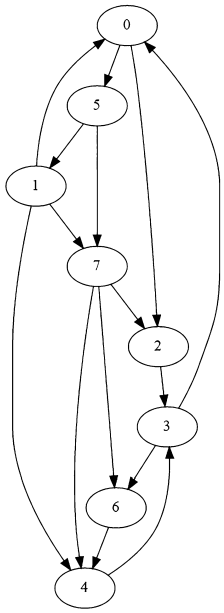
While BFS is among the simplest graph applications, it exhibits the challenges characteristic of more complex applications: problem decomposition for extracting parallelism, taking advantage of privatization, implementing fine-grained load-balancing, and ensuring proper synchronization. Graph computation is applicable to a wide range of interesting problems, particularly in the areas of making recommendations, detecting communities, finding patterns within a graph, and identifying anomalies. One significant challenge is to handle graphs whose size exceeds the memory capacity of the GPU. Another interesting opportunity is to preprocess the graph into other formats or reorder the vertices in the graph before beginning computation in order to expose more parallelism or locality, or facilitate load balancing.

18.9 Summary

In this chapter, we have seen the challenges associated with parallelizing graph traversal computations, using breadth-first search as an example. We started with a brief introduction to the representation of graphs. We discussed the difference between vertex-centric and edge-centric parallel implementations and observed the trade-offs between them. We saw how to eliminate redundant work and improve work efficiency by using frontiers, and we optimized the use of frontiers using privatization. We also showed how cooperative groups can be used to perform a grid-wide barrier synchronization to avoid the overhead of terminating grids and launching new ones. Finally, we briefly discussed other advanced optimizations to improve load balance. Programmers can already find efficient parallel implementations of many graph algorithms in RAPIDS cuGraph [5]. The techniques and optimizations covered in the chapter equip programmers with the understanding to properly use such libraries, as well as the foundations for using GPUs to accelerate new classes of graph algorithms.

18.10 Exercises

1. Consider the following directed unweighted graph:



a. Represent the graph using an adjacency matrix.

b. Represent the graph in the CSR format. The neighbor list of each vertex must be sorted.

c. Parallel BFS is executed on this graph starting from vertex 0 (i.e., vertex 0 is in level 0). For each iteration of the BFS traversal:

- If a vertex-centric push implementation is used:
  - A. How many threads are launched?
  - B. How many threads iterate over their vertex's neighbors?
- If a vertex-centric pull implementation is used:
  - A. How many threads are launched?
  - B. How many threads iterate over their vertex's neighbors?
  - C. How many threads label their vertex?
- If an edge-centric implementation is used:
  - A. How many threads are launched?
  - B. How many threads may label a vertex?
- If a vertex-centric push frontier-based implementation is used:
  - A. How many threads are launched?
  - B. How many threads iterate over their vertex's neighbors?

2. Implement the host code for the direction-optimized BFS implementation described in Section 18.3.

3. Modify the kernel in Fig. 18.17 to use only a single grid-wide barrier synchronization per level instead of three.

References

[1] J. Kepner, J. Gilbert, Graph Algorithms in the Language of Linear Algebra, SIAM, 2011.
[2] T.A. Davis, Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra, ACM Transactions on Mathematical Software 45 (4) (2019) 1–25.
[3] L. Luo, M. Wong, W.-m. Hwu, An effective gpu implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference, 2010, pp. 52–55.
[4] D. Merrill, M. Garland, A. Grimshaw, Scalable gpu graph traversal, ACM SIGPLAN Notices 47 (8) (2012) 117–128.
[5] A. Fender, B. Rees, J. Eaton, Rapids cugraph, in: Massive Graph Analytics, Chapman and Hall/CRC, 2022, pp. 483–493.

This page intentionally left blank

CHAPTER

Convolutional neural networks

19

With special contributions from Juan Gómez-Luna

CONTENTS

19.1 Convolutional neural networks 454
19.2 A CUDA convolutional layer kernel 460
19.3 Formulating convolutional layer as GEMM 463
19.4 CUDNN library 472
19.5 Summary 474
19.6 Exercises 474
References 475

Deep learning [1] is a branch of the machine learning discipline whose studies give computers the ability to learn to perform certain tasks without being explicitly programmed. Machine learning for a task starts with a human developer architecting a parameterized model for given datasets that explicitly or implicitly specify the expected action for each input instance. The model parameters are "learned" by processing the datasets. The complexity and size of the machine learning models can vary widely based on the task being performed and the size of the datasets used.

Deep learning models are designed for complex tasks, such as human facial recognition and natural language processing, whose expected actions can only be captured with and learned from datasets with massive numbers of examples. These models typically employ a deep hierarchy of classifiers that consist of a massive number of parameters to learn a large number of complex features. Thus, they can achieve more accurate pattern recognition results if there is enough training data to allow the system to properly learn the parameter values of all the layers of feature extractors in the model to automatically discover an adequate number of relevant patterns.

While machine learning has existed as a subject of computer science for a long time and has been applied successfully to automate many simple, routine tasks, its deep learning branch had stayed dormant for decades because of the lack of massive datasets and the computing power needed to learn the values of the large number of model parameters from the massive datasets. Since 2012, deep learning based on multi-layer Artificial Neural Networks has gained a great deal of industry attention for two reasons. The first reason is the massive datasets available for training deep learning models from the pervasive use of the internet by both consumers and businesses. The second reason is the availability of ample computing power in GPU

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00030-5
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

computing systems that can effectively and timely learn model parameter values from these massive datasets. The readers may find the introduction to machine learning and neural networks in Appendix B helpful.

This chapter presents the acceleration of the convolutional neural networks (CNNs) whose demonstrated efficacy in computer vision triggered the first wave of deep learning revolution in 2012. In particular, the convolutional layers of the CNNs have a high compute-to-memory-access ratio, and high levels of parallelism, which makes them a perfect candidate for GPU acceleration. We will present a basic CUDA implementation of the forward (inference) path in a convolutional layer and a few optimizations that improve the speed of this basic implementation. We will then show how one can formulate the convolutional layers into matrix multiplication, which can be systematically accelerated using highly optimized hardware and software in modern GPUs.

19.1 Convolutional neural networks

The CNNs, invented in late 1980s [2], are a type of feed-forward networks (Appendix B). By the early 1990s, CNNs had been experimentally applied to automated speech recognition, optical character recognition (OCR), handwriting recognition, and face recognition [3].

However, until the late 1990s, the mainstream of computer vision and that of automated speech recognition had been based on carefully engineered features. The amount of labeled data was insufficient for training a deep-learning system to compete with recognition/classification functions crafted by human experts. Furthermore, it was a common belief that it was computationally infeasible to automatically learn the parameter values of hierarchical feature extractors with enough layers to outperform human-crafted application-specific feature extractors even if the required labeled datasets existed

Interest in deep, feed-forward networks were revived around 2006 by a group of researchers who introduced unsupervised learning methods that could create multilayer, hierarchical feature detectors without requiring labeled data [4,5]. The first major application of this approach was in speech recognition. The breakthrough was made possible by GPUs that allowed researchers to train networks ten times faster than traditional CPUs. This advancement, coupled with the massive amount of media data available online, drastically elevated the position of deep-learning approaches. Despite their success in speech, convolutional neural networks were largely ignored in the field of computer vision until 2012.

In 2012, a group of researchers from the University of Toronto trained a convolutional neural network to classify 1000 different classes in the ILSVRC contest [6]. The network was huge by the norms of the time: it had approximately 60 million parameters and 650,000 neurons. It was trained on 1.2 million high-resolution images from the ImageNet database. The network was trained in only one week on two GPUs using a CUDA-based convolutional neural network library written by Alex

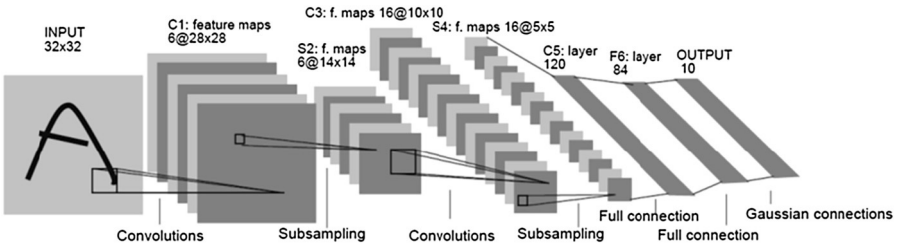


FIGURE 19.1

LeNet-5, a convolutional neural network for handwritten digit recognition. The letter A in the input should be classified as none of the 10 classes (digits).

Krizhevsky [7]. The network achieved break-through results with a winning test error rate of 15.3%. In comparison, the second-place team that used the traditional computer vision algorithms had an error rate of 26.2%. This success triggered a revolution in computer vision, and convolutional neural networks became a mainstream tool in computer vision, natural language processing, reinforcement learning, and many other areas.

We will use LeNet-5, a CNN designed in the late 1980s for digit recognition [3] as a working example. As shown in Fig. 19.1, LeNet-5 is composed from three types of layers: convolutional layers, subsampling layers, and fully connected layers. While these three types of layers continue to be important components of today's neural networks, the convolutional layers account for the majority of the execution time of CNNs. We will focus on the logical design and acceleration of the convolutional layers. Interested readers can refer to Appendix B and the literature for more treatment of other types of CNN layers.

The *forward path* of LeNet-5 receives input on the left side and flows through the layers from left to right. The input to the network is shown as a gray image with a hand-written digit represented as a 2D 32×32 pixel array. The last layer computes the output, which is the probability for the original image to belong to each one of the 10 classes (digits) that the network is set up to recognize.

During inference, the output generated by the forward path of the network is used as the answer (i.e., which digit is recognized from the input image) for the input image. During training, on the other hand, the output of the forward path is compared against the label, i.e., the expected output value for the input image. If the output disagrees with the expected output, the *backward-propagation path* is activated to adjust the model parameters of the network (Appendix B). That is, the forward path is activated during both the training phase and the inference phase. We will therefore use the forward path to illustrate the acceleration of the convolutional layers.[1]

[1] Appendix B also presents the backward-propagation path of CNNs. Interested readers should explore the CUDA kernel implementation of the backward-propagation of the CNN layers by applying the skills that they acquired from studying the forward path.

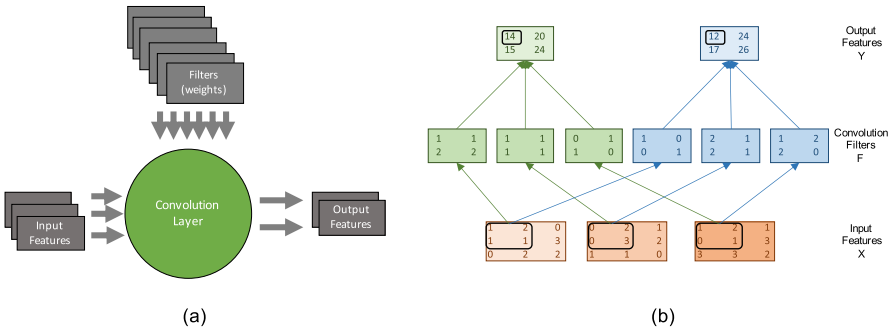


FIGURE 19.2

Forward propagation path of a convolutional layer.

The inputs to and outputs from the CNN layers are referred to as *feature maps*, or simply, *features*. For example, in Fig. 19.1, the computation of the C1 convolutional layer (Convolutions) at the input end of the network is organized to generate six (28 $\times$ 28) output feature maps from the (32 $\times$ 32) INPUT pixel array. The convolutional layer output to be produced for the input feature maps consists of pixels, each of which is produced by performing a convolution between a small local patch of the feature map pixels produced by the previous layer (INPUT in the case of C1) and a set of weights (i.e., a convolution mask as defined in Chapter 7) called a *filter*.

The convolution result is then fed into an activation function like sigmoid (Appendix B) to produce an output pixel in the output feature map. One can think of the convolutional layer as a collection of perceptrons (Appendix B) where each pixel of an output feature map is generated by one of the perceptrons whose inputs are patches of pixels in the input feature maps. For the purpose of this chapter, we will focus on the efficient generation of the convolution result. Once an efficient kernel for generating the convolution result is in place, it is straightforward to add the activation functions either as a separate kernel or at the end the same kernel. That is, for the purpose of this chapter, the value of each output pixel is the sum of convolution results from the corresponding patches in all input feature maps.

Fig. 19.2 shows a small convolutional layer example. There are three input feature maps, two output feature maps, and six filters. Different input and output feature map pairs in a layer use different filters. Since there are three input feature maps and two output feature maps in Fig. 19.2, we need 3 $\times$ 2 = 6 filters in the bank. For the C3 layer of LeNet in Fig. 19.1, there are 6 input feature maps and 16 output feature maps. Thus, a total of 6 $\times$ 16 = 96 filters are used in C3's filter bank.

Fig. 19.2(b) illustrates more details of the calculations done by a convolutional layer. We omitted the activation function for the output pixels for simplicity. We show that each output feature map is the sum of convolutions of all input feature maps. For example, the upper left corner element of output feature map 0 (value 14)

```
1  void convLayer_forward(int M, int C, int H, int W, int K,
       float* X, float* F, float* Y) {
2     int H_out = H - K + 1;
3     int W_out = W - K + 1;
4     for(int m = 0;  m < M;  m++) // for each output feature
          map
5        for(int h = 0; h < H_out; h++) // for each output
             element
6           for(int w = 0; w < W_out; w++) {
7              Y[m, h, w] = 0;
8              for(int c = 0;  c < C; c++) // sum over all
                   input feature maps
9                 for(int p = 0; p < K; p++) // KxK filter
10                   for(int q = 0; q < K; q++)
11                      Y[m, h, w] +=  X[c, h + p, w + q] * F
                            [m, c, p, q];
12          }
13 }
```

FIGURE 19.3

A C implementation of the forward propagation path of a convolutional layer.

is calculated as the convolution between the circled patches of the input feature maps and the corresponding filters[2]:

$$(1,2,1,1)\cdot(1,1,2,2)+(0,2,0,3)\cdot(1,1,1,1)+(1,2,0,1)\cdot(0,1,1,0)=$$
$$1+2+2+2+0+2+0+3+0+2+0+0=14 \quad (19.1)$$

Although not shown in Fig. 19.1, all 2D filters used in LeNet-5 are 5×5 convolution filters. Recall from Chapter 7 that generating a convolution output image from an input image and a convolution filter requires one to make assumptions about the "ghost cells". Instead of making such assumptions, the LeNet-5 design simply uses four elements at the right edge and four elements at the bottom edge of each input feature map as "ghost cells". This reduces the size of each dimension by four: four at the right and four at the bottom. As a result, we see that for layer C1, the 32×32 INPUT image results in an output feature map that is a 28×28 image. Fig. 19.1 illustrates this computation by showing that a pixel in the C1 layer is generated from a square (5×5 although not explicitly shown) patch of INPUT pixels.

Fig. 19.3 shows `convLayer_forward`, a sequential C function implementation of the forward propagation path of a convolutional layer. The input feature maps are stored in a 3D array parameter `X` whose size is `C`×`H`×`W`, where `C` is the second input

[2] Although we use integer values for elements in the feature maps and filter bank for simplicity, they are floating numbers in reality.

parameter that specifies the number of input feature maps (i.e., number of channels), `H` is the third input parameter that specifies the height of each input map image, and `W` is the fourth input parameter that specifies the width of each input map image (line 1). That is, the highest dimension index for `X` selects one of the feature maps (often referred to as channels) and the indices of the lower two dimensions select one of the pixels in an input feature map. For example, the input feature maps for the C1 layer of Fig. 19.1 are stored in `X` whose dimensions are 1×32×32 since there is only one input feature map (INPUT in Fig. 19.1) that consists of 32 pixels in each of the *x* and *y* dimensions. This also reflects the fact that one can think of the 2D input feature maps to a layer altogether form a 3D input feature map, which is also referred to as the *input tensor* to the layer.

The output feature maps of a convolutional layer are to be stored in a 3D `M×(H-K+1)×(W-K+1)` array parameter `Y`, where `M` is the number of output feature maps and `K` is the height (and width) of each 2D filter (line 1). For example, the output feature maps for the C1 layer in Fig. 19.1 is stored in `Y` whose dimensions are 6×28×28 since C1 generates six output feature maps using the 5×5 filters.

The filter bank is stored in a 4D `M×C×K×K` array parameter `W`. There are `M×C` filters, one for each pair of input feature map and output feature map. Filter `F[m, c, _, _]` is used when calculating the contribution of input feature map `X[c, _, _]` to output feature map `Y[m, _, _]`, where "_" represents an index that ranges from 0 to `K`. Recall that each output feature map is the sum of convolutions of all input feature maps.

Each iteration of the outermost (`m`) for-loop (lines 4-12) generates an output feature map. Each iteration of the next two levels (`h` and `w`) of for-loops (lines 5-12) generates one pixel of the current output feature map. The next three loop levels (lines 8-11) perform the convolution between the input feature maps and the associated filter banks before accumulating the convolution result for each input feature map into the output feature map (line 11).

To complete our example, the convolutional layer C3 in Fig. 19.1 has 6 input feature maps and 16 output feature maps. Each input feature map is a 14×14 image and each output feature map is a 10×10 image. This layer has $6 \times 16 = 96$ filter banks and each filter bank has $5 \times 5 = 25$ weights. The output of C3 is passed into the subsampling layer S4 which generates 16 5×5 output feature maps. Finally, the last convolutional layer C5 which uses a bank of 16 *120 = 1920 5×5 filters to generate 120 one-pixel output feature maps from its 16 input feature maps.

The output feature maps from C5 are passed through fully connected layer F6 which has 84 output units, where each output is fully connected to all inputs. The output is computed as a product of a weight matrix with an input vector, then a bias is added and the output is passed through sigmoid. For F6 in our example, the weight matrix (`W`) is a 120×84 matrix. In summary, the output is an 84-element vector `Y6 = sigmoid(W * X + b)`. The reader should recognize that this is equivalent to 84 perceptrons and each perceptron takes all 120 one-pixel values generated by the C5 layer as its input. We leave the detailed implementation of a fully connected layer as an exercise.

```
1  void convLayer_batched(int N, int M, int C, int H, int W,
       int K, float* X, float* F, float* Y) {
2     int H_out = H - K + 1;
3     int W_out = W - K + 1;
4     for(int n = 0;  n < N;  n++) // for each sample in the
          batch
5        for(int m = 0;  m < M;  m++) // for each output
             feature map
6           for(int h = 0; h < H_out; h++) // for each output
                element
7              for(int w = 0; w < W_out; w++) {
8                 Y[n, m, h, w] = 0;
9                 for(int c = 0;  c < C; c++) // sum over all
                       input feature maps
10                   for(int p = 0; p < K; p++) // K x K
                         filter
11                      for(int q = 0; q < K; q++)
12                         Y[n,m,h,w] = Y[n,m,h,w] + X[n, c,
                               h+p, w+q] * F[m,c,p,q];
13             }
14 }
```

FIGURE 19.4

Forward path of a batched convolutional layer.

The final stage is an output layer that uses Gaussian connections [3] to generate a vector of 10 elements, which correspond to the probability that the input image contains one of the 10 digits (0 through 9).

Batched Convolution Layer. To make full use of the computation resources, implementers of CNNs often resort to batched execution by collecting multiple inputs and running them through the network as a batch. The input feature maps and output feature maps generated for each input in the batch will be collectively referred to as a *sample* in the batch. During batched execution, batching allows the user to launch a much larger grid with more thread blocks, which can significantly improve the utilization of GPU execution resources. Batched CNN execution is especially popular for the training phase since one must run a massive number of inputs through the CNN during training. It is very natural to divide up the large number of training inputs into batches before running them through the network.

Fig. 19.4 shows the revised forward path implementation of a convolutional layer that generates the output feature maps for all the samples of a batch. When compared to Fig. 19.3, there is an additional outer loop (line 4) that iterates through all the samples in the batch. Note that `X` and `Y` have an additional dimension, which consists of the different samples in the batch, whereas `F` does not because the same weights are used across samples in the same batch. We will use this batched implementation as the base for the CUDA kernels.

19.2 A CUDA convolutional layer kernel

The computation pattern in training a convolutional neural network is like matrix multiplication: it is both compute intensive and highly parallel. We can process different samples in a mini-batch, different output feature maps for the same sample, and different elements for each output feature map in parallel. In Fig. 19.4, the `n`-loop (line 4, over samples in a mini-batch), `m`-loop (line 5, over output feature maps), the nested `h-w`-loops (lines 6-7, over pixels of each output feature map) are all parallel loops in that their iterations can be executed in parallel. These four loop levels together offer a massive level of parallelism.

The innermost three loop levels, the `c`-loop (over the input feature maps or channels) and the nested `p-q`-loops (over the weights in a filter bank), also offer a significant level of parallelism. However, to parallelize them, one would need to make heavy use of atomic operations in accumulating into the `Y` elements since different iterations of these loop levels would need to perform read-modify-write on the same `Y` elements. Therefore, we will keep these loops serial unless we really need more parallelism.

Assuming that we exploit the four levels of "easy" parallelism `(n, m, h, w)` in the convolutional layer, the total number of parallel iterations is the product `N * M * H_out * W_out`. This high degree of available parallelism makes the convolutional layer a promising candidate for GPU acceleration. We can easily design a kernel with thread organizations designed to capture the parallelism.

We need to first make some high-level design decisions on the thread organization. Assume that we will have each thread to compute one element of one output feature map. We will use 2D thread blocks, where each thread block computes a tile of `TILE_WIDTH * TILE_WIDTH` pixels in one output feature map. For example, if we set `TILE_WIDTH` = 16, we would have a total 256 threads per block. This captures part of the nested `h-w`-loop level parallelism in generating the pixels of each output feature map. By assigning each thread block to generate `TILE_WIDTH * TILE_WIDTH` pixels in one of the output feature maps, the "easy" parallelism can support up to `N * M * (H_out/TILE_WIDTH) * (W_out/TILE_WIDTH)` thread blocks.

Thread blocks can be organized into a 3D grid in several different ways. Each option designates the grid dimensions to capture the `n`, `m`, and `h-w` parallelism in different combinations. We will present the details of one of the options and leave it as an exercise for the reader to explore different options and evaluate the potential pros and cons of each option. The option that we present in detail is as follows:

1. The first dimension (X) corresponds to the (`M`) output features maps covered by each block. This assignment allows the threads in a thread block to use the X dimension of their common block index (`blockIdx.x`) to identify the output feature map that is assigned to them.
2. The second dimension (Y) reflects the location of a block's output tile inside the output feature map selected by X. This assignment allows the threads in a thread block to use the Y dimension of their block index (`blockIdx.y`) to produce their assigned tile within their assigned output feature map.

```
1  #define TILE_WIDTH 16
2  W_grid = W_out/TILE_WIDTH; // number of tiles in the
       horizontal dimension of each output feature map
3  H_grid = H_out/TILE_WIDTH; // number of tiles in the
       vertical dimension of each output feature map
4  T = H_grid * W_grid; // Total number of tiles in each
       output feature map
5  dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
6  dim3 gridDim(M, T, N);
7  ConvLayerForward_Kernel<<<gridDim, blockDim>>>(...);
```

FIGURE 19.5

Host code for launching a convolutional layer kernel.

3. The third dimension (Z) in the grid corresponds to inputs (`N`) in the batch. This assignment allows each block to use the Z dimension of their block index (`blockIdx.z`) to select one of the samples in the batch as its input.

Fig. 19.5 shows the host code that launches a kernel based on the thread organization proposed above. The number of blocks in the X and Z dimensions of the grid are straightforward; they are simply `M`, the number of output feature maps, and `N`, the number of samples in a batch. That is, the X index and Z index of a thread block select the output feature map and the input sample of the current batch for the thread block to work on.

The arrangement in the Y dimension is a little more complex and is illustrated in Fig. 19.6. Ideally, we would like to dedicate two dimensions of the grid indices to the vertical and horizontal tile indices for simplicity. However, we have only one dimension for both since we are using X for the output feature map index and Z for the sample index in a mini-batch. Therefore, we linearize the tile indices to encode both the horizontal and vertical tile indices of output feature map tiles.

In the example in Fig. 19.6, each sample has 4 output feature maps (`M` = 4) and each output feature map consists of 2 × 2 tiles (`W_grid` = 2 in line 2 and `H_grid` = 2 in line 3) of 16 × 16 = 256 pixels each. The grid organization assigns each block to calculate one of these tiles.

We have already assigned each output feature map to the X dimension, which is reflected as the four blocks in the X dimension each corresponding to one of the output feature maps. As shown in the bottom of Fig. 19.6, we linearize the 4 tiles in each output feature map and assign them to the blocks in the Y dimension. That is, tiles (0,0), (0,1), (1,0), and (1,1) are mapped using row-major order to the blocks with `blockIdx.y` values 0, 1, 2, and 3, respectively. Thus, the total number of blocks in the Y dimension is 4 (`T = H_grid * W_grid` = 4 in line 4). Therefore, we will launch a grid with `gridDim(4, 4, N)` in lines 6-7.

Fig. 19.7 shows a kernel based on the thread organization above. Note that in the code, we use multi-dimensional indices in array accesses for clarity. We leave it to

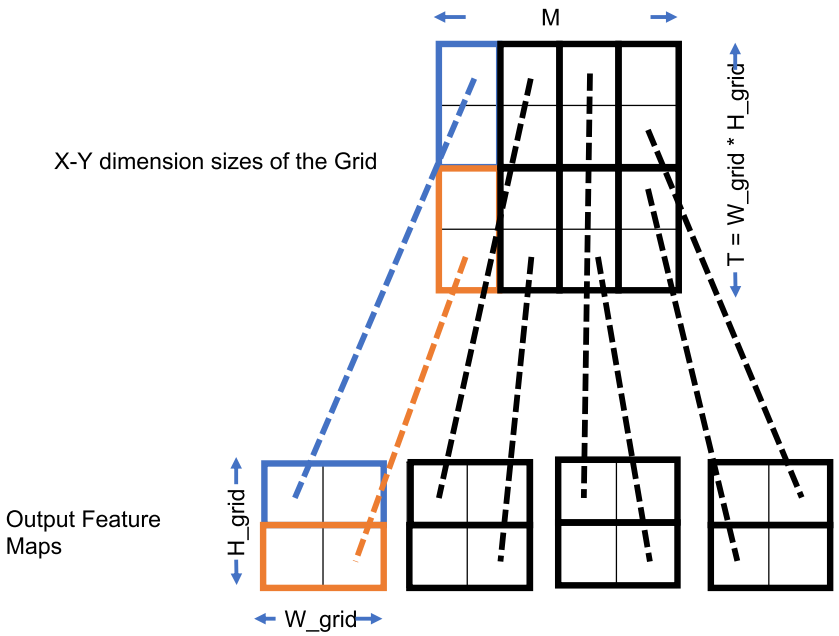


FIGURE 19.6

Mapping output feature map tiles to blocks in the X-Y dimension of the grid. The same mapping is done for all samples in the Z dimension.

```
1  __global__ void
2  ConvLayerForward_Kernel(int C, int W_grid, int K, float* X
       , float* F, float* Y) {
3     int m = blockIdx.x;
4     int h = (blockIdx.y / W_grid) * TILE_WIDTH + threadIdx.
          y;
5     int w = (blockIdx.y % W_grid) * TILE_WIDTH + threadIdx.
          x;
6     int n = blockIdx.z;
7     float acc = 0.;
8     for (int c = 0;  c < C; c++) { // sum over all input
          channels
9        for (int p = 0; p < K; p++) // loop over KxK  filter
10          for (int q = 0; q < K; q++)
11             acc += X[n, c, h + p, w + q] * F[m, c, p, q];
12    }
13    Y[n, m, h, w] = acc;
14 }
```

FIGURE 19.7

Kernel for the forward path of a convolutional layer.

reader to translate this pseudo-code into regular C assuming that `X`, `Y`, and `F` must be accessed via linearized indexing based on row-major layout (Chapter 3).

Each thread starts by generating the `n` (sample within batch), `m` (feature map), `h` (vertical), and `w` (horizontal) indices of its assigned output feature map pixel. The `n` (line 6) and `m` (line 3) indices are straightforward given the host code. For the `h` index calculation in line 4, the `blockIdx.y` value is first divided by `W_grid` to recover the tile index in the vertical direction, as illustrated in Fig. 19.6. This tile index is then expanded by the `TILE_WIDTH` and added to the `threadIdx.y` to form the actual vertical pixel index into the output feature map (line 4). The derivation of the horizontal pixel index is similar (line 5).

Each thread generates an output feature map pixel iteratively. In each iteration, it performs a convolution between a patch of the input feature map pixels and the selected convolution filter bank (lines 7-12). The convolution result is accumulated in the `acc` local variable. The process completes when the thread finishes the convolution for the same patch over all input feature maps. The thread will then assign the final result into the output feature map pixel (line 13).

The kernel in Fig. 19.7 has a high degree of parallelism but consumes too much global memory bandwidth. Like the convolution pattern discussions in Chapter 7, the execution speed of the kernel will be limited by the global memory bandwidth. As we have also seen in Chapter 7, we can use techniques such as shared memory tiling to dramatically reduce the global memory traffic and improve the execution speed of the kernel. These optimizations to the convolution layer kernel are left as an exercise for the reader. Instead, we will present a more systematic approach to reducing memory bandwidth consumption in the following section.

19.3 Formulating convolutional layer as GEMM

The convolutional layer presents multiple opportunities for reducing memory bandwidth consumption through data reuse. Since the execution speed of the kernel shown in Fig. 19.7 is limited by global memory bandwidth, any improvement in memory bandwidth consumption would likely lead to higher execution speed. For example, each input feature map is used in calculating all output feature maps. One can potentially tile the input feature maps and reuse these tiles in calculating multiple output feature maps. However, different thread blocks are used to calculate different output feature maps in the convolution layer kernel in Fig. 19.7. Tiling input feature maps into one thread block's shared memory would not allow other thread blocks to reuse the tiled data. Enabling reuse of input feature map pixels would require major redesign of the kernel.

This section presents an approach proposed by Chellapilla et al. [8] to formulating convolutional layers into equivalent matrix multiplication operations. With this formulation, well-defined tiling techniques (Chapter 15) can be readily used to enable the reuse of the input feature maps as well as the filter banks to reduce global memory bandwidth consumption. In fact, one can just use or slightly adapt one of the

highly optimized GEMM (General Matrix Multiply) kernels from the CUDA linear algebra library cuBLAS to implement high-performance convolutional-layer kernels.

The central idea of the formulation is that each output feature map element is a dot product between elements of the input feature maps and the elements of their corresponding filter banks. Since matrix multiplication consists of dot products between row vectors of the first input and column vectors of the second input, one should be able to arrange the input feature maps and the filter banks into matrices so that each output element of the output feature map is an element of the product matrix. That is, one can make arrangements so that the filter banks needed to generate one output feature map pixel form one row vector of the first input matrix. One can also unfold and duplicate the input feature map pixels in such way that all elements needed to compute one output feature map pixel are stored as one column of the second input matrix.[3] An example of such rearrangements is shown in Fig. 19.8.

Consider a small example convolutional layer which takes as one input $C = 3$ feature maps each of which is of size 3×3 and produces $M = 2$ output features each of which is of size 2×2, as was shown in Fig. 19.2(b) and is shown again at the top of Fig. 19.8 for convenience. It uses $M\times C = 6$ filter banks, where each filter bank is of size 2×2. Since the computation performed for each sample in a batch is completely independent of others in the same batch, Fig. 19.8 shows the input feature maps and the output feature maps of only one sample whose index is assumed to be `n`.

Let's consider the calculation needed to generate output feature map pixel $Y_{n,0,0,0}$, i.e., pixel (0,0) of output feature map 0 of sample `n` shown at the upper left corner of Fig. 19.8, whose value should be 14. All the patches of input feature pixels needed for calculating $Y_{n,0,0,0}$ are circled in the input feature maps. The expression for generating the $Y_{0,0,0}$ is:

$$\begin{aligned} Y_{n,0,0,0} &= (1,1,2,2)\cdot(1,2,1,1)+(1,1,1,1)\cdot(0,2,0,3)+(0,1,1,0)\cdot(1,2,0,1) \\ &= (1+2+2+2)+(0+2+0+3)+(0+2+0+0)=14 \end{aligned} \quad (19.2)$$

where the first input of each inner product is a vector that is formed by linearizing (in row-major order, Chapter 3) the filter bank used for the convolution. For example, the vector formed by linearizing filter bank 0 using row major order is (1, 1, 2, 2), which is the first input to the first inner product. Since the layout of C matrices in the global memory is already in row major order, there is no rearrangement needed inside these filter banks!

All one needs to do is to ensure that the filter banks are concatenated in the order where all filter banks for an output feature with a larger index follow those for an output feature map with a smaller index. Among filter banks for the same output feature map, a filter bank for an input feature map with larger index follows those with smaller index. In our example, the filter banks should be concatenated in the order

[3] See also https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/ for very detailed explanation.

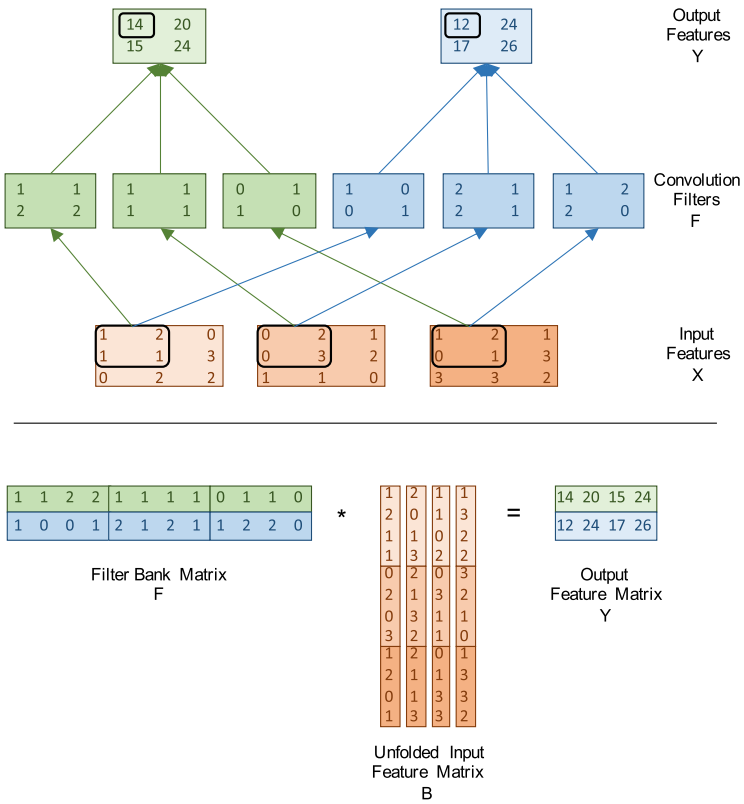


FIGURE 19.8

Formulation of convolutional layer as GEMM. Each sample in the batch has its own input feature maps and output feature maps but share the same filter banks. Since the samples within a batch are completely independent of each other, this figure shows the input feature maps and the output feature maps of only one sample (whose index is assumed to be `n`).

$F_{0,0}$, $F_{0,1}$, $F_{0,2}$, $F_{1,0}$, $F_{1,1}$, $F_{1,2}$. Recall that the dimensions of F array in Fig. 19.3 (line 11) and Fig. 19.7 (line 11) are assumed to be `m`×`c`×`p`×`q`. With the output feature map index as the highest dimension and the input feature map as the second highest dimension, the filter banks are already concatenated in the desired order! With this order, the filter banks form a row major $2 \times (3 * 4) = 2 \times 12$ matrix, as shown on the lower left side of Fig. 19.8. The bottom line is that the filter banks are already organized for the matrix multiplication formulation without any further rearrangement.

The second input to each inner product for calculating $Y_{n,0,0,0}$ is a vector formed by linearizing (in row-major order) a patch of input map X pixels which are circled in Fig. 19.8. For example, the vector formed by linearizing the circled patch in input feature map 0 using row major order is (1, 2, 1, 1), which is the second input to the first inner product. We will further concatenate the linearized vectors from all the

input feature maps into a single vector, which allows us to reformulate the three inner products for calculating $Y_{n,0,0,0}$ into one inner product:

$$\begin{aligned} Y_{n,0,0,0} &= (1,2,1,1,0,2,0,3,1,2,0,1) \cdot (1,1,2,2,1,1,1,1,0,1,1,0) \\ &= 1+2+2+2+0+2+0+3+0+2+0+0 = 14 \end{aligned} \quad (19.3)$$

The first input to the inner product is the vector formed by concatenating $F_{0,0}$, $F_{0,1}$, and $F_{0,2}$. The second input is the concatenation of the linearized patches of the input feature map. If we make the first input vector a row of a matrix F and the second input vector a column of a matrix B, the inner product for producing $Y_{n,0,0,0}$ would be naturally part of the matrix-matrix multiplication of F and B. During matrix multiplication, one row of the filter bank matrix (F) and the column of the input feature matrix (B) will produce one pixel of the output feature map. Since we already know how to form matrix F, we just need to complete the formation of matrix B from the input feature map patches.

Note that there are `H_out * W_out` pixels in each output feature map and each column of the B matrix is used to generate one output map pixel. Therefore, B must consist of `H_out * W_out` columns. In our example, `H_out = 2` and `W_out = 2` so matrix B consists of four columns, as shown in Fig. 19.8. Each column of B will be generated by unfolding and concatenating patches of the input feature maps. In our example, since we have determined that each column of B consists of $3 * 4 = 12$ elements, the B matrix is of size 12×4, as shown in Fig. 19.8.

By ordering the columns of B properly, we can generate the output feature maps in their desired row-major order in `Y`. For example, in the bottom half of Fig. 19.8, the top section of the output feature map matrix is the row-major view of output feature map 0 and the bottom is that of output feature map 1. Both are already in row-major order assumed by the C language so they can be used as input feature maps for the next layer. In general, the output feature map matrix generated by the matrix multiplication is the correct form of `Y[n, m, p, q]` with the `p` and `q` dimensions viewed in their linearized row-major order. The matrix multiplication generates the output feature maps in exactly the form of the input feature maps that is expected by the following or downstream layers in the network.

Cost of Explicitly Unfolding Input Feature Maps

While it is tempting to present the details of rearranging the input feature maps `X` into matrix `B` as our next step, we will first take a pause and analyze the cost of explicitly generating and storing `B`. As we see in Fig. 19.8, the B matrix has $12 * 4 = 48$ elements whereas the three the input feature maps altogether consist of only $3 * 3 * 3 = 27$ elements. That is, there is a significant size increase of $1.78\times$ from `X` to `B`.

The root cause of the size increase is that the patches of input feature map for calculating different pixels of the output feature map overlap with each other due to the nature of convolution. This means that each input feature map pixel can be replicated multiple times as we produce matrix `B`. For example, in Fig. 19.8 the center pixel of each 3×3 input feature map is used four times to compute the four pixels

of an output feature, so it will be duplicated four times when generating the B matrix. The pixels in the middle the edges of each input feature map are used two times so it will be duplicated two times. The four pixels at corners of each input feature map are used only one time and will not need to be duplicated. Therefore, the total number of pixels in the matrix B for one input feature map is 4 * 1 + 2 * 4 + 1 * 4 = 16. Since each original input feature map has only 9 pixels, the explicit generation of matrix B incurs an expansion ratio of $16/9 = 1.78\times$ over the original input feature maps.

In general, the size of the unfolded input feature map matrix B can be derived from the number of input feature map elements required to generate each output feature map element. The height, or the number of rows, of the expanded matrix is the number of input feature elements contributing to each output feature map element, which is C * K * K; each output element is the convolution of K * K elements from each input feature map and there are C input feature maps. In our example, the K is 2 since the filter bank is 2×2 and there are three input feature maps. Thus, the height of the expanded matrix should be 3 * 2 * 2 = 12, which is exactly the height of the matrix shown in Fig. 19.8.

The width, or the number columns, of the unfolded matrix B is the number of elements in each output feature map, i.e., H_out * W_out. In our example, each output feature map is a 2×2 matrix, i.e., H_out = 2 and W_out=2, yielding four columns in the expanded matrix. Note that the number of output feature maps M does not play into the duplication. This is because all output feature maps are computed from the same unfolded input feature map matrix B.

The ratio of expansion for the input feature maps is the size of the expanded matrix over the total size of the original input feature maps. The reader should verify that the expansion ratio is:

$$\frac{C * K * K * H\_out * W\_out}{C * H\_in * W\_in} = \frac{K^2 * H\_out * W\_out}{H\_in * W\_in} \tag{19.4}$$

where H_in and W_in are the height and width of each input feature map. In our example, the ratio is (3 * 2 * 2 * 2 * 2)/(3 * 3 * 3) = 16/9. In general, if the input feature maps and output feature maps are much larger than the filters, the ratio of expansion will approach K^2. While the expansion seems to be modest for our small example, for practical feature map sizes where H_out and W_out are in the range of hundreds or thousands and K can be 5 or larger, the expansion can be easily $20\times$ or more! Furthermore, explicit generation of unfolded matrix B requires an extra step that will further add to global memory bandwidth consumption, negating any reduction in global memory bandwidth consumption from the matrix multiplication formulation. This motivates the implicit approach as follows.

An Implicit Approach to Unfolding Input Feature Maps

To avoid the downside of input feature map size expansion, practical systems that implement convolutional layers with matrix multiplications typically generate the B matrix in a piece-meal manner as part of the tiled matrix multiplication process. The idea is that tiled matrix multiplication kernels, such as the one shown in Fig. 5.9, load

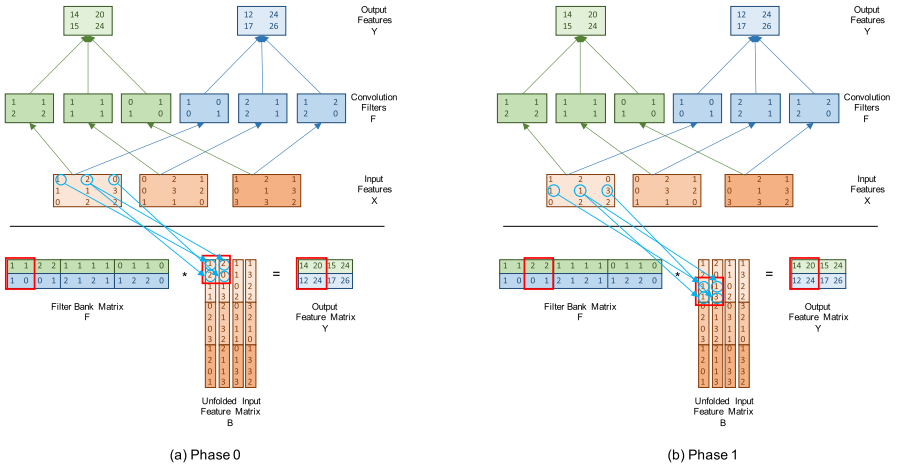


FIGURE 19.9

During each phase of a tiled matrix multiplication that implements a convolutional layer, the B matrix tile elements are loaded from their corresponding X elements.

tiles of the two input matrices before performing a phase of the multiplication for a tile of the output matrix. This presents an opportunity for leaving the input feature maps in their original form in the global memory. When a tile of the `B` matrix is to be loaded from the global memory, we adapt the tile loading code to assemble the `B` matrix tile by loading the pixels from the corresponding locations in the *X* array. In computer science terms, matrix B is only conceptual and is *materialized* on-demand in a tile-by-tile manner during a tiled matrix multiplication process.

The key to the implicit approach is a systematic mapping between the B matrix elements and the *X* array elements. That is, when conceptual `B` element B[u,v] is to be loaded into a tile, one can identify a unique *X* array element to be used. For example, assume that the matrix multiplication is performed as a tiled matrix multiplication with 2×2 tiles, which is illustrated in Fig. 19.9(a).

In this small example, the 2×2 tile configuration will result in launching a grid that consists of two thread blocks with $2 * 2 = 4$ threads each. Each thread generates one output feature map pixel and each block will collaboratively generate a tile of output feature map pixels. We will focus on the thread block that generates the output tile that consists pixels, i.e., `Y[n,0,0,0]`, `Y[n,0,0,1]`, `Y[n,0,1,0]`, and `Y[n,0,1,1]`, i.e., the left half of the Y array in the marked box in Fig. 19.9(a). For the phase 0 of the matrix multiplication, the `B` tile to be loaded consists of `B[0,0]`, `B[0,1]`, `B[1,0]`, and `B[1,1]` (upper left corner of `B` in the marked box in Fig. 19.9(a)). As shown by the arrows in Fig. 19.9, `B[0,0]` can be loaded from `X[n,0,0,0]`, `B[0,1]` from `X[n,0,0,1]`, `B[1,0]` from `X[n,0,0,1]`, and `B[1,1]` from `X[n,0,0,2]`.

Fig. 19.9(b) shows the relevant mappings for phase 1 of the matrix multiplication. The `B` tile to be loaded by the same thread block during this phase will consists of

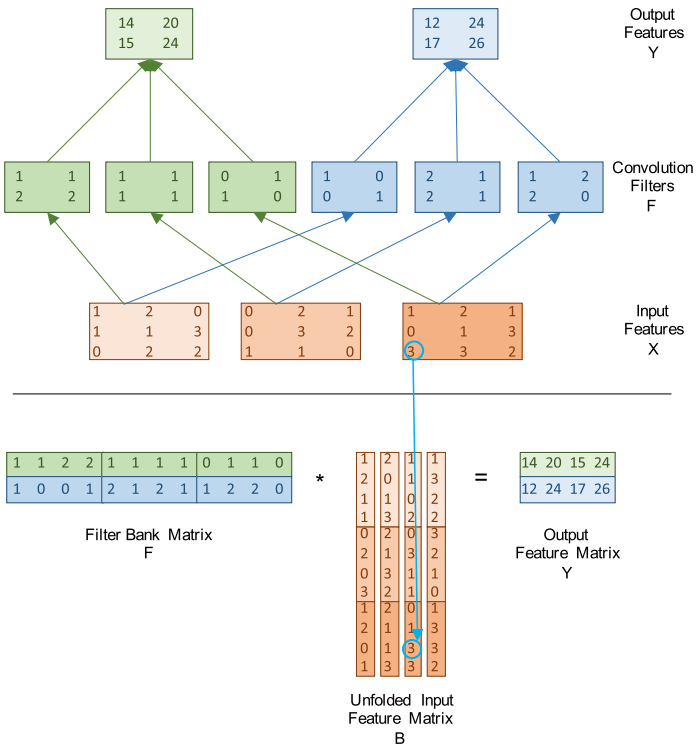


FIGURE 19.10

A mapping example for `B[10,2]`, i.e., `u=10` and `v=2`.

`B[2,0]`, `B[2,1]`, `B[3,0]`, and `B[3,1]`. As shown by the arrows in Fig. 19.9(b), `B[2,0]` can be loaded from `X[n,0,1,0]`, `B[2,1]` from `X[n,0,1,1]`, `B[3.0]` from `X[n,0,1,1]`, and `B[3,1]` from `X[n,0,1,2]`. We will use these two tiles as examples as we derive the general formula for mapping from `B` elements to `X` elements.

Assume that a thread is responsible for `B[u,v]` when its thread block loads a tile. From our discussion above, `v` is a row-major linearized index that identifies the output feature map pixel to receive the contribution from this `B` element. We can recover the last two dimensions of the index into `Y` as `v/W_out`, `v%W_out`. That is, the output feature map element is `Y[n,m,v/W_out,v%W_out]`. These two indexes also identify the beginning position (upper left corner) of the patch of input feature map pixels involved in the convolution operation that generates the output feature map pixel.

The `u` index specifies the input feature map where the patch of pixels comes from and the row-major linearized index for its `B` element inside the patch. We can identify the channel number of the input feature map by $u/(K^2)$. For example, in Fig. 19.10, `u` is equal to 2 for element `B[10,2]`. We know that `B[10,2]` comes from input feature map 2 (c = 2) since $10/2^2 = 2$. Indeed, Fig. 19.10 shows that `B[10,2]` resides in the bottom section of `B`, which comes from input feature map 2.

Within the patch, $u\%K^2$ is the row-major linearized index of the pixel. That is, $(u\%K^2)/K$ is the local row index and $(u\%K^2)\%K$ the column index inside the patch. For example, for `B[10,2]`, $(10\%4)/2 = 1$ and $(10\%4)\%2 = 0$, which means that `B[10,2]` came from row 1 and column 0 of the input patch.

Next, we project these local indexes within a patch onto the input feature map that contains the patch. This can be done by adding these local indexes to the indexes of the beginning pixel (upper left corner) of the patch, i.e., (`v/W_out`, `v%W_out`). Therefore, the row index and column index of the pixel within the selected input feature map are $(u\%K^2)/K + v/W\_out$ and $(u\%K^2)\%K + v\%W\_out$, respectively. In our example, the input feature map pixel for `B[10,2]` has a row index of $1 + 1 = 2$ and column index of $0 + 0 = 0$. We also know that the pixel comes from input feature map 2. Thus, the pixel can be loaded from `X[n,2,2,0]` whose value is 3.

In summary, we have the following mapping from the conceptual `B` matrix to the input feature map array `X` when formulating a batched convolutional layer into matrix-matrix multiplication:

$$B[u, v] \leftarrow X[n, u/(K^2), (u\%K^2)/K + v/W\_out, (u\%K^2)\%K + v\%W\_out] \tag{19.5}$$

where `n` is the sample index in the batch currently being processed, K is the height and width of each filter bank of the convolutional layer, W_out is the width of each output feature map of the convolutional layer, / is integer divide operation, and % is integer remainder operation.

Fig. 19.11 shows a CUDA tiled matrix multiplication kernel that implements a convolutional layer. The kernel is adapted from the tiled matrix multiplication kernel in Fig. 5.9. First, the parameters are adapted to specify the number of input maps (i.e., channels `C`), the number of output maps (`M`), the height of each input map (`H`), the width of each input feature map (`W`), the height/width of each filter (`K`), a pointer to the filter bank array (`F`), a pointer to the input feature map array (`X`), and a pointer to the output feature map array (`Y`) (line 3).

The kernel assumes that it is launched with the input batch size as the `Z` dimension size (number of blocks) of the grid. Thus, each thread uses its `blockIdx.z` as an index to its assigned sample in the input feature map array (line 9). This index will be used when loading the tiles for B (line 28). As was in Fig. 5.9, each thread uses its block and thread indices in the `X` and `Y` dimensions to calculate the row and column indices of its assigned output feature map element (lines 14-15).

The number of phases for the tiled matrix multiplication is the width of the `F` matrix (`C×K×K`) divided by the tile width (`TILE_WIDTH`) (line 19). Loading the `F` matrix tile (line 23) uses the same row and column indices as Fig. 5.9. Note that we did not include bounds checking in this version since we focus on the adaptation of the indices used for loading the `B` matrix tile. We leave the addition of bounds checking as an exercise.

The row and column indices into `B` is the same as those used for matrix `N` in Fig. 5.9. The difference is that we need to map the access to `B` to an access to `X`, which follows Eq. (19.5). Note that we linearize the index to the 4D `N×C×H×W X`

```
1  #define TILE_WIDTH 16
2
3  __global__ void ConvLayer_MM_Kernel(int C, int M, int H,
       int W, int K, float* F, float* X, float* Y) {
4
5    __shared__ float Fds[TILE_WIDTH][TILE_WIDTH];
6    __shared__ float Bds[TILE_WIDTH][TILE_WIDTH];
7
8    int bx = blockIdx.x; int by = blockIdx.y;
9    int bz = blockIdx.z; // bz is used for sample index
10
11   int tx = threadIdx.x; int ty = threadIdx.y;
12
13   // Identify the row/column of the Y element to work on
14   int Row = by * TILE_WIDTH + ty;
15   int Col = bx * TILE_WIDTH + tx;
16
17   // Loop over the F and B tiles for the Y element
18   float Pvalue = 0;
19   for (int ph = 0; ph < (C*K*K)/TILE_WIDTH; ++ph) {
20     // C*K*K is the width of the F matrix
21
22     // Load F and B tiles into shared memory
23     Fds[ty][tx] = F[Row*(C*K*K) + (ph*TILE_WIDTH+tx)];
24
25     int u = ph*TILE_WIDTH + ty;
26     int v = Col;
27
28     Bds[ty][tx]=X[bz*(C*H*W)+(u/(K*K))*(H*W)+((u%(K*K))/K
           + v/(W-K+1))*W+((u%(K*K))%K + (v%(W-K+1)))];
29
30     __syncthreads();
31
32     for (int k = 0; k < TILE_WIDTH; ++k)
33       Pvalue += Fds[ty][k] * Bds[k][tx];
34
35     __syncthreads();
36   }
37   Y[bz*M*(H-K+1)*(W-K+1)+Row*(H-K+1)*(W-K+1)+Col]=Pvalue;
38 }
```

FIGURE 19.11

A tiled matrix multiplication kernel that implements a convolutional layer.

array using row-major layout (line 28). *W_out* is expressed as `W-K+1`, where `W` is the width of the input feature map.

After the thread block completes all phases, each thread writes into its designated `Y` element using the sample index, the row index, and the column index (line 37).

Implementing convolutions with matrix multiplication can be very efficient since matrix multiplication is highly optimized on all hardware platforms. Tiled matrix multiplication is especially fast on GPUs because it has a high ratio of floating-point operations per byte of global memory data access. Recall from Chapter 5 that the number of reuse for each element is proportional to the dimension of the tiles. This implies that matrix multiplication is less efficient on small matrices whose small size may restrict the tile size. Furthermore, the kernel launch overhead tends to be significant for small matrix multiplication. Accordingly, the matrix multiplication approach to convolution is most effective when it creates large matrices for multiplication.

As we mentioned earlier, the filter-bank matrix is an `M × (C*K*K)` matrix and the unfolded input feature map matrix is a `(C*K*K) × (H_out*W_out)` matrix. Note that except for the height of the filter-bank matrix, the sizes of all dimensions depend on products of the parameters to the convolution, not the parameters themselves. While individual parameters can be small, their products tend to be large. For example, it is often true that in early layers of a convolutional network, `C` is small, but `H_out` and `W_out` are large. On the other hand, at the end of the network, `C` is large, but `H_out` and `W_out` are small. Hence, the product `C * H_out * W_out` is usually large for all layers. This means that the sizes of the matrices tend to be consistently large for all layers and thus the utilization of GPU execution resources using this approach tends to be high for all layers in the network, resulting in high execution speed.

19.4 CUDNN library

CUDNN is a library of optimized routines for implementing deep learning primitives. It was designed to make it much easier for deep learning frameworks to take advantage of GPUs. It provides a flexible and easy-to-use C-language deep learning API that integrates neatly into existing deep learning frameworks (e.g., PyTorch, Caffe, TensorFlow, MXNet, ...). The library requires that input and output data be resident in the GPU device memory as we discussed earlier in this chapter. This requirement is analogous to that of cuBLAS.

The library is thread-safe in that its routines can be called from different host threads. Let us consider CNNs to explain the capabilities provided by CUDNN. Convolutional routines for the forward and backward paths use a common descriptor that encapsulates the attributes of the layer. Tensors and filters are accessed through opaque descriptors, with the flexibility to specify the tensor layout using arbitrary strides along each dimension. The most important computational primitive in convolutional neural networks is a special form of batched convolution. In this section, we describe the forward form of this convolution. The CUDNN parameters governing this convolution are listed Table 19.1.

There are two inputs to the convolution:

- `D` is a four-dimensional `N * C * H * W` tensor, which contains the input data,[4]

[4] Tensor is a mathematical term for arrays that have more than two dimensions. In mathematics, matrices have only two dimensions. Arrays with three or more dimensions are called tensors. For the purpose of this book, one can simply treat a T-dimensional tensor as a T-dimensional array.

Table 19.1 Convolution parameters for CUDNN. Note that the CUDNN naming convention is slightly different than what we have been using in previous sections.

Parameter	Meaning
N	Number of images in mini-batch
C	Number of input feature maps
H	Height of input image
W	Width of input image
K	Number of output feature maps
R	Height of filter
S	Width of filter
u	Vertical stride
v	Horizontal stride
pad_h	Height of zero-padding
pad_w	Width of zero-padding

- `F` is a four-dimensional `K * C * R * S` tensor, which contains the convolutional filters.

The input data array (tensor) `D` ranges over `N` samples in a mini-batch, `C` input feature maps per sample, `H` rows per input feature map, and `W` columns per input feature map. The filters range over `K` output feature maps, `C` input feature maps, `R` rows per filter bank, and `S` columns per filter bank. The output is also a four-dimensional tensor `O` that ranges over `N` samples in the mini-batch, `K` output feature maps, `P` rows per output feature map, and `Q` columns per output feature map, where `P = f(H; R; u; pad_h)` and `Q = f(W; S; v; pad_w)`, meaning that the height and width of the output feature maps depend on the input feature map and filter bank height and width, along with padding and striding choices. The striding parameters `u` and `v` allow the user to reduce the computational load by computing only a subset of the output pixels. The padding parameters allow users to specify how many rows or columns of 0 entries are appended to each feature map for improved memory alignment and/or vectorized execution.

CUDNN [9] supports multiple algorithms for implementing a convolutional layer: matrix-multiplication-based (GEMM [10] and Winograd [11]), FFT-based [12], etc. The GEMM-based algorithm to implement the convolutions with a matrix multiplication is similar to the approach presented in Section 19.3. As we discussed at the end of Section 19.3, materializing the expanded input feature matrix in global memory can be costly in both the global memory space and bandwidth consumption. CUDNN avoids this problem by lazily generating and loading the expanded input feature map matrix into on-chip memory only (in a similar way to the implicit approach presented in the previous section), rather than by gathering it in off-chip memory before calling a matrix multiplication routine. NVIDIA provides a matrix-multiplication-based routine that achieves a high utilization of the maximal

theoretical floating-point throughput on GPUs. The algorithm for this routine is similar to the algorithm described in [10]. Fixed sized sub-matrices of the input matrices A and B are successively read into on-chip memory and are then used to compute a sub-matrix of the output matrix C. All indexing complexities imposed by the convolution are handled in the management of tiles in this routine. We compute on tiles of A and B while fetching the next tiles of A and B from off-chip memory into on-chip caches and other memories. This technique hides the memory latency associated with the data transfer, allowing the matrix multiplication computation to be limited only by the time it takes to perform the arithmetic calculations.

Since the tiling required for the matrix multiplication routine is independent of any parameters from the convolution, the mapping between the tile boundaries of the expanded input feature map matrix and the convolution problem is non-trivial. Accordingly, the CUDNN approach entails computing this mapping and using it to load the correct elements of A and B into on-chip memories. This happens dynamically as the computation proceeds, which allows the CUDNN convolution implementation to exploit optimized infrastructure for matrix multiplication. It requires additional indexing arithmetic compared to a matrix multiplication, but fully leverages the computational engine of matrix multiplication to perform the work. After the computation is complete, CUDNN performs the required tensor transposition to store the result in the user's desired data layout.

19.5 Summary

This chapter starts with a brief introduction to machine learning and convolutional neural networks. Then, we present a basic convolutional neural network and the implementation of its major types of layers. We then build on the convolution pattern in Chapter 7 to present a CUDA kernel implementation of the convolutional layer, the most computationally intensive layer of convolutional neural networks.

We then present techniques for formulating convolutional layers as matrix multiplications by implicitly unrolling the input feature maps into a matrix. The conversion allows the convolutional layers to benefit from highly optimized GEMM libraries for GPUs. We also present the CUDA implementation of a matrix multiplication kernel that performs implicit unrolling of the input feature map.

We end the chapter with an overview of the CUDNN library, which is used by most deep learning frameworks. Users of these frameworks can benefit from the highly optimized layer implementations without writing CUDA kernels themselves.

19.6 Exercises

1. Implement the forward pass for the subsampling layer that is introduced in the Section 19.1. An explanation about this type of layer can be found in Appendix B.

2. We used an `[N * C * H * W]` layout for input and output features. Can we reduce the memory bandwidth by changing it to an `[N * H * W * C]` layout? What are potential benefits of using a `[C * H * W * N]` layout?
3. Implement the backward pass for the convolutional layer introduced in the Section 19.1, and further described in Appendix B.
4. Analyze the memory access patterns in the matrix multiplication kernel of Fig. 19.11 and discuss whether the memory accesses to global memory are coalesced and whether the shared memory accesses may suffer from bank conflicts.

References

[1] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[2] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.

[3] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, L. Jackel, Handwritten digit recognition with a back-propagation network, Advances in Neural Information Processing Systems 2 (1989).

[4] G.E. Hinton, S. Osindero, Y.-W. Teh, A fast learning algorithm for deep belief nets, Neural Computation 18 (7) (2006) 1527–1554.

[5] R. Raina, A. Madhavan, A.Y. Ng, Large-scale deep unsupervised learning using graphics processors, in: Proceedings of the 26th Annual International Conference on Machine Learning, 2009, pp. 873–880.

[6] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, Advances in Neural Information Processing Systems 25 (2012).

[7] A. Krizhevsky, cuda-convnet, https://code.google.com/p/cuda-convnet/, 2014.

[8] K. Chellapilla, S. Puri, P. Simard, High performance convolutional neural networks for document processing, in: Tenth International Workshop on Frontiers in Handwriting Recognition, Suvisoft, 2006.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, cuDNN: efficient primitives for deep learning, arXiv preprint, arXiv:1410.0759, 2014.

[10] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, N. Sun, Fast implementation of DGEMM on Fermi GPU, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–11.

[11] A. Lavin, S. Gray, Fast algorithms for convolutional neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 4013–4021.

[12] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, Y. LeCun, Fast convolutional nets with fbfft: a GPU performance evaluation, arXiv preprint, arXiv:1412.7580, 2014.

This page intentionally left blank

CHAPTER

Large language models 20

With special contributions from Juan Gómez-Luna

CONTENTS

20.1 Transformer architecture 478
20.2 Multi-head attention 482
20.3 Implementing attention in CUDA 486
20.4 KV caching 488
20.5 Flash attention 492
20.6 KV cache arithmetic intensity and memory requirement 504
20.7 Alleviating the memory requirements of the attention mechanism 508
20.8 Summary 510
20.9 Exercises 511
References 511

Before Google Researchers introduced Transformers [1] in 2017, language tasks had been predominantly based on statistical language modeling techniques pioneered by IBM (e.g., [2]). In 2018, Google introduced the BERT language model based on Transformers and achieved unprecedented results in natural language tasks. Since then, the Transformer architecture[1] has become the backbone of the deep learning approaches to natural language processing (NLP) based on neural networks.

Since then, LLMs have captured the attention of the industry and advanced rapidly, demonstrating human-level performance in diverse tasks such as text generation, summarization, translation, and conversation. These language models have grown by orders of magnitude in size compared to BERT to accommodate more training data. The term Large Language Models (LLMs) has been coined to distinguish them from previous generations of language models. While LLMs were originally developed to process and generate information in text form, *multi-modal models* (i.e., trained with and used for different types of visual, audio, and other types of data such as scientific data) started to appear.

The core idea of LLMs is to predict the next *token* (e.g., a word, sub-word, character) given a sequence of previous tokens. In chatbot applications such as OpenAI

[1] In this chapter, the term "architecture" refers to "neural network architecture", not "hardware architecture" as we do through the rest of the book.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00031-7
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

ChatGPT [3], the sequence of previous tokens, collectively called the *context*, is composed of the system prompt, user input, and the tokens already generated by the model. In order for LLMs to generate accurate results,[2] they contain billions of parameters (i.e., tens of layers with hundreds of millions of parameters each) that must be trained on huge datasets. During inference, large contexts help to provide more accurate responses. A mechanism, referred to as *attention* is needed to maintain the context and to identify the parts of the large context that are the most relevant to the generation of the next token. It is therefore important to understand the attention mechanism as well as the Transformer Architecture [1] which is the most popular approach to incorporating attention mechanisms into LLMs.[3]

20.1 Transformer architecture

Most LLMs today employ the *transformer* architecture and its *attention* mechanism to capture dependencies between tokens and their relative importance in generating the next token. The attention mechanism resembles a classification process within a neural network. It uses the *softmax* function to generate probabilities. The probability assigned to each token in the sequence of previous tokens serves as an indication of the relative importance of that token to the generation of the next token.

The transformer architecture was originally proposed for transduction tasks, such as machine translation. As such, it was designed with two types of layers. The *encoder* layers contextualize (or mix) the input tokens by generating a sequence of continuous representations, i.e., floating-point *embedding vectors*, from a sequence of symbols (e.g., tokens) in the source language. The *decoder* layers take the sequence of continuous representations from the encoder and generate output tokens (one at a time) in the target language. The decoder does not only take the output of the encoder as input but also the most recent output tokens generated by itself. For this reason, the transformer models are *auto-regressive* models.

Since its first introduction, transformers have been adapted to other types of tasks. Depending on the specific task being performed, a transformer may not need both types of layers. For example, models that perform *discriminative* tasks such as classification do not need a decoder because they do not generate text in a target language. On the other hand, *decoder-only* models can perform *generative* tasks, such as text generation. This is the case of LLMs, where text is generated as a response to a user's query but not as a sequence-to-sequence transformation. Such generative tasks

[2] For some LLM tasks (e.g., text summarization), *accuracy* may not be the right metric, since many different answers can be correct. However, in this chapter, we use "accurate" with the more general meaning of "correct", "valid", or "useful".

[3] Although we focus on Transformers and the attention mechanism in this chapter, the research community keeps actively exploring other approaches to LLMs that provide different advantages. Notable examples are *state-space models* (e.g., Mamba [4]), convolutional models (e.g., Hyena [5]), and hybrid models. They may provide certain advantages over transformers, e.g., less memory requirements.

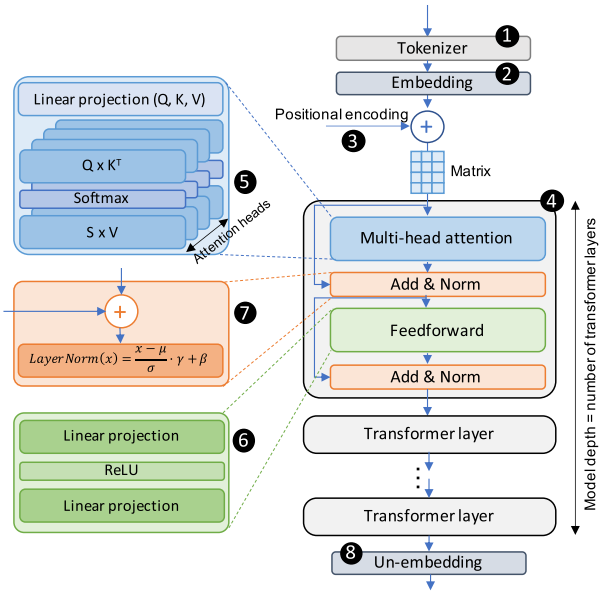


FIGURE 20.1

Decoder-only transformer architecture. This is a baseline architecture without KV caching.

execute the model multiple times. Each time, the model predicts the likelihood of candidates for a new output token following a sequence of previous tokens. In this section, we focus on the architecture of decoder-only transformers. For more details on the encoder-decoder transformer architecture, we refer the reader to [1].

A sample LLM design based on the decoder-only transformer architecture is shown in Fig. 20.1. We briefly describe its main components as follows. The **Tokenizer ❶** converts the text for the input document or query into a sequence of *tokens*, i.e., integer IDs that represent a character or a short sequence of characters. The mapping of text into tokens is based on the concept that a finite vocabulary can be learned from any given set of training documents in the target language. The collection of unique tokens thus learned and recognized by the tokenizer in an LLM form the vocabulary of the LLM. Each tokenizer has its own definition of tokens and approach to translating text input into tokens.

When an LLM is trained, unique IDs are assigned to the tokens as they are encountered in the training document. That is, each time the tokenizer decides to form a token from the next text fragment, it will check if the same text fragment was previously encountered. If not, a new text fragment has been discovered and a new ID number is assigned to this text segment and will be used as the token representation of the text fragment. A mapping from the text fragment to its assigned token ID is recorded for future lookup. If the text fragment was encountered earlier in the training process, the previously assigned token ID is found through the mapping and will

be used to represent the text fragment. The mapping from text fragments to token IDs is part of the outcome of training.

During inference, each time the tokenizer decides to form a token from a text fragment, the mapping is used to look up the token ID assigned to the text fragment. This token ID becomes the next output of the tokenizer. Strictly speaking, tokens are the text fragments that form the vocabulary of an LLM whereas token IDs are the integer representation of the tokens inside the LLM. When we describe the operations of an LLM, we will use the terms token and token ID interchangeably unless there is a need to be precise.

The **Embedding** layer ❷ implements a function that converts tokens, i.e., token IDs generated by the tokenizer, into *embeddings*, i.e., their floating-point vector representation. We will use the terms "embedding" and "embedding vector" interchangeably. This is done via a lookup into an embedding table. The embedding value for each token is also learned from the documents during training. The reason for this conversion is that LLMs are machine learning models that are trained using numerical methods. The training process typically employs gradient-descent approaches that require all functions of the model to be differentiable and all input and parameter values to be in continuous form which is approximated by the floating-point representation.

The **Positional Encoding** layer ❸ adds sequence order information to the input embedding vectors of the input tokens. Positional encoding informs the attention mechanism of the physical position and proximity of tokens in the input, a factor that is considered when determining the relevance of previous input and output tokens to the generation of the next output token.

The embeddings of the entire sequence of tokens of the current conversation between the user and the system (user input, system added input, and output) are accumulated into a matrix according to their order of generation. Each row of the matrix is the embeddings of one token. This matrix is the input of the first transformer layer.

The **Transformer** layers ❹ transform the embeddings to extract dependencies among tokens. Each transformer layer contains a **multi-head attention** sub-layer, an **addition and normalization** sub-layer, a **feed-forward** sub-layer, and another **addition and normalization** sub-layer. The multi-head attention sub-layer ❺ consists of a parallel array of *attention heads*, each of which performs a linear projection of the embeddings using matrix multiplication. The matrices used in the linear projection are learned in the training process and effectively perform a coordinate transformation on the embedding vectors to make subsequent similarity comparisons more effective. The projections then go through another matrix multiplication that measures the similarity between a token and all the tokens that were generated before it.

The outputs from the multiple attention heads are concatenated together and then usually linearly transformed to produce the combined output of the multi-head attention sub-layer.

An **Addition and Normalization** (Add & Norm) sub-layer ❼ follows the multi-head attention sub-layer. The addition operator after each feed-forward sub-layer or

attention sub-layer enforces a *residual connection* to help retain information from the original input. The normalization operator stabilizes the process and improves convergence by normalizing the resulting vectors to fit the desired numerical range of input values to the next layer. Its formula takes the mean (μ) and the standard deviation (σ) of its input embedding vector elements and learned parameters (γ, β).

The **Feed-forward** sub-layer ❻ typically consists of two linear transformations with a ReLU activation in between.[4] Intuitively, the attention layers identify similarities/dependencies between tokens and produce new versions of the token embeddings based on similarity-weighted linear combination of token embeddings generated before them. The feed-forward sub-layers add semantic meaning to these new versions of the token embeddings.

Another Addition and Normalization sub-layer follows the feed-forward sub-layer to help retain the information from the original input to the feed-forward sub-layer and improve convergence while finalizing the output of the transformer layer.

The number of transformer layers is referred to as the *model depth* and varies across LLMs. The attention heads, which perform large-scale matrix multiplications, are the most computationally expensive components of the transformer architecture and will be the focus of the remaining parts of this chapter.

The **Un-embedding layer** ❽ converts the embeddings from the last transformer layer into probability distributions among tokens, which is used to generate the next output token.

Model dimensions are critical for generating accurate answers. First, model depth specifies the number of transformer layers in the model. Deeper models provide levels of classifiers and thus can capture more general patterns of dependencies as explained in Appendix B. The second dimension of the model is the number of elements in each embedding vector, called *head dimension*. Wide embedding vectors, i.e., large head dimensions, capture more semantic information, disambiguate across different concepts, and identify more meaningful dependencies. The third model dimension is the number of *heads* in each attention sub-layer. These heads capture different types of dependencies between tokens. The size of the model in terms of the number of parameters is the product of the three model dimensions and can be very large, reaching billions or even trillions of parameters (i.e., model weights) in practice.

While LLMs require long context for accurate responses, the user inputs are typically too terse and implicit to provide adequate contextual information. As a remedy, the user input is often extended with additional contextual information using a technique called *retrieval-augmented generation (RAG)*. RAG enhances the generation of responses by retrieving relevant documents that provide more specific and up-to-date contextual information and concatenate them with the user input to form a much larger, richer context for the LLMs.

[4] See Appendix B for the definition of ReLU and activation.

During LLM training, the most relevant computation is large-scale matrix-matrix multiplication, which is performed by the attention heads and feed-forward sub-layers and can take advantage of the advanced optimization strategies covered in Chapter 15. These matrix multiplications make the attention heads the most computationally expensive computational components of LLM models. We will elaborate on the design of attention heads and the nature of the matrix multiplications that they perform in Section 20.2.

20.2 Multi-head attention

Each attention sub-layer contains multiple heads which collectively perform multiple classification tasks simultaneously and capture different types of dependencies such as positional (e.g., distance between tokens), semantic (e.g., synonyms or topics), contextual (e.g., connecting articles to names), and syntactic (e.g., connecting verb and object) dependencies. This specialization of attention heads happens during training. The weight matrices of different heads are initialized differently. Then, the gradient descent during back propagation avoids overlap between different heads and prevents overfitting by amplifying gradient variance with techniques such as dropout (i.e., randomly zeroing outputs/probabilities) and layer normalization (i.e., normalizing activations).

Attention heads are independent of each other, so they can be easily concurrently computed over multiple thread blocks. They are identical in their design and thus we show the design of one head in this chapter. The computation of an attention head is illustrated in Fig. 20.2. It starts with input matrix X, whose rows correspond to the embedding vectors for the sequence of tokens generated by the LLM so far. Recall that this includes all the tokens generated from the system prompt, the user input, any system augmentation to the user input, and the output tokens. For the attention sub-layer of the first transformer layer, each row of X contains an embedding generated by the Position Encoding component. For the attention sub-layer of a subsequent transformer layer, the input X is simply the output of the preceding transformer layer.

The number of tokens generated so far, and thus the number of rows in matrix X, is referred to as the *sequence length* or *context length* and is represented by the symbol `N` for the remainder of this chapter. For example, in chatbot applications, `N` grows as the conversation between the user and the system progresses. Models that support large `N` provide token generation with longer contexts and are more reliable because the additional context information can be used to disambiguate terms and phrases, identify contextual relevance, maintain consistency across successive responses, etc. The *attention sub-layers* typically enforce a *causality policy* that the generation of the current token depends only on the tokens that were generated before, and ignores future tokens.

The number of elements in each embedding vector, i.e., the head dimension, is represented by the symbol `d`. Thus, X is an `N`×`d` matrix. The head dimension and the number of heads in each sub-layer are often referred to as the *hidden dimensions* of

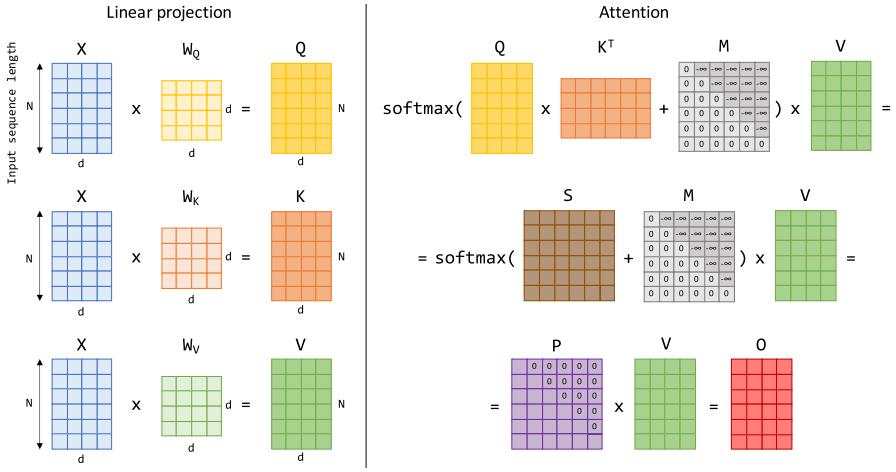


FIGURE 20.2

Matrices (2D tensors) involved in the computation of an attention head. For simplicity, a scaling factor $\frac{1}{\sqrt{d}}$ that is multiplied to all elements of $S = QK^T$ is omitted from this figure and those derived from this figure. Recall that d is the head dimension or the number of elements in each embedding vector. This scaling factor is applied to the elements of S to prevent the matrix multiplication results from becoming too large and causing converging issues for training processes based on gradient-descent methods. We will include the scaling factor in the CUDA implementations later in this chapter.

an LLM as they are internal design parameters and are typically not exposed to the users of the LLM.

The first computational step of an attention sub-layer is to calculate three matrices called the *query* (Q), *key* (K), and *value* (V) matrices. These matrices are obtained with linear projection or coordinate transformation of the row vectors of X, i.e., multiplying the input matrix X by three weight matrices, which we call W_Q, W_K, and W_V, as shown on the left side of Fig. 20.2. These `d`×`d` coordinate transformation matrices are obtained from the training process.[5] Mathematically, each element in a row of the Q, K, and V matrices is a linear combination of the `d` elements in the same row in X. The difference between the transformations performed on Q, K, and V is defined by the difference in the weight values of W_Q, W_K, and W_V.

The second step of an attention sub-layer is specified by Eq. (20.1) and illustrated on the right side of Fig. 20.2.

$$O = softmax(\frac{QK^T}{\sqrt{d}} + M)V \tag{20.1}$$

[5] For simplicity, in this chapter, we assume that the weight matrices are square. In the general case, W_Q, W_K, and W_V are, respectively, of sizes $d_x \times d_q$, $d_x \times d_k$, and $d_x \times d_v$. However, as per QK^T, d_q is always equal to d_k.

Since Q is an $N \times d$ matrix and K^T (transposed matrix K) is a $d \times N$ matrix, the matrix multiplication QK^T generates an $N \times N$ matrix, which we call S in Fig. 20.2. Element (i, j) of QK^T is an inner product between a linearly projected embedding vector for token i (i.e., row i of matrix Q) and that for token j (i.e., column j of matrix K^T or row j of K). Mathematically, an inner product calculates the cosine similarity between its two input vectors assuming that both vectors are of unit length, which can be enforced with the normalization mechanism. Recall that the more similar the two vectors, i.e., the smaller the angle between them, the larger the value of the cosine similarity. Therefore, the $N \times N$ product matrix is a map of similarity between embedding vectors.

The scaling factor $\frac{1}{\sqrt{d}}$ is applied to all elements of $S = QK^T$ to prevent the dot products of the matrix multiplication from growing too large for large head dimensions. Excessively large S elements may cause the `softmax` function to saturate and produce tiny (vanishing) gradients during training [1]. We omit the scaling factor from Fig. 20.2 as well as its associated discussions for simplicity. As we show later in this chapter, it is straightforward to incorporate the scaling factor into the CUDA implementation of the `softmax` function.

The mask matrix `M` adds $-\infty$ to all elements of $\frac{QK^T}{\sqrt{d}}$ whose column indices are greater than their row indices. This transformation enforces the causality policy that the generation of a token should not be influenced by the tokens that are generated after the token.

The `softmax` function in Eq. (20.1) translates each row of $QK^T + M$ into a probability distribution. Let us denote element (r, c) at column c in row r of the $QK^T + M$ matrix as *logit* $l_{r,c}$ that the `softmax` function takes as inputs to generate probabilities $p_{r,c}$, as Eq. (20.2) shows:

$$p_{r,c} = \frac{e^{l_{r,c}}}{\sum_{j=1}^{N} e^{l_{r,j}}} = \frac{e^{l_{r,c}-m_r}}{\sum_{j=1}^{N} e^{l_{r,j}-m_r}} \tag{20.2}$$

where m_r is the maximum element value of row r. Note that $l_{r,c}$ can be large, so $e^{l_{r,c}}$ can overflow. For this reason, we subtract $m_r = \max_{j=1}^{N}(l_{r,j})$, from $l_{r,c}$ before applying the exponential function, which prevents the exponential function from overflowing and thus maintains numerical stability. Because the subtraction of m_r from $l_{r,c}$ is equivalent to multiplying $e^{l_{r,c}}$ by e^{-m_r} and both the numerator and the denominator are multiplied by e^{-m_r}, the resulting p elements are mathematically equivalent.

With the causality mask matrix, any logit element whose row index r is smaller than the column index c becomes $-\infty$. This makes $e^{l_{r,c}-m_r} = e^{-\infty-m_r} = e^{-\infty} = 0$. That is, each row of the output matrix of $P = softmax(QK^T + M)$ in Fig. 20.2, is a probability distribution based on similarity between token r's embedding vector in Q and those for the tokens that were generated before token r in K. All remaining elements of the row are 0s after `softmax`. That is, the $N \times N$ matrix generated by $softmax(QK^T + M)$ is a lower-triangular matrix where all elements with indices (r, c) are zeros if $r < c$.

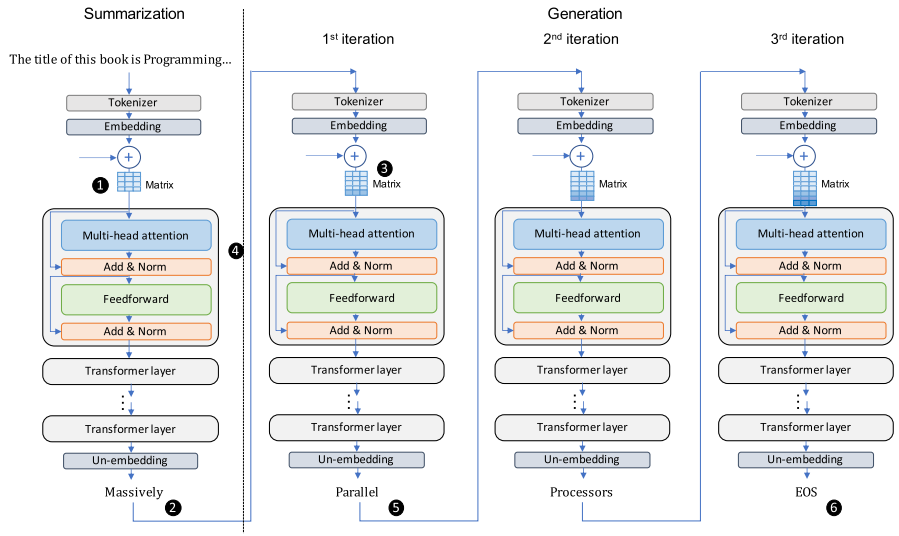


FIGURE 20.3

LLM stages: summarization and generation.

The final step of the attention sub-layer is the matrix multiplication of the $N \times N$ probability matrix P and the V matrix. For each token, this matrix multiplication generates a new embedding vector where each element is a similarity/probability weighted sum of the same element of all embeddings in the V matrix. That is, the new embedding vector for each token in the output matrix O is a similarity-weighted sum of the embedding vectors of all previously generated tokens. The output O matrix then goes through the addition-and-normalization and feed-forward sub-layers to become the input matrix X for the next attention sub-layer.

Inferencing with an LLM based on the decoder-only architecture in Fig. 20.1 starts with a *summarization phase* where the system prompt tokens, the user input tokens, and the system-augmented (e.g., RAG) tokens are concatenated to form the X input matrix of the first transformer layer. During the summarization phase, the matrices involved and the computation for each attention head are as shown in Fig. 20.2. The accumulated X matrix is fed to the transformer layers. Each transformer layer generates the input for the next transformer layer. Upon completion of the last transformer layer, the un-embedding layer is used to generate an output token as a text fragment. The next inference iteration starts the *autoregression phase*, which is also called the *decoding phase* or the *generation phase*. The generated output text fragment is fed to the tokenizer, appending the generated embedding vector to the X matrix of the first transformer layer, and activating the transformer layers to generate the next text fragment.

Fig. 20.3 illustrates the two phases of LLM inference. During the summarization phase, the initial sequence of the system's prompt as well as the initial user's prompt

is processed by the tokenizer and embedding layers and accumulated into the X matrix ❶. For simplicity, we omit the systems' prompt and the typical preamble of the user input in this toy example. As shown in Fig. 20.3, the user wants the LLM to complete the title of this book by entering a prompt "The title of this book is Programming".

For simplicity, we assume that the LLM in this example uses entire words as tokens. Thus, during the summarization phase, the tokenizer layer, the embedding layer, and the position encoding layer accumulate the embedding vectors for the seven tokens generated from the user input into the X matrix of the first transformer layer. The LLM then invokes the first transformer layer whose completion activates the second transformer layer, etc. When activated, each transformer layer calculates its K, V, Q, QK^T, and O before activating the next transformer layer. After all transformer layers and the final un-embedding layer complete their calculation, the LLM produces the first output token ("Massively" in our example ❷).

With the generation of the first output token, the LLM enters the generation phase. During the generation phase, each output token is generated based on the tokens of the initial prompt and the output tokens of all previous iterations. In our example, the token produced at the end of the summarization phase ("Massively") becomes part of the input to the first iteration of the generation phase, which will produce the next output token. For each new output token, the LLM generates a corresponding new embedding vector (i.e., a new row of matrix X) that will go through the transformer layers ❸.

At the end of the first iteration, we obtain a new output token ("Parallel"), which will be the input of the next iteration of the generation phase ❺. The generation phase continues iterating until an end-of-sequence (EOS) token is generated ❻.

20.3 Implementing attention in CUDA

Attention computation in the transformer architecture shown in Fig. 20.1 consists of two matrix multiplications, one matrix element-wise scaling, one matrix addition, and the `softmax` computation. The matrix multiplications can be performed with either custom CUDA matrix multiplication kernels or calls to the highly optimized GEMM functions in available libraries such as cuBLAS. The scaling factor $\frac{1}{\sqrt{d}}$ can be applied to the QK^T multiplication in the same custom kernel, or as the scaling factor α of cuBLAS's GEMM API. The only new aspect of attention computation is the implementation of the `softmax` function. We will thus focus on the CUDA implementation of the `softmax` function and leave the rest as an exercise.

Fig. 20.4 shows a CUDA kernel that fuses the matrix addition with M and the `softmax` computation within the same kernel. The fused kernel avoids an additional pass through its input S matrix before the `softmax` computation and improves the arithmetic intensity and thus GPU utilization.

The execution configuration of this kernel (not shown in Fig. 20.4) is a 1D grid of thread blocks. The kernel assumes that the number of thread blocks (i.e., `gridDim.x`)

```
1  __global__ void softmax_kernel(float* S, float* D, float* P, int N)
       {
2    typedef cub::BlockReduce<float, BLOCK_SIZE> BlockReduce;
3    __shared__ typename BlockReduce::TempStorage temp_store;
4    __shared__ float max_or_sum;
5    float* S_row = &S[blockIdx.x * N];
6    float max_val_thread = NEG_INFINITY;
7    for(int idx=threadIdx.x; idx<=blockIdx.x; idx+=blockDim.x){
8      if(S_row[idx] > max_val_thread) max_val_thread = S_row[idx];
9    }
10   __syncthreads();
11   float max_val_row = BlockReduce(temp_store).Reduce(max_val_thread
         , cub::Max());
12   if(threadIdx.x == 0) max_or_sum = max_val_row;
13   __syncthreads();
14   max_val_row = max_or_sum;
15   float sum_thread = 0.f;
16   for(int idx=threadIdx.x; idx<=blockIdx.x; idx+=blockDim.x){
17     sum_thread += exp(S_row[idx] - max_val_row);
18   }
19   __syncthreads();
20   float sum_row = BlockReduce(temp_store).Reduce(sum_thread, cub::
         Sum());
21   if(threadIdx.x == 0) max_or_sum = sum_row;
22   __syncthreads();
23   sum_row = max_or_sum;
24   for(int idx=threadIdx.x; idx<N; idx+=blockDim.x){
25     P[blockIdx.x * N + idx] = idx <= blockIdx.x ? exp(S_row[idx] -
           max_val_row) / sum_row : 0;
26   }
27   if(threadIdx.x == 0) D[blockIdx.x] = sum_row;
28 }
```

FIGURE 20.4

A CUDA kernel implementation for the softmax function.

is set to the number of rows in matrix `S`, i.e., the sequence length `N`. Both `gridDim.y` and `gridDim.z` are 0. This assumption limits the kernel to a context length that is less than or equal to the maximal number of thread blocks allowed in the x dimension. If a larger context length is needed, one can make the code more general by adding an outer loop to the kernel code so that the threads in the grid iterate until all rows of S are processed.

Each thread block is assumed to be 1D with `BLOCK_SIZE` threads. `BLOCK_SIZE` is a compile-time constant that is set by the programmer. Since all threads in the block collaboratively process a row of S, one should choose a `BLOCK_SIZE` value that minimizes the number of expected iterations for a thread block to process its assigned row while maximizing the occupancy of the SMs in the GPU.

With the scaling factor applied as part of the matrix multiplication QK^T and the causality fused on the `softmax`, each S element of a row can be viewed as $l_{r,c}$ in Eq. (20.2). Lines 6-14 in Fig. 20.4 obtain the maximum element value in the row assigned to the thread block, i.e., m_r in Eq. (20.2), and deposit it into local variable

max_val_row. In each iteration of the for loop in lines 7-8, each thread in the block examines one S element in a section of blockDim.x elements of the assigned row and update a thread-private variable max_val_thread. Note that the exit condition of the for loop in 7-8 is idx <= blockIdx.x. This way, the code ignores row elements whose row index (blockIdx.x) is less than their column index (idx), which implements the causality policy without explicitly adding matrix M in Eq. (20.1), i.e., adding $-\infty$ to all elements of S with column indices greater than their row indices.

After the threads exit the for loop in lines 7-8, each thread holds a thread-private maximal value among the S elements examined by the thread. In line 10, all threads of the block synchronize to ensure that all completed examining their assigned S elements. In line 11, all threads in the block collaborate to obtain the maximum of the entire row by calling the BlockReduce function of the CUB library. The total reduction result is obtained by the thread with threadIdx.x == 0. Thus, this thread needs to share the result with all other threads via the shared memory variable max_or_sum (line 12). After synchronizing (line 13), all threads read max_or_sum (line 14). Note that the code in lines 7-14 is essentially a reduction with thread coarsening, which we already introduced in Chapter 10.

Next, lines 15-23 obtain the denominator of Eq. (20.2). In lines 16-17, the threads calculate a thread-private sum. The exit condition of the for loop implements the causality policy. After synchronizing (line 19), CUB's BlockReduce function computes the total sum for the row. In lines 21-23, thread 0 broadcasts the final sum result via max_or_sum in shared memory. As we have mentioned above for the max reduction, lines 15-23 implement a sum reduction with coarsening, similar to the one covered in Chapter 10.

Finally, lines 24-25 calculate the softmax output for each element of S and store them in memory (in output matrix P). Note that this for loop iterates over all elements of a row, and those elements with column index greater than row index are set to 0, per the causality policy. The denominator is also stored in output vector D (line 27) for reuse during training.

In summary, the softmax kernel in Fig. 20.4 implements the causality policy and the softmax function within one kernel. No global barriers are needed (i.e., no additional kernel for adding M); only the block-level __syncthreads() barrier is used. The reader is encouraged to complete the implementation of the attention sub-layer by adding the matrix multiplication for generating S and the matrix multiplication for generating O.

20.4 KV caching

The transformer architecture shown in Fig. 20.1 has an obvious inefficiency, if it is implemented in the straightforward way that Fig. 20.3 illustrates. In each iteration, five matrix multiplications are performed in each transformer layer. Since the sequence length N can grow to the level of tens of thousands of tokens or more, these matrix multiplications can be very expensive. Since there is only one additional row

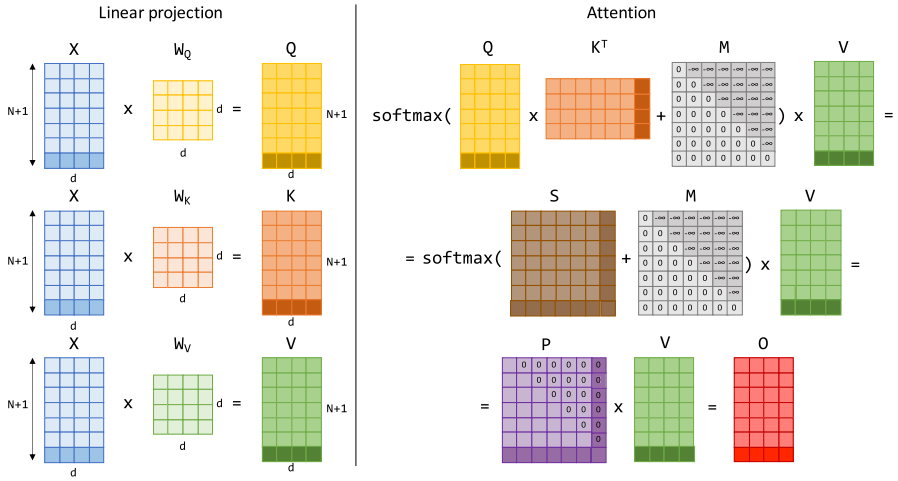


FIGURE 20.5

Optimization opportunities for reducing the matrix multiplication work from one LLM iteration to the next.

in the input matrix X from one iteration of the decoding phase to the next, one should ask if we really need to perform full matrix multiplications in every iteration. The answer is no.

Fig. 20.5 highlights the new parts of each matrix generated from iteration i ($N = i$) to iteration $i+1$ ($N = i+1$). For input matrix X, one new row, i.e., the embedding vector of the new token, is attached to the bottom, which increases the dimensions of X from $N \times d$ to $(N+1) \times d$. The effect of this new row of X on Q, K, and V is straightforward. Each of them becomes an $(N+1) \times d$ matrix. The new row in the bottom of each matrix is generated by performing a vector-matrix multiplication between the new bottom row of X and W_Q, W_K and W_V, respectively.

The step of generating QK^T is a bit more complicated. The new product is an $(N+1) \times (N+1)$ matrix. The contents of the matrix are largely the same as the previous iteration, except for the $(N+1)^{th}$ row and the $(N+1)^{th}$ column (highlighted with darker color in Fig. 20.5). For example, element (0, 0) of the product matrix is generated with an inner product between the 0th row of Q and the 0th row of K (i.e., the 0th column of K^T). Note that both of them are the same as the previous iteration so there is no change in the element (0, 0) of the product matrix. In general, the value of any element (r, c) of QK^T remains the same as the previous iteration as long as both r and c are less than N since the element is generated from rows of Q and K that remain the same as the previous iteration.

The new N^{th} row of QK^T is calculated as a vector-matrix multiplication between the (new) N^{th} row of matrix Q and matrix K^T. The new N^{th} column of QK^T consists of all zero elements because of the zero-masking performed under the causality policy: the tokens generated by the LLM should not be influenced by the token generated after them. The only exception is the Nth element of the Nth column. This element value is not zero and is covered by the generation of the new N^{th} row.

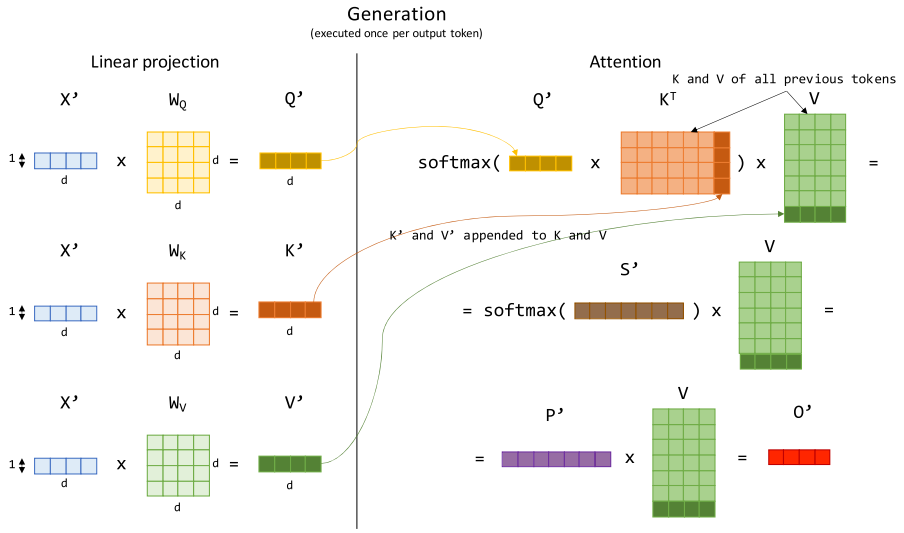


FIGURE 20.6

Vectors and matrices involved in the generation phase of the attention computation with KV caching.

The output of the `softmax` function can also be incrementally updated from the previous iteration. Recall that `softmax` is applied to each row of the QK^T matrix. Since the new N^{th} element of each row (except for the new N^{th} row) is 0, these new elements do not impose any change to the elements in these rows from iteration N to iteration $N+1$ as they do not dilute any of the probabilities from these previous elements.

The reader should verify that all first N (0th through $(N-1)^{th}$) rows of the O matrix in iteration $N+1$ remain the same as the previous iteration. One simply needs to perform a vector-matrix multiplication between the new row of $softmax(QK^T)$ and the new V.

The vectors and matrices involved in generating the new parts of Q, K, V, and O are shown in Fig. 20.6. Note that for each iteration, a multi-head attention sub-layer only needs to receive one new row of X. Therefore, each of them only needs to deliver a new row of its output O to the next layer. Note that the attention computation during the generation phase still requires the entire K and V. This requirement can be met by *memoizing* K and V (i.e., storing and reusing later) in the form of the so-called *KV cache*,[6] which is a common optimization of attention sub-layers.

[6] Note that the term KV in LLM inference should not be confused with the term key-value in the key-value store literature but they are analogous terms. LLM inference produces an output that is a weighted sum of values (V), where the weight is proportional to how close a query (Q) and the keys (K) are [1].

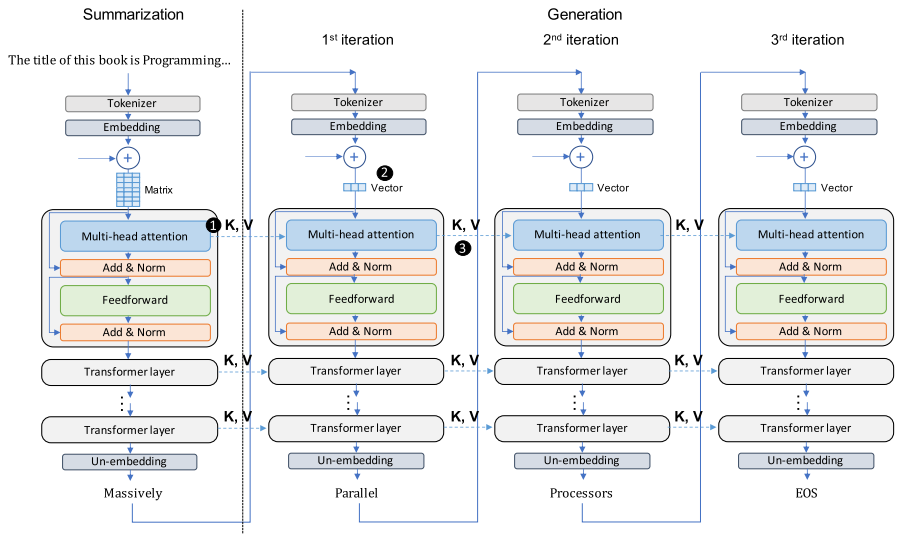


FIGURE 20.7

LLM stages: summarization and generation with KV caching.

Fig. 20.7 revises Fig. 20.3 to illustrate the two phases of LLM inference with KV caching. In order to avoid recomputation in each iteration of the LLM inference, the K and V generated by each transformer layer are entered into the layer's KV cache to be reused for the generation of future tokens.

During the summarization phase, all transformer layers generate their initial K and V matrices and fill their KV caches ❶. With KV caching, the summarization phase is also called the *prefill phase* as the transformer layers prefill their KV caches with the initial contents of their K and V matrices during this phase.

Then, during the generation phase, each output token is generated based on the last output token as well as the K and V matrices of all previous iterations. The last generated text fragment is used to obtain a new embedding vector that is fed to the first transformer layer ❷. In each transformer layer, three vector-matrix multiplications are first performed to generate one new row for Q, K, and V respectively. These new rows are shown as vectors Q', K', and V' in Fig. 20.6. The LLM attaches these new rows to the K and V matrices from the previous iteration in the KV cache ❸ to form the new K and V matrices for the current iteration and use them to calculate `O'` and send it through the rest of the layer to generate the new input vector for the next transformer layer.

It should be clear to the reader that the prefill (summarization) phase still requires one transformer pass for a large number of tokens and invokes several large matrix-matrix multiplications (GEMMs). It is typically compute-bound (high arithmetic intensity) and can achieve high GPU utilization. In Section 20.5, we present *flash attention*, a technique to further improve the speed of the prefill phase by reducing the consumption of global memory bandwidth during the transitions between these matrix multiplications and the `softmax` operation.

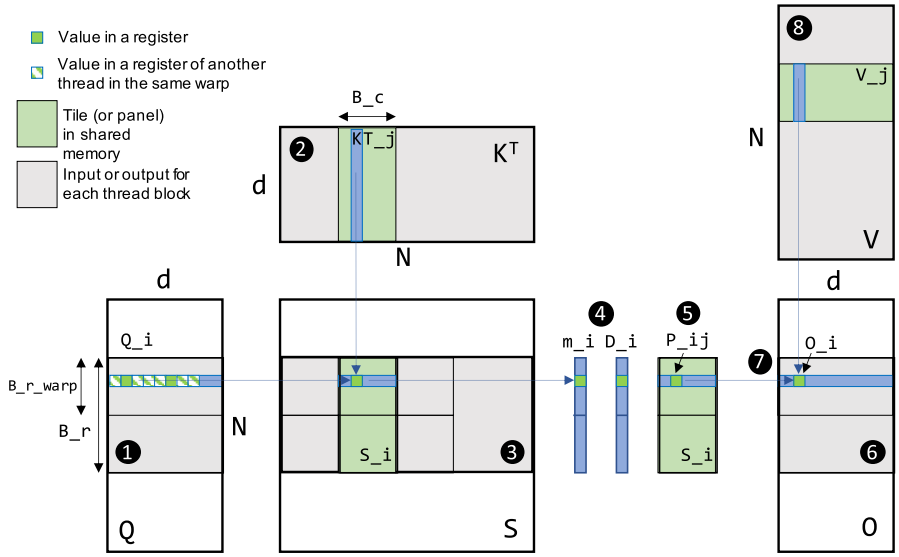


FIGURE 20.8

Tiling approach used in flash attention. Note that we use the same tile in shared memory `S_i` to store the tile of `P`.

In contrast, the generation phase invokes one transformer pass per output token. With KV caching, each pass performs mostly vector-matrix multiplications (GEMVs) and is typically memory bandwidth bound (low arithmetic intensity). Thus, it tends to under-utilize the GPU compute resources. In Section 20.6, we introduce some optimization techniques to increase the arithmetic intensity of the generation phase (namely, *batching* and *speculative decoding*).

20.5 Flash attention

The reader may observe that the implementation of attention with a separate `softmax` kernel, as the one presented in Fig. 20.4, implies global barriers at kernel boundaries as well as several loads/stores of entire matrices from/to the global memory. These global barriers and global memory accesses create a major performance bottleneck that the *flash attention* formulation [6] alleviates.

Flash attention mathematically reformulates the attention heads so that the operations in each attention head can be reordered and fused into a single kernel, and reorganized in a tiled manner. Within the fused kernel, as shown in Fig. 20.8, each thread block takes a horizontal panel (contiguous rows) of Q ❶, the entire K^T ❷, and the entire V ❽ to compute their contribution to a horizontal panel of O ❻. The inputs and output for each thread block are shaded gray in Fig. 20.8.

Furthermore, flash attention enables each thread block to iterate through the K^T and V matrices in a tiled manner while calculating their contribution to the O tile. These tiles are illustrated in Fig. 20.8 and they are small enough to fit into the on-chip memory. In each iteration, each thread block loads a vertical panel (contiguous columns) of K^T ❷ and a horizontal panel of V ❽ to compute their contribution to the O tile. All thread blocks execute in parallel, without any global barrier. This iterative approach allows the kernel to keep all intermediate matrices ❸❺ only in the on-chip memory in tiled form and eliminate the global memory traffic for these intermediate matrices.

Flash attention is an exact reformulation of the attention mechanism in transformers. It is mathematically identical to the original formulation. flash attention is applicable to training (both forward pass and backward pass) and inference (forward pass). In this chapter, we focus on the forward pass. The backward pass is conceptually simpler because it does not require softmax rescaling, but it requires five matrix multiplications while the forward pass requires only two. For details on applying flash attention to the backward pass, we refer the reader to the original flash attention paper [6].

The key idea of flash attention is tiling. However, tiling in attention head implementations is not as straightforward as in matrix multiplication, because the formulation of `softmax` in Eq. (20.2) requires multiple passes through each row of its input. The reason is that the computation (see Eq. (20.2)) depends on m_r, which is the maximum of all values in a row r of the matrix that results from $QK^T + M$, and the sum $D_r = \sum_{j=1}^{N} e^{l_{r,j} - m_r}$, which is used as the denominator when generating p elements in Eq. (20.2). With this formulation, one would need to scan through the row at least once to gather both the max value and the sum before doing another pass to calculate the elements of the row in the output P. We would need a global barrier synchronization to ensure that all elements of the $QK^T + M$ matrix are available before calling the `softmax` kernel.

In CUDA, a global barrier synchronization can be done by terminating a kernel and launching a new one. Alternatively, one could launch a persistent grid and use grid-level cooperative group synchronization API to avoid kernel call overheads. However, good thread grid configurations for matrix multiplication are likely different from those for the `softmax` operation. This is an important reason for launching separate kernels, each with its own grid configuration, for matrix multiplications and the `softmax` in the baseline CUDA implementation of the transformer architecture. However, regardless of which global synchronization strategy we choose, the multiple passes through the rows of S of the `softmax` incurs global memory traffic that can significantly limit the throughput of our implementation.

Fortunately, we can remove the need for multiple passes of `softmax` through entire rows of S by defining a partial series for the denominator, as Eq. (20.3) shows:

$$D_{r,A} = \sum_{j \in A} e^{l_{r,j} - m_{r,A}} \tag{20.3}$$

where A is a subset of column positions in row r. Note that when A encompasses all column positions of row r, $D_{r,A}$ is equal to D_r.

If we can derive a composition rule for merging the partial series for A with the partial series for another subset of column positions B into a partial series for subset $A \cup B$, we have a path to compute the partial series for subsets independently (potentially, in parallel) and simply merge them together as they become available [7]. Once the merged subset contains all column positions of the row r, we have the correct value for D_r.

Eq. (20.4) shows the composition rule for merging $D_{r,A}$ and $D_{r,B}$ into $D_{r,A\cup B}$:

$$\begin{aligned} D_{r,A\cup B} &= \sum_{j\in A\cup B} e^{l_{r,j}-m_{r,A\cup B}} = \sum_{j\in A} e^{l_{r,j}-m_{r,A\cup B}} + \sum_{j\in B} e^{l_{r,j}-m_{r,A\cup B}} = \\ &\left(\sum_{j\in A} e^{l_{r,j}-m_{r,A}}\right) e^{m_{r,A}-m_{r,A\cup B}} + \left(\sum_{j\in B} e^{l_{r,j}-m_{r,B}}\right) e^{m_{r,B}-m_{r,A\cup B}} = \\ &D_{r,A} e^{m_{r,A}-m_{r,A\cup B}} + D_{r,B} e^{m_{r,B}-m_{r,A\cup B}} \end{aligned} \tag{20.4}$$

where $m_{r,A\cup B}$ is the larger one between $m_{r,A}$ and $m_{r,B}$. Note that the second step of the derivation is correct as long as A and B do not have common elements. With this composition rule, we can easily merge two partial series sums as long as we also keep the maximal values with each subset, i.e., $m_{r,A}$ and $m_{r,B}$. In the CUDA implementation of flash attention that we present below, the subsets A and B are tiles of each row of matrix `S`, and are guaranteed to have no common elements.

Flash attention [6] takes this approach one step further: it applies the composition rule to the output matrix O as well. Let's first focus on the computation of an O matrix element $o_{r,c}$, which is an inner product between row r of P and column c of V. We express the contribution of the subset A of row r to $o_{r,c}$ as $o_{r,c,A}$ in Eq. (20.5):

$$o_{r,c,A} = \sum_{j\in A} p_{r,j,A} v_{j,c} = \frac{1}{D_{r,A}} \sum_{j\in A} e^{l_{r,j}-m_{r,A}} v_{j,c} \tag{20.5}$$

where $p_{r,j,A}$ is defined as the intermediate value of the P matrix element $p_{r,j}$ when considering only the subset of elements in A. Similarly, $m_{r,A}$ is defined as the maximal value among the elements in A, not the entire row. The difference between $p_{r,j,A}$ and $p_{r,j}$ is that the denominator and the maximal value are both based on only the elements in A, not all the elements in row r. As more elements are merged into A, it will eventually contain all elements of row r. The following composition rule is derived to guarantee that $p_{r,j,A}$ will become $p_{r,j}$ when A contains all elements of row r.

$$o_{r,c,A\cup B} = \sum_{j\in A\cup B} p_{r,j,A\cup B} v_{j,c} = \sum_{j\in A\cup B} \frac{e^{l_{r,j}-m_{r,A\cup B}}}{D_{r,A\cup B}} v_{j,c}$$

$$
\begin{aligned}
&= \sum_{j\in A} \frac{e^{l_{r,j}-m_{r,A\cup B}}}{D_{r,A\cup B}} v_{j,c} + \sum_{j\in B} \frac{e^{l_{r,j}-m_{r,B}}}{D_{r,A\cup B}} v_{j,c} \\
&= \sum_{j\in A} \frac{e^{l_{r,j}-m_{r,A\cup B}+m_{r,A}-m_{r,A}}}{D_{r,A\cup B}} v_{j,c} + \sum_{j\in B} \frac{e^{l_{r,j}-m_{r,A\cup B}+m_{r,B}-m_{r,B}}}{D_{r,A\cup B}} v_{j,c} \\
&= \sum_{j\in A} \frac{e^{l_{r,j}-m_{r,A\cup B}+m_{r,A}-m_{r,A}} D_{r,A}}{D_{r,A\cup B} D_{r,A}} v_{j,c} + \sum_{j\in B} \frac{e^{l_{r,j}-m_{r,A\cup B}+m_{r,B}-m_{r,B}} D_{r,B}}{D_{r,A\cup B} D_{r,B}} v_{j,c} \\
&= \sum_{j\in A} \frac{e^{l_{r,j}-m_{r,A}} e^{m_{r,A}-m_{r,A\cup B}} D_{r,A}}{D_{r,A} D_{r,A\cup B}} v_{j,c} + \sum_{j\in B} \frac{e^{l_{r,j}-m_{r,B}} e^{m_{r,B}-m_{r,A\cup B}} D_{r,B}}{D_{r,B} D_{r,A\cup B}} v_{j,c} \\
&= \frac{D_{r,A} e^{m_{r,A}-m_{r,A\cup B}}}{D_{r,A\cup B}} \sum_{j\in A} \frac{e^{l_{r,j}-m_{r,A}}}{D_{r,A}} v_{j,c} + \frac{D_{r,B} e^{m_{r,B}-m_{r,A\cup B}}}{D_{r,A\cup B}} \sum_{j\in B} \frac{e^{l_{r,j}-m_{r,B}}}{D_{r,B}} v_{j,c} \\
&= \frac{D_{r,A} e^{m_{r,A}-m_{r,A\cup B}}}{D_{r,A\cup B}} \sum_{j\in A} p_{r,j,A} v_{j,c} + \frac{D_{r,B} e^{m_{r,B}-m_{r,A\cup B}}}{D_{r,A\cup B}} \sum_{j\in B} p_{r,j,B} v_{j,c} \\
&= \frac{D_{r,A} e^{m_{r,A}-m_{r,A\cup B}}}{D_{r,A\cup B}} o_{r,c,A} + \frac{D_{r,B} e^{m_{r,B}-m_{r,A\cup B}}}{D_{r,A\cup B}} o_{r,c,B} \qquad (20.6)
\end{aligned}
$$

where $o_{r,c,A}$ and $o_{r,c,B}$ are versions of the $o_{r,c}$ that account for the contributions of subsets A and B of row r of P. In particular, the D_r and m_r values used to calculate these P element values in these subsets also only account for their respective subset elements. Thus, the composition rule calls for rescaling $o_{r,c,A}$ and $o_{r,c,B}$ before merging them into $o_{r,c,A\cup B}$, as shown in the final step of the derivation in Eq. (20.6). While the composition rule in Eq. (20.6) applies to any subset of a row of P, it is especially useful when the subset is a contiguous segment of P that is generated from tiles of Q and K^T. Fig. 20.8 illustrates the flash attention tiling approach [8] that is enabled by the composition rule.

Fig. 20.9 shows a kernel that implements this tiling approach. The code corresponds to the computation of the forward pass of the attention layer of one head. The entire grid is involved in this computation. In a more complete implementation, multiple heads can be computed in the same kernel by launching additional dimensions of thread blocks in the grid. The same tiling approach also applies to the summarization phase in LLM inference. The code in Fig. 20.9 computes on 32-bit floating point values, but state-of-the-art implementations typically operate on 16-bit floating point (half precision) or even 8-bit floating point values. Such implementations can benefit from the sheer performance of GPU tensor cores.

The flash attention kernel code assumes that the input matrices `Q`, `K`, and `V` are of dimensions `N` × `d`, as shown in Fig. 20.8. The host code (not shown and left as an exercise) allocates global memory space, initializes the contents of these matrices in the global memory and passes them by reference as kernel arguments. The host code also allocates space in the global memory for the output matrix `O` (`out_O` in the code in Fig. 20.9) whose contents will be generated by the execution of the kernel. The

```
#define WARP_SIZE 32
#define BLOCK_SIZE 512
#define N_WARPS BLOCK_SIZE / WARP_SIZE
#define B_r 32
#define B_r_warp B_r / N_WARPS
#define B_c 32
#define d 128

__global__ void flashattention_forward_kernel(const float* Q, const
      float* K, const float* V, const int N, const float scaling,
     float* out_D, float* out_O) {
  const int T_r = N / B_r;
  const int T_c = N / B_c;
  const int d_size = d > WARP_SIZE ? d / WARP_SIZE : 1;
  __shared__ float KT_j[B_c*d+((B_c*d)>>LOG_NUM_BANKS)];
  __shared__ float S_i[B_r][B_c];
  __shared__ float V_j[B_c][d];
  for (int i = blockIdx.x; i < T_r; i+=gridDim.x) {
    float O_i[B_r_warp][d_size];
    float D_i[B_r_warp];
    float m_i[B_r_warp];
    // Initialize O, D, and m
    initialize(O_i, D_i, m_i);
    float Q_i[B_r_warp][d_size];
    // Load Q to registers
    load_Q(Q, Q_i, i);
    for (int j = 0; j < T_c; j++) {
      // Load KT and V to shared memory
      load_KT_and_V(K, V, KT_j, V_j, j);
      __syncthreads();
      for (int ii = 0; ii < B_r_warp; ii++) {
        // Compute S row and maximum
        float curr_max_warp = compute_S_and_max(Q_i, KT_j, S_i, i,
            j, ii, scaling);
        // Update m_i and D_i
        float last_m = update_m_and_D(m_i, D_i, curr_max_warp, ii);

        // Compute P and update denominator
        float curr_sum_warp = compute_P_and_update_D(S_i, m_i, D_i,
            i, j, ii);
        // Compute O
        compute_O(S_i, V_j, O_i, ii, last_m, m_i);
      }
      __syncthreads();
    }
    // Store O and denominator to global memory
    store_O(out_O, out_D, O_i, D_i, i, N);
  }
}
```

FIGURE 20.9

A CUDA kernel implementation of the forward pass of flash attention [8].

kernel generates an `N`-element output vector D (`out_D` in the code). Each element of D contains the final value of the denominator used to calculate the probability values in a row of `P`. This output vector is needed for training, not inference, and is included in the kernel code so that the kernel can be used in training as well.

At the high level, the flash attention kernel is designed to be executed by a 1D grid of thread blocks, i.e., only the x-dimension is used. Each block is responsible for generating a horizontal panel (a tile of a contiguous collection of rows) ❻ of the output matrix `O`. The horizontal dimension of the output tile is `d`, the width of the attention head. The vertical dimension of the output tile, i.e., the number of rows in the horizontal panel, is specified by `B_r` whose value is set to maximize the level of data reuse in the on-chip memory as limited by the amount of on-chip memory available as well as the number of thread blocks needed to fully utilize the GPU. The value of `B_r_warp` is `B_r` divided by the number of warps per thread block.

The outer-most `for` loop in line 16 ensures that all `O` panels are generated. In each iteration, every thread block processes one panel of `Q` and generates one panel of `O`. Thus, all blocks in the grid collectively generate `gridDim.x` panels of `O` in each iteration of the `for` loop. All thread blocks iterate until all the `T_r` = $\frac{N}{B\_r}$ (line 10) panels of `O` are generated.

For each panel of `Q`, its designated thread block generates the corresponding `O` tile in three conceptual steps that the code in Fig. 20.9 shows between lines 25 and 40. First, each thread block performs a matrix multiplication between a `B_r` × `d` horizontal panel of `Q` ❶ and the entire K^T matrix ❷ to generate a `B_r` × `N` horizontal panel of `S` ❸. The code of the device function `compute_S_and_max()` (line 31) is in Fig. 20.12. We will discuss it shortly. In the second step, the `S` panel is translated by softmax into its corresponding `B_r` × `N` horizontal panel of `P`. The code of the device function `compute_P_and_update_D()` (line 35) is in Fig. 20.15. The `P` panel is then multiplied with the entire `V` matrix to generate its `B_r` × `d` output tile of `O`. The code of the device function `compute_O()` (line 37) is in Fig. 20.13.

As we discussed at the beginning of this section, a naive implementation of these three steps could use three kernels. These kernels would write and read `S` and `P` to and from the global memory, resulting in low arithmetic intensity as well as kernel invocation overhead. The flash attention formulation in Eq. (20.6) allows the three conceptual steps to be fused into one kernel and performed in a piece-meal, iterative manner in on-chip memory, which eliminates the need to materialize `S` and `P` in the global memory as well as the corresponding undesirable global memory traffic.

We will apply both shared memory tiling and register tiling to matrix-matrix multiplication, which take advantage of the bandwidth of both the register file and the shared memory. We will choose registers for Q and O tiles and the shared memory for K^T, S/P, and V tiles. Line 14 of the code declares `S_i` in the shared memory to hold both the input `S` and output `P` tiles of softmax. Lines 13 and 15 allocate space in shared memory for panels of the input matrices, `KT_j` and `V_j`, as illustrated in Fig. 20.8.

In lines 17-21, each thread allocates registers for values of the row maximum (i.e., `m_i` ❹ in Fig. 20.8), the denominator of the softmax (i.e., `D_i` ❹ in Fig. 20.8), and the output (`O_i`), and the device function called in line 21 initializes them. We leave the implementation of this device function as an exercise for the reader. Line 22 allocates registers for `Q_i` and the device function `load_Q()` (line 24) loads the corresponding panel from global memory. Note that the total number of registers needed to host

```
1  __device__ inline void load_Q(const float* Q, float Q_i[B_r_warp][
       d_size], int i) {
2    for (int ii = 0; ii < B_r_warp; ii++) {
3      for (int dd = laneIdx(), ddd = 0; dd < d; dd += WARP_SIZE, ddd
           ++) {
4        Q_i[ii][ddd] = Q[(B_r*i+B_r_warp*warpIdx()+ii)*d+dd];
5      }
6    }
7  }
```

FIGURE 20.10

Flash attention: device function to load `Q` into registers.

the tiles of `Q` and `O`, as well as `D_i` and `m_i`, is determined by the dimension `d` and `B_r`. Furthermore, each thread only holds a portion of these tiles and exchange the elements with each other through the warp-shuffle API.

As shown in Fig. 20.8, each panel of `Q` ❶ for a thread block is further divided into sub-panels of size `B_r_warp` × `d`. Each sub-panel is assigned to a warp in the thread block. The device function `load_Q()` that loads a sub-panel of `Q` into registers is shown in Fig. 20.10. Within a sub-panel, each row `ii` is assigned to the threads in the warp in an interleaved manner (note that in this device function and later ones in this chapter, we use the device functions `laneIdx()` and `warpIdx()` defined in Chapter 10). That is, a row is divided into sections of `WARP_SIZE` elements. The elements of a section are assigned sequentially to the threads in the warp. This way, each section of the Q rows can be loaded by the warp in a coalesced manner. With this arrangement, each thread holds `B_r_warp` × `(d/WARP_SIZE)` elements of Q in its registers. For simplicity, we assume that `d` is a multiple of `WARP_SIZE` for our example kernel. Interested readers are encouraged to extend the kernel to handle arbitrary `d` values.[7]

For simplicity, let us first ignore softmax. Without softmax, it is conceptually easier to fuse the three steps of each thread block. In a fused kernel, the thread blocks take an iterative approach to performing the two conceptual matrix multiplications. Each thread block loads its `Q` tile into registers (line 24) and initializes its `O` tile elements to zero by calling device function `initialize()` in line 21 before going into the iterative process that implements the two conceptual steps (matrix-matrix multiplications) in a tiled manner.

The iterative process for a thread block to perform the two conceptual matrix multiplications has two levels: the block level and the warp level. Each iteration at the block level is an iterative process at the warp level. At the block level, each thread

[7] This distribution of `Q` corresponds to the second version of flash attention [8]. It is different from the first version [6] that distributes `K` and `V` across warps (a scheme called *split-K*), which is inefficient because warps need to write intermediate results to shared memory, synchronize, and reduce them. In the second version [8], warps do not need to exchange any intermediate data with each other. All data exchanges are between threads within the same warp.

```
1  __device__ inline void load_KT_and_V(const float* K, const float* V
       , float* KT_j, float V_j[B_c][d], int j) {
2    for (int jj = 0; jj < B_c; jj++) {
3      for (int dd = threadIdx.x; dd < d; dd += blockDim.x) {
4        KT_j[addr(dd * B_c + jj)] = K[(B_c*j+jj) * d + dd];
5        V_j[jj][dd] = V[(B_c * j + jj) * d + dd];
6      }
7    }
8  }
```

FIGURE 20.11

flash attention: device function to load KT and V into shared memory.

block starts from the left end of K^T ❷ and the top of V ❸. All thread blocks iterate from left to right for K^T and from top to bottom for V. Each iteration covers B_c columns of K^T and B_c rows of V. Since there are N columns in K^T and N rows in V, each thread block will take T_c $= \frac{N}{B\_c}$ (line 11) iterations to complete the process.

Each iteration requires a merge operation on all elements in the block's O tile. Therefore, the value of B_c is set by the host code to minimize the number of iterations and thus the number of merges required to produce the O elements (Eq. (20.6)). The value of B_c is constrained by the available shared memory. For simplicity, we only allow B_c value to be multiples of WARP_SIZE, which avoids the need for checking boundary conditions or data padding. The reader is encouraged to extend the code to allow more general B_c values.

In a block-level iteration, each block starts by loading B_c columns of K^T ❷ and B_c rows of V ❺ into the shared memory (lines 27-28). These are the tiles of K^T and V to be used by the block for the iteration. The device function load_KTV() called in line 27 is shown in Fig. 20.11. For efficiency, we load the tile of K^T from K in a transposed manner (line 4). To avoid shared memory bank conflicts, we pad each column of KT_j when allocating it in the shared memory in line 13 of Fig. 20.9. With padding, the index used in each access to KT_j is transformed by the addr(x) = x + (x » LOG_NUM_BANKS) introduced in Chapter 16. The barrier synchronization in line 28 of Fig. 20.9 ensures that all the tiles are in place before proceeding with the first matrix multiplication.

The generation of each O tile is implemented by the nested for loop from line 29 to 38 as a warp-level iterative process. The reader should refer to Fig. 20.8 for better understanding of the code. In an iteration, each warp of the block takes a row of its Q sub-panel ❶, performs a vector-matrix multiplication between the row and the current panel of K^T ❷ to generate the corresponding row of S_i ❸ (compute_S_and_max() in line 31), uses softmax to transform the row of S_i into a segment of P in place ❺ (compute_P_and_update_D() in line 35), and performs a vector-matrix multiplication between the row of S_i and the current panel V ❸ to generate a subset contribution to be merged into its panel of O ❻ (compute_O() in line 37). Since we are ignoring the softmax step for now, we will focus on the two vector-matrix multiplications.

```
1  __device__ inline float compute_S_and_max(const float Q_i[B_r_warp
       ][d_size], const float* KT_j, float S_i[B_r][B_c], int i, int
       j, int ii, float scaling) {
2    typedef cub::WarpReduce<float> WarpReduce;
3    __shared__ typename WarpReduce::TempStorage temp_store;
4    float curr_max = NEG_INFINITY;
5    for (int jj = laneIdx(); jj < B_c; jj += WARP_SIZE) {
6      float S_ij = 0.f;
7      for (int dd = 0; dd < d; dd++) {
8        float q = __shfl_sync(0xFFFFFFFF, Q_i[ii][dd / WARP_SIZE], dd
               % WARP_SIZE);
9        S_ij += q * KT_j[addr(dd * B_c + jj)];
10     }
11     int row = B_r * i + B_r_warp * warpIdx() + ii;
12     int col = B_c * j + jj;
13     S_ij = row < col ? NEG_INFINITY : scaling * S_ij;
14     S_i[B_r_warp * warpIdx() + ii][jj] = S_ij;
15     if(S_ij > curr_max) curr_max = S_ij;
16   }
17   float curr_max_warp = cub::WarpReduce<float>(temp_store).Reduce(
         curr_max, cub::Max());
18   curr_max_warp = __shfl_sync(0xFFFFFFFF, curr_max_warp, 0);
19   return curr_max_warp;
20 }
```

FIGURE 20.12

flash attention: device function to compute a tile of S and its maximum.

For the first vector-matrix multiplication, the kernel calls `compute_S_and_max()`, which Fig. 20.12 shows. The current panel of K^T is further divided into sub-panels of `WARP_SIZE` columns each. The `for` loop in line 5 iterates through the sub-panels. In each iteration, each thread in the warp performs an inner product between the current row of the warp's `Q` sub-panel and a column of the current K^T sub-panel (lines 7-10 in Fig. 20.12). The columns of the K^T sub-panel are assigned to the threads in the warp sequentially. Thus, in each iteration of the `for` loop, the warp performs a vector-matrix multiplication between the current row of the `Q` sub-panel and a `d` $\times$ `WARP_SIZE` K^T sub-panel. The result of the iteration is a segment of `WARP_SIZE` elements in the corresponding row in the `S` tile (see ❸ in Fig. 20.8).

In each iteration of the `for` loop in line 5 of Fig. 20.12, each thread executes the `for` loop in line 7 to sequentially perform an inner product. In each iteration, all threads in the warp need the same `Q` element to multiply with their respective K^T elements. Recall that the row of `Q` elements is stored in an interleaved manner in the registers of threads in the warp (❶ in Fig. 20.8). Thus, each in iteration, the thread who has the `Q` element for that iteration broadcasts the element to all other threads in the same warp using the `__shfl_sync()` intrinsic function in line 8. Line 13 implements the causality policy. It compares the row index and the column index in the entire `S` matrix. If the column index is greater than the row index, we store $-\infty$ in the corresponding shared memory position. If not, we multiply the element `S_ij` by a scaling factor ($\frac{1}{\sqrt{d}}$) and store the result into the `S` tile element in the shared memory (line 14).[8]

[8] Note that the shuffle instruction in line 8 broadcast a value of `Q` from one thread register to threads in the warp. This implementation of matrix multiplication uses tiling in shared memory and registers, but it

```
1   __device__ inline void compute_O(const float S_i[B_r][B_c], const
        float V_j[B_c][d], float O_i[B_r_warp][d_size], int ii, float
        last_m, float* m_i) {
2     for (int dd = laneIdx(), ddd = 0; dd < d; dd += WARP_SIZE, ddd++)
          {
3       O_i[ii][ddd] *= exp(last_m - m_i[ii]);
4       float O_ij = 0.f;
5       for (int jj = 0; jj < B_c; jj++) {
6         O_ij += S_i[B_r_warp*warpIdx()+ii][jj] * V_j[jj][dd];
7       }
8       O_i[ii][ddd] += O_ij;
9     }
10  }
```

FIGURE 20.13

flash attention: device function to compute `O`.

Since we are ignoring softmax for the moment, we will skip to line 37 of Fig. 20.9. In this line, the kernel calls the `compute_O()` device function, which is shown in Fig. 20.13. This device function consists of a nested `for` loop where each warp multiplies its current row of `S_i` with the sub-panel of `V` to produce their contribution to the elements in the output tile of `O`. This is performed as a warp-level iterative process similar to the first vector-matrix multiplication (Fig. 20.12). In the outer `for` loop (line 2 of Fig. 20.13), the warps iterate through the sub-panels of `V`. In each iteration of the outer `for` loop, all threads in the warp collectively perform a vector-matrix multiplication between the row of `S_i` and the current sub-panel of `V`. Each thread executes the inner `for` loop (line 5) to perform an inner product between the row of `S_i` and a column of `V`.

Note that each element of the output matrix `O` is an inner product of one full row of the `S` matrix and one full column of the `V` matrix. Each row in the `S` tile and column in the `V` sub-panel only accounts for a `B_c` section of the inner product for one `O` element. Thus, the warp accumulates the vector-matrix multiplication result between the `S_i` row and the `V` sub-panel into the elements of the `O` tile at the end of each block-level iteration (line 8 of Fig. 20.13).

At the exit of the `for` loop in line 29 of Fig. 20.9, each warp completes an iteration of the block-level iterative process. Each warp has completed the contribution of its assigned sub-panel of `Q`, the current sub-panel of K^T, and the current sub-panel of `V` to its assigned sub-panel of `O`.

We are now ready to add softmax back into each iteration. After generating each row of `S_i`, all threads in a warp collaborate to derive the maximum of the row (`m_i`) and the denominator of the row (`D_i`) ❹. The thread block then uses `m_i` and `D_i` to obtain a `P` row ❺, by applying the softmax function to the row of `S_i` ❸. The `S_i`

could benefit from more sophisticated optimizations (e.g., hierarchical tiling, tensor cores) as shown in Chapter 15.

```
1  __device__ inline float update_m_and_D(float* m_i, float* D_i,
       float curr_max_warp, int ii) {
2    float last_m = m_i[ii];
3    float D = D_i[ii];
4    if (m_i[ii] < curr_max_warp) {
5      m_i[ii] = curr_max_warp;
6      D *= exp(last_m - m_i[ii]);
7    }
8    D_i[ii] = D;
9    return last_m;
10 }
```

FIGURE 20.14

flash attention: device function to update `m` and `D`.

elements can be updated into the corresponding `P` elements in the shared memory. The `P` row is then multiplied with the `V` sub-panel and merged into the `O` tile ❼, using the composition rule in Eq. (20.6).

Lines 15-18 of Fig. 20.12 compute the maximum of the `S` row. First, each thread obtains the current maximum of the `S` elements that it has computed (line 15). Second, we use CUB's `WarpReduce` to compute the maximum inside a warp in line 17. Third, because only thread 0 of a warp receives the result of `WarpReduce`, the shuffle instruction in line 18 broadcasts the maximum to all threads of the warp.

Each thread of the warp saves the maximum in `curr_max_warp` (line 31 of Fig. 20.9) and call `update_m_and_D()` (line 33), which is a device function that is shown in Fig. 20.14. This function updates the maximum and the denominator of the row. Inside `update_m_and_D()` (Fig. 20.14), the threads save the current row maximum `m_i` into `last_m` (line 2), and update it if the maximum `curr_max_warp` calculated for the row is greater than `m_i[ii]` (line 4). Note that these three values correspond to $m_{r,A}$ (`last_m`), $m_{r,B}$ (`curr_max_warp`), and $m_{r,A\cup B}$ (`m_i[ii]`) in Eqs. (20.4) and (20.6), where A is the previous subset of column positions in an `S` row, and B corresponds to the new subset (i.e., the results of the last matrix multiplication). Next, line 6 computes the term $D_{r,B}e^{m_{r,B}-m_{r,A\cup B}}$ of Eq. (20.4).

After updating the maximum and the denominator, the threads call `compute_P_and_update_D()` in line 35 of Fig. 20.9. Fig. 20.15 shows the code of this device function. This function computes the exponential step of softmax and applies the causality policy (line 8) stores them in `S_i` in shared memory (line 9), and accumulates them into `D` (lines 10-14). This corresponds to the sum $\sum_{j\in B} e^{l_{r,j}-m_{r,B}}$ in Eq. (20.4). In line 10, each thread accumulates the exponentials that it has computed into `curr_sum`. Line 12 computes the sum reduction in a warp using CUB's `WarpReduce`. The subsequent shuffle instruction (line 13) broadcasts the reduction result to all threads of the warp, since they will need to use the denominator for the calculation of their `O` elements. Line 14 updates the denominator `D_i[ii]`.

Each element $o_{r,c}$ should receive contributions from all elements of row r of `P`. However, each element of the matrix multiplication between the `P` tile and the `V` tile

```
1  __device__ inline float compute_P_and_update_D(float S_i[B_r][B_c],
       float* m_i, float* D_i, int i, int j, int ii) {
2    typedef cub::WarpReduce<float> WarpReduce;
3    __shared__ typename WarpReduce::TempStorage temp_store;
4    float curr_sum = 0.f;
5    for (int jj = laneIdx(); jj < B_c; jj += WARP_SIZE) {
6      int row = B_r * i + B_r_warp * warpIdx() + ii;
7      int col = B_c * j + jj;
8      float P_ij = row < col ? 0 : (exp(S_i[B_r_warp * warpIdx() + ii
           ][jj] - m_i[ii]));
9      S_i[B_r_warp * warpIdx() + ii][jj] = P_ij;
10     curr_sum += P_ij;
11   }
12   float curr_sum_warp = cub::WarpReduce<float>(temp_store).Reduce(
         curr_sum, cub::Sum());
13   curr_sum_warp = __shfl_sync(0xFFFFFFFF,curr_sum_warp,0);
14   D_i[ii] += curr_sum_warp;
15   return curr_sum_warp;
16 }
```

FIGURE 20.15

flash attention: device function to compute `P` and update `D`.

only carries the contributions from a segment of the row. Thus, one needs to scale the current content of the `O` tile elements, i.e., A in Eq. (20.6), and the matrix multiplications results, i.e., B in Eq. (20.6), using Eq. (20.6) before accumulating the matrix multiplication result into the output tile.

The `compute_O()` device function, called in line 37 of Fig. 20.9, performs the merge operation that corresponds to Eq. (20.6). This function is in Fig. 20.13 and we discussed the vector-matrix multiplication of `S_i` and `V` above. To perform the merge operation, line 3 multiplies `O_i[ii][ddd]` by `exp(last_m - m_i[ii])`, which corresponds to the numerator of the first addend of the equation. As we discussed above, the inner `for` loop (line 5 of Fig. 20.13) performs the inner product between the row of the `S_i` and a column of `V` and merges the second addend of the equation. Note that the new contributions are computed based on the maximal of all `S` elements up to the new row in line 8 of Fig. 20.15. There is no need to rescale these P elements when merging them into `O`.

At the end of the iterative process for a thread block, its output tile contains the full contributions from the conceptual `S` and `P` matrices although they were never fully generated and only existed in tiled form in the shared memory. In line 42, the threads call the `store_O` device function, which stores the output tile `O_i` and the denominators of softmax `D_i` into `out_O` and `out_D` in global memory. This function is shown in Fig. 20.16. Note that we divide the output values by the denominator, same as in Eq. (20.6), right before storing into global memory (line 6).

Note that the iterative process described in this section is only one of the many possibilities. One would use a different process, for example, if the tensor cores are used to compute the two matrix multiplications performed by each warp. We encourage interested readers to explore these potential variations.

```
__device__ inline void store_O(float* out_O, float* out_D, const
     float O_i[B_r_warp][d_size], const float D_i[B_r_warp], int i,
      int N) {
  for (int ii = 0; ii < B_r_warp; ii++) {
    int row = B_r * i + B_r_warp * warpIdx() + ii;
    for (int dd = 0; dd < d_size; dd++) {
      int col = dd * WARP_SIZE + laneIdx();
      if(row<N && col<d) out_O[row*d+col] = O_i[ii][dd]/D_i[ii];
    }
    if(laneIdx() == 0) out_D[row] = D_i[ii];
  }
}
```

FIGURE 20.16

flash attention: device function to store `O` and the denominator into global memory.

The flash attention kernel in Fig. 20.9 applies to the baseline transformer implementation without KV caching. It also applies to the prefill phase of inference with KV caching. For the generation (decode) phase, with KV caching, Q, S, P, and O matrices become vectors Q', S', P', and O'. Thus, the computations inside a single head change from matrix multiplications to vector-matrix multiplications. To maintain a similar work distribution to that in Fig. 20.8, we can batch multiple requests, i.e., multiple `Q_i`, into a single matrix. We will discuss batching in the next section.

The most recent version of flash attention [9] takes advantage of new capabilities of the Hopper GPU architecture for an additional performance boost. These capabilities are (1) the Tensor Memory Accelerator (TMA), which can asynchronously move tiles from global memory to shared memory for use by the tensor cores without involving compute threads, (2) the tensor core WarpGroup-wide Matrix Multiply Accumulate (WGMMA) instructions, which are asynchronous in contrast to prior architectures, and (3) FP8 precision of the tensor cores. By using warp specialization, this flash attention version achieves better overlap of data movement and computation, and hides softmax execution with asynchronous block-wise GEMMs.

20.6 KV cache arithmetic intensity and memory requirement

Fig. 20.2 illustrates the generation of the K and V matrices during the prefill phase of an LLM inference with KV caching. K and V elements are calculated and cached in memory. Then, during the generation phase (Fig. 20.6), one new row of K and one new row of V are generated for each new token and appended to the KV cache. These K and V rows are generated with vector-matrix operations, which have low arithmetic intensity and underutilize GPU compute resources (see Chapter 5). For this reason, there are a number of optimizations, such as *batching* and *speculative decoding* that aim to increase the arithmetic intensity of the generation phase.

Batching is a common optimization in all types of DNNs, and has traditionally been used to increase their arithmetic, i.e., computational, intensity. For example, a

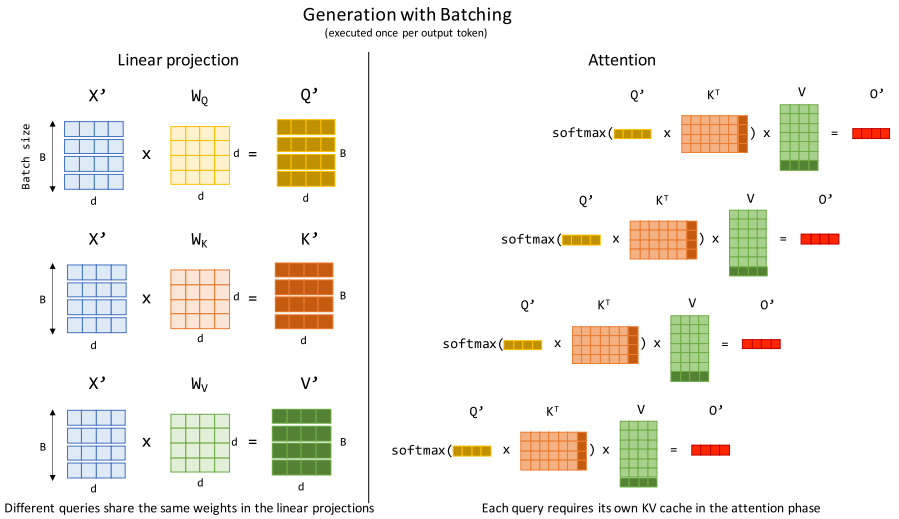


FIGURE 20.17

Batching in LLM inference.

linear layer with 1024 inputs and 4096 outputs (in 16-bit precision) has a very low arithmetic intensity of 1 floating point operation per byte (FLOPS/B) when the batch size is 1, but its arithmetic intensity increases to 315 FLOPS/B with batch size 512.[9] The increased arithmetic intensity is due to the reuse of weights across input vectors. Such a high arithmetic intensity is critical for high GPU utilization. However, LLMs impose a tradeoff between batch size and context length (i.e., KV cache size). In LLM inference, since input queries from multiple users use the same model (e.g., in a chatbot LLM), the weights can be shared among them. This increases the utilization of the GPU in different parts of the computation. For example, the left part of Fig. 20.17 illustrates batching in the fully-connected layers that generate matrices Q, K, and V. Batching increases the arithmetic intensity of the QKV projection by as many times as the batch size. However, note that the KV cache is dependent on the user's prompt and context. Thus, attention layers do not benefit from batching because they need to maintain and access separate KV caches for different user's conversations. As a result, batching in LLMs is limited due to the memory requirements of the model weights and the KV cache.

Let's analyze the memory requirements of LLM inference with KV caching and batching. The two main contributors are the model weights (e.g., matrices W_Q, W_K, and W_V) and the KV cache. The memory size of the model weights depends on the number of parameters of the model (typically, an order of magnitude of several

[9] This is the theoretical increase of the arithmetic intensity. In practice, it depends on the tile size used for the implementation.

billions). For example, a model with 7 billion parameters loaded in 16-bit format (FP16 or BF16) takes about 14 GB. Interested readers are referred to the literature for details on the memory requirements of model weights (e.g., [10,11]).

The size of the KV cache depends on the number of transformer layers (l) and the hidden dimension. Recall that the hidden dimension is defined as the product of number of (attention) query heads (h_q) and the embedding dimension (d). Eq. (20.7) shows the size of the KV cache per token in bytes for multi-head attention (MHA). Note that we multiply by 2 to include both K and V matrices. p is the precision in bytes.

$$KV\_cache\_size\_MHA = 2 \times l \times h_q \times d \times p \tag{20.7}$$

To estimate the memory requirements in a real scenario, we need to consider the KV cache size across all tokens of the sequences in a batch. Eq. (20.8) show the total size of the KV cache across a batch of b sequences of a certain length N.

$$Total\_KV\_cache\_size\_MHA = b \times N \times 2 \times l \times h_q \times d \times p \tag{20.8}$$

For example, let us find the total KV cache size in 16-bit precision for 1 sequence of 4096 tokens for different example LLMs. For the PaLM 2 LLM from Google, the total KV cache size is $1 \times 4096 \times 2 \times 64 \times 8192 \times 2 = 8$ GB, where 64 is the number of transformer layers (l) and 8192 is the hidden dimension ($h_q \times d$). For the GPT-3 LLM from OpenAI, the total size is $1 \times 4096 \times 2 \times 96 \times 12288 \times 2 = 18$ GB. For the Llama 2 7B LLM from Meta, the total size is $1 \times 4096 \times 2 \times 32 \times 4096 \times 2 = 2$ GB. With such large KV cache sizes, memory requirements for batching multiple queries in LLM inference can easily exceed the memory capacity of a single GPU. As a result, multi-GPU and multi-node systems are commonly necessary for LLM inference.

Note that in the previous analysis, we assume that all sequences of a batch are of the same length. However, in reality, since queries from different users are different, the input sequences as well as the generated output sequences will likely have very different lengths across different users. Thus, batching in LLM inference may cause imbalance across thread blocks, which would result in under-utilization of the GPU resources. To avoid this situation, state-of-the-art LLM model serving systems apply *in-flight batching* mechanisms [12], which schedule and execute batches of multiple requests at a time and schedule a new request to a batch with the lowest workload as soon as a previous one is served.

As mentioned before, while batching is effective at increasing the arithmetic intensity (AI) of linear projection operations, batching is not effective at increasing the arithmetic intensity of MHA during inference. We can estimate the arithmetic intensity of MHA by approximating the amount of data movement and the number of floating point operations [13]. For one layer and sequence length N (and assuming $p = 1$), from Eq. (20.7), the KV cache size is $2 \times h_q \times d \times N$. The amount of data movement for reading Q' and writing O' is $2 \times h_q \times d$. The number of floating point

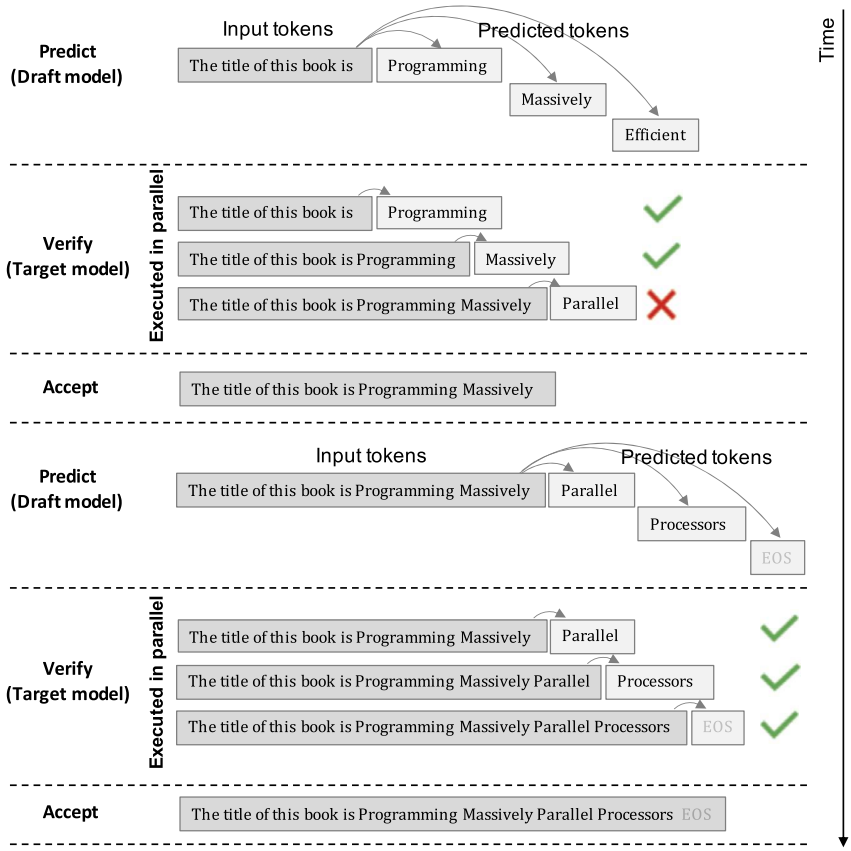


FIGURE 20.18

Speculative decoding.

operations to compute $(Q'K^T)V$ is $2 \times N \times h_q \times d$ (we ignore softmax in this estimate). Eq. (20.9) shows that the AI of MHA is approximately as low as 1 floating point operation per byte.

$$AI_{MHA} \approx \frac{2 \times N \times h_q \times d}{2 \times h_q \times d + 2 \times h_q \times d \times N} = \frac{N}{1+N} \approx 1 \tag{20.9}$$

A different approach to increase the arithmetic intensity of LLM inference is *speculative decoding* [14]. This is a model service optimization where a tiny model called the *draft* model, predicts the output tokens of a large model called the *target* model. The target model verifies the predictions by the draft model. While the target model has much longer latency, it performs verification on batches of tokens produced by the draft model in parallel.

Fig. 20.18 shows an example of speculative decoding. In this example, the *speculation depth* is 3, i.e., the draft model predicts three consecutive output tokens. Then,

the target model verifies the three tokens in parallel. In the first iteration of this example, the target model accepts "Programming" and "Massively" but rejects "Efficient". After the verification by the target model, the accepted tokens are incorporated to the input tokens for the second iteration of the draft model, which will predict three new tokens.

Speculative decoding is similar to batching, which is why it increases the arithmetic intensity of the linear projection operations. However, it is a special form of batching where all the tokens in the batch have a common initial set of tokens. For this reason, when applying the attention mechanism, the same KV cache data for these common input tokens is loaded and reused to compute all the output tokens. This reuse of KV cache data is what results in the increase in arithmetic intensity. Without speculative decoding, this KV cache data would have been loaded multiple times, once for each output token.

20.7 Alleviating the memory requirements of the attention mechanism

As discussed above, the attention mechanism employs multiple heads. Each of them uses different matrices of weights to perform different projections of Q, K, and V matrices. This is a main reason for the large memory requirements of multi-head attention (MHA) that Eq. (20.8) shows. Two approaches have been proposed to reduce the memory requirements while maintaining an acceptable model accuracy: *multi-query attention* (MQA) and *grouped-query attention* (GQA). Figs. 20.19(a), (b), and (c) depict in a simplified way the memory requirements of MHA, MQA, and GQA, respectively.

MQA shares keys and values across the attention heads. This reduces the size of the hidden dimension. Eq. (20.10) shows the total KV cache size for MQA. The reduction with respect to Eq. (20.8) is inversely proportional to the number of heads.

$$Total\_KV\_cache\_size\_MQA = b \times N \times 2 \times l \times d \times p \tag{20.10}$$

As a result of this reduction in the KV cache size, the AI of MQA is approximately h_q times higher than that of MHA, as Eq. (20.11) shows.

$$AI_{MQA} \approx \frac{2 \times N \times h_q \times d}{2 \times h_q \times d + 2 \times d \times N} = \frac{N \times h_q}{h_q + N} \approx h_q \tag{20.11}$$

GQA [15] balances between MHA and MQA by using a few groups (g_q) of query heads. Eq. (20.12) shows the total memory requirements of GQA.

$$Total\_KV\_cache\_size\_GQA = b \times N \times 2 \times l \times \frac{h_q}{g_q} \times d \times p \tag{20.12}$$

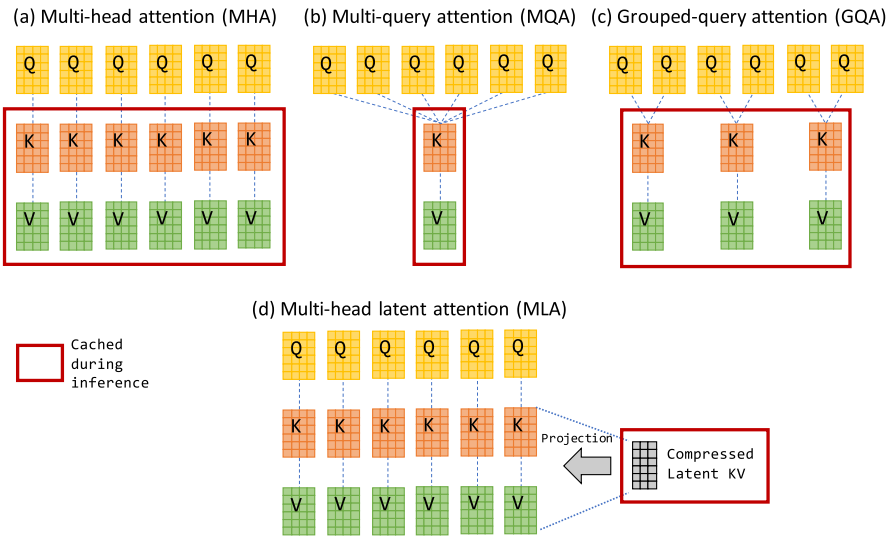


FIGURE 20.19

Q, K, and V matrices in multi-head attention, multi-query attention, grouped-query attention, and multi-head latent attention.

In the case of GQA, the AI increases by the number of groups g_q, as Eq. (20.13) shows.

$$AI_{GQA} \approx \frac{2 \times N \times h_q \times d}{2 \times h_q \times d + 2 \times \frac{h_q}{g_q} \times d \times N} = \frac{N \times h_q}{h_q + \frac{h_q}{g_q} \times N} \approx g_q \tag{20.13}$$

All three of MHA, GQA, and MQA can take advantage of the flash attention tiling approach explained in Section 20.5. The three alternatives entail different tradeoffs in terms of computation, memory requirements, and model accuracy.

Although MQA and GQA alleviate the memory requirements of the KV cache, the batch size and the context length can be still constrained by the limited GPU memory capacity. This limitation is worsened by the fact that the memory reserved for KV caches is typically over-provisioned for the largest possible input. As a result, memory tends to be fragmented and wasted. An example attempt to overcome this scenario is the PagedAttention algorithm [12], which is inspired by the traditional paging mechanisms of operating systems. PagedAttention partitions the KV cache into blocks of equal size that are loaded into the GPU memory (from the larger system memory) when they are needed for computation.

Another approach to managing the KV cache is Multi-head Latent Attention (MLA) [16], which employs low-rank joint compression of K and V to reduce memory usage drastically. The key idea of MLA is to compress both K and V vectors for each token into a compressed latent vector (Fig. 20.19(d)). This compression is done

in all heads and layers so the reduction in the KV caching memory requirement can be very significant, as Eq. (20.14) shows.

$$Total\_KV\_cache\_size\_MLA = b \times N \times l \times d \times p \tag{20.14}$$

The arithmetic intensity (Eq. (20.15)) increases due to sharing a single latent head across a large number of query heads.

$$AI_{MLA} \approx \frac{2 \times N \times h_q \times d}{2 \times h_q \times d + d \times N} = \frac{2 \times N \times h_q}{h_q + N} \approx 2 \times h_q \tag{20.15}$$

The projection matrices used in the compression scheme are learned during training. During inference, K and V are generated (decompressed) by multiplying their latent representations by two up-projection matrices before they are used in the attention calculation. As a result, K and V only need to be preserved in their latent, compressed form between iterations in KV caches of all attention heads and decompressed only when a head is active during the generation phase of an LLM inference.

A different, orthogonal technique for scaling LLMs is the *Mixture of Experts* technique (MoE, e.g., Mixtral [17]). MoE is a sparsely activated architecture where a gating mechanism decides what small subset of specialized sub-networks is used for each input. This allows models to be huge in parameter count but have reduced computational needs per token.

20.8 Summary

In this chapter, we introduced the operations that underlie Large Language Models (LLMs). As the leading LLMs are based on the Transformer Architecture, we focus on the unique computational components of Transformers, namely attention heads. We start with a naive, multi-kernel approach with emphasis on the softmax kernel, which is the most novel component of the attention heads. We then discuss KV caching, an important optimization that reduces the redundant computation in each iteration of the generation phase of an LLM inference. We further presented the flash attention formulation and the resulting attention head kernel that improves the arithmetic intensity and reduces the latency of the prefill phase of the LLM inference. The chapter ends with a discussion of the memory requirements of KV caching and techniques to alleviate such requirements when batching is used to improve the GPU utilization during inference. It should be clear that this chapter is designed to provide the reader with a clear understanding of the fundamentals of LLM implementations on GPUs. There is a very large design space that the reader is ready to explore after reading this chapter.

20.9 Exercises

1. Complete the host code for invoking the flash attention kernel in Fig. 20.9. Refer to Fig. 20.8 when setting the sizes of the input and output arrays based on the dimensions of the matrices they hold.
2. Read the CUB document for details of the data types in the WarpReduce namespace and the semantics of the arguments when calling the Reduce function. Pay special attention to the variety of arithmetic functions available to the reduction.
3. Write the initialization device function in line 21 in Fig. 20.9. Explain the reasons why `O_i` and `D_i` elements need to be initialized to `0.f`.
4. Show that all the loads from `K` and `V` in the nested `for` loop in the device function of Fig. 20.11 are coalesced.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 6000–6010.

[2] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A.A. Kalyanpur, A. Lally, J.W. Murdock, E. Nyberg, J. Prager, et al., Building Watson: an overview of the DeepQA project, AI Magazine 31 (3) (2010) 59–79.

[3] OpenAI, ChatGPT, https://chat.openai.com/, 2025. (Accessed 25 January 2025).

[4] A. Gu, T. Dao, Mamba: linear-time sequence modeling with selective state spaces, arXiv preprint, arXiv:2312.00752, 2023.

[5] M. Poli, S. Massaroli, E. Nguyen, D.Y. Fu, T. Dao, S. Baccus, Y. Bengio, S. Ermon, C. Ré, Hyena hierarchy: towards larger convolutional language models, in: International Conference on Machine Learning, PMLR, 2023, pp. 28043–28078.

[6] T. Dao, D. Fu, S. Ermon, A. Rudra, C. Ré, FlashAttention: fast and memory-efficient exact attention with IO-awareness, Advances in Neural Information Processing Systems 35 (2022) 16344–16359.

[7] M. Milakov, N. Gimelshein, Online normalizer calculation for softmax, arXiv preprint, arXiv:1805.02867, 2018.

[8] T. Dao, FlashAttention-2: faster attention with better parallelism and work partitioning, arXiv preprint, arXiv:2307.08691, 2023.

[9] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, T. Dao, FlashAttention-3: fast and accurate attention with asynchrony and low-precision, arXiv preprint, arXiv:2407.08608, 2024.

[10] Z. Zhang, D. Zhao, X. Miao, G. Oliaro, Q. Li, Y. Jiang, Z. Jia, Quantized side tuning: fast and memory-efficient tuning of quantized large language models, arXiv preprint, arXiv:2401.07159, 2024.

[11] J. Zhao, Z. Zhang, B. Chen, Z. Wang, A. Anandkumar, Y. Tian, Galore: memory-efficient llm training by gradient low-rank projection, arXiv preprint, arXiv:2403.03507, 2024.

[12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C.H. Yu, J. Gonzalez, H. Zhang, I. Stoica, Efficient memory management for large language model serving with PagedAttention, in: Proceedings of the 29th Symposium on Operating Systems Principles, 2023, pp. 611–626.

[13] T. Zadouri, H. Strauss, T. Dao, Hardware-efficient attention for fast decoding, arXiv preprint, arXiv:2505.21487, 2025.
[14] Y. Leviathan, M. Kalman, Y. Matias, Fast inference from transformers via speculative decoding, in: International Conference on Machine Learning, PMLR, 2023, pp. 19274–19286.
[15] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, S. Sanghai, GQA: training generalized multi-query transformer models from multi-head checkpoints, arXiv preprint, arXiv:2305.13245, 2023.
[16] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo, et al., Deepseek-v2: a strong, economical, and efficient mixture-of-experts language model, arXiv preprint, arXiv:2405.04434, 2024.
[17] A.Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D.S. Chaplot, D.d.l. Casas, E.B. Hanna, F. Bressand, et al., Mixtral of experts, arXiv preprint, arXiv: 2401.04088, 2024.

CHAPTER

Electrostatic potential map

21

With special contributions from John Stone

CONTENTS

21.1 Background 513
21.2 Scatter vs. gather in kernel design 515
21.3 Thread coarsening 519
21.4 Memory coalescing 521
21.5 Cutoff binning for data size scalability 523
21.6 Summary 527
21.7 Exercises 528
References 528

In this chapter, we use a molecular dynamics application based on regular grid data structures to illustrate the use of optimization techniques that achieve global memory access coalescing and improved computation throughput. We present a series of implementations of an electrostatic potential map calculation kernel, with each version improving upon the previous one. Each version adopts one or more practical techniques from Chapter 6. Some of the techniques are in common with the previous chapters but some are different: gather-to-scatter transformation, systematic reuse of computational results, and fast boundary condition checking. The effective use of these practical techniques can significantly improve application speed.

21.1 Background

The application studied in this chapter is based on VMD (Visual Molecular Dynamics) [1], a popular software system designed for displaying, animating, and analyzing bio-molecular systems. It is an important foundation for modern "computational microscopes" for biologists to observe tiny life forms such as viruses that are too small for traditional microscopy techniques. While it has strong built-in support for analyzing bio-molecular systems such as calculating electrostatic potential values at spatial grid points of a molecular system (the focus of this chapter), it has also been a popular tool for displaying other large data sets such as sequencing data, quantum chemistry simulation data, and volumetric data due to its versatility and user extensibility.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00032-9
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

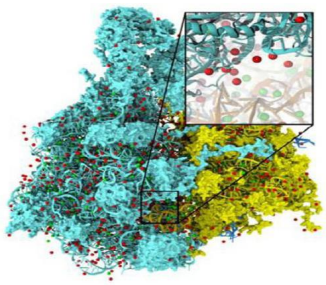


FIGURE 21.1

Electrostatic potential map is used in building stable structures for molecular dynamics simulation.

Most users use VMD as a desktop science application for interactive 3D visualization and analysis. For computation that runs too long for interactive use, VMD can also be used in a batch mode to render movies for later use. A motivation for accelerating VMD is to make batch mode jobs fast enough for interactive use. This can drastically improve the productivity of scientific investigations. With CUDA devices widely available in desktop PCs, such acceleration can have broad impact on the VMD user community. To date, multiple aspects of VMD have been accelerated with CUDA, including electrostatic potential map calculation, ion placement, molecular orbital calculation and display, and imaging of gas migration pathways in proteins.

The computation covered in this case study is the calculation of electrostatic potential maps in a grid space. This calculation is often used in placement of ions into a molecular structure for molecular dynamics simulation. Fig. 21.1 shows the placement of ions (small dots) into a protein structure in preparation for molecular dynamics simulation. In this application, the electrostatic potential map is used to identify spatial locations where the ions can fit in according to physical laws. The function can also be used to calculate time-averaged electrical field potential maps during molecular dynamics simulation, which is useful for the simulation process as well as the visualization/analysis of simulation results.

There are several methods for calculating electrostatic potential maps. Among them, Direct Coulomb Summation (DCS) is a highly accurate method that is particularly suitable for GPUs [2]. The DCS method calculates the electrostatic potential value of each grid point as the sum of contributions from all atoms in the system. This is illustrated in Fig. 21.2. The contribution of atom i to a lattice point j is the charge of atom i divided by the distance from lattice point j to atom i. Since this needs to be done for all grid points and all atoms, the number of calculations is proportional to the product of the total number of atoms in the system and the total number of grid points. For a realistic molecular system, this product can be very large. Therefore, the calculation of the electrostatic potential map has been traditionally done as a batch job in VMD.

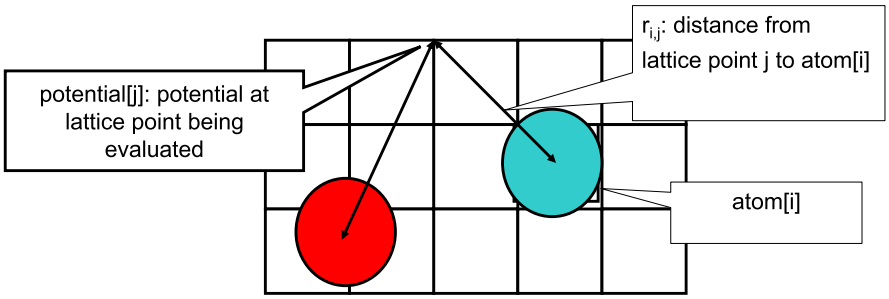


FIGURE 21.2

The contribution of atom[i] to the electrostatic potential at lattice point j (potential[j]) is atom[i].charge/r_{ij}. In the Direct Coulomb Summation method, the total potential at lattice point j is the sum of contributions from all atoms in the system.

```
01 void cenergy(float *energygrid, dim3 grid, float gridspacing, float z,
02               const float *atoms, int numatoms) {
03   int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
04   for (int j=0; j<grid.y; j++) {
        // calculate y coordinate of the grid point based on j
05      float y = gridspacing * (float) j;
06      for (int i=0; i<grid.x; i++) {
           // calculate x coordinate based on i
07         float x = gridspacing * (float) i;
08         float energy = 0.0f;
09         for (int n=0; n<atomarrdim; n+=4) {
10            float dx = x - atoms[n  ];
11            float dy = y - atoms[n+1];
12            float dz = z - atoms[n+2];
13            energy += atoms[n+3] / sqrtf(dx*dx + dy*dy+ dz*dz);
14         }
15         energygrid[grid.x*grid.y*z + grid.x*j + i] = energy;
16      }
17    }
18 }
```

FIGURE 21.3

An unoptimized DCS C code for a 2D slice.

21.2 Scatter vs. gather in kernel design

Fig. 21.3 shows the base C code of the DCS code. The function is written to process a two-dimensional slice of a three-dimensional grid. The function is called repeatedly for all the slices of the modeled space. The structure of the function is quite simple with three levels of for loops. The outer two levels iterate over the y-dimension (line 04) and the x-dimension (line 06) of the grid point space. For each grid point, the innermost for loop iterates over all atoms (line 09), calculating the contribution of electrostatic potential energy from all atoms to the grid point (lines 10-13). Note that each atom is represented by four consecutive elements of the `atoms[]` array. The first

```
01 void cenergy(float *energygrid, dim3 grid, float gridspacing, float z,
02               const float *atoms, int numatoms) {
03   int atomarrdim = numatoms * 4; //x,y,z, and charge info for each atom
     // starting point of the slice in the energy grid
04   int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
     //  calculate potential contribution of each atom
05   for (int n=0; n<atomarrdim; n+=4) {
06      float dz = z - atoms[n+2];  // all grid points in a slice have the same
07      float dz2 = dz*dz;          // z  value, no recalculation in inner loops
08      float charge = atoms[n+3];
09      for (int j=0; j<grid.y; j++) {
10         float y = gridspacing * (float) j;
11         float dy = y - atoms[n+1];  // all grid points in a row have the same
12         float dy2 = dy*dy;          // y value
13         int grid_row_offset =  grid_slice_offset+ grid.x*j;
14         for (int i=0; i<grid.x; i++) {
15            float x = gridspacing * (float) i;
16            float dx = x - atoms[n    ];
17            energygrid[grid_row_offset+i] += charge / sqrtf(dx*dx + dy2+ dz2);
18         }
19      }
20    }
21 }
```

FIGURE 21.4

An optimized DCS C code for a 2D slice.

three elements store the x, y, and z coordinates of the atom and the fourth element the electrical charge of the atom. After the innermost loop, the accumulated value of the grid point is written out to the grid data structure (line 15). The outer loops then iterate and take the execution to the next grid point.

Note that DCS function in Fig. 21.3 calculates the x and y coordinates of each grid point on the fly by multiplying the grid point index values by the spacing between grid points. This is a uniform grid method where all grid points are spaced at the same distance in all three dimensions. The function takes advantage of the fact that all the grid points in the same slice have the same z coordinate. This value is pre-calculated by the caller of the function and passed in as a function parameter (z). There are, however, several optimizations that can be done to the sequential C code in Fig. 21.3 to significantly improve its execution speed.

Fig. 21.4 shows a C code for DCS with a few optimizations to improve its execution speed and efficiency. First, the inner most loop on line 09 of Fig. 21.3 has been exchanged into the outer most loop on line 05 in Fig. 21.4. Thus, the code iterates over all atoms. For each atom, the inner loops (line 09 and line 14) scatter the contribution of the atom to all the grid points. The loop interchange is permissible because the three levels of the loops in Fig. 21.3 are perfectly nested, and the all the iterations are independent of each other. The net effect of the loop interchange is that the execution order of the innermost loop body instances is changed. Since all these instances are independent of each other, they can be executed in any order without affecting the final outcome.

The loop interchange enables two optimizations. First, the z components of the distances between an atom and all grid points in the plane are identical and can be calculated once for the entire plane. Therefore, the calculation can be done outside

```
01  __constant__ float atoms[CHUNK_SIZE*4];
02  void  __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
03                           float z) {
04    int n = (blockIdx.x * blockDim .x + threadIdx.x) * 4;
05    float dz = z-atoms[n+2];  // all grid points in a slice have the same
06    float dz2 = dz*dz;        // z value
      // starting position of the slice in the energy grid
07    int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
08    float charge = atoms[n+3];
09    for (int j=0; j<grid.y; j++) {
10       float y = gridspacing * (float) j;
11       float dy = y-atoms[n+1];   // all grid points in a row have the same
12       float dy2 = dy*dy;      // y value
         // starting position of the row in the energy grid
13       int grid_row_offset =  grid_slice_offset+ grid.x*j;
14       for (int i=0; i<grid.x; i++) {
15           float x = gridspacing * (float) i;
16           float dx = x - atoms[n    ];
17           atomicAdd(&energygrid[grid_row_offset+i],
18                     charge / sqrtf(dx*dx+dy2+dz2));
19       }
20    }
21  }
```

FIGURE 21.5

DCS Kernel using the scatter approach.

the two inner loops (lines 06-07). Similarly, the y components of the distances between an atom and all grid points in the same row are identical and can be calculated outside the innermost loop (lines 11-12). In comparison, the y and z components of the distance were both calculated in the innermost loop in Fig. 21.3. This drastic reduction in the number of calculations makes the C code in Fig. 21.4 faster. These optimizations cannot be done in Fig. 21.3 because the innermost loop iterates over all atoms so one must recalculate the x, y, and z components of the distance as they change from atom to atom.

For GPU execution, we assume that the host program inputs and maintains the atomic charges and their coordinates in the system memory. It also maintains the grid point data structure in the system memory. The DCS kernel is designed to process a 2D slice of the electrostatic potential grid point structure (not to be confused with thread grids). These grid points are like the grid points for discretization that was discussed in Chapter 8. For each 2D slice, the CPU transfers its grid data to the device global memory. The atom information is divided into chunks to fit into the constant memory. For each chunk of the atom information, the CPU transfers the chunk into the device constant memory, invokes the DCS kernel to calculate the contribution of the current chunk to the current slice, and prepares to transfer the next chunk. After all chunks of the atom information have been processed for the current slice, the slice is transferred back to update the grid point data structure in the CPU system memory. The system moves on to the next slice.

Let us now focus on the design of DCS kernel. It is natural to parallelize the optimized C code in Fig. 21.4. The resulting kernel is shown in Fig. 21.5. The defined constant `CHUNK_SIZE` specifies the number of atoms that should be transferred into the GPU constant memory for each kernel call. The value of `CHUNK_SIZE*4` should be

```
01  __constant__ float atoms[CHUNK_SIZE*4];
02  void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
03                          float z, int numatoms) {
04     int i = blockIdx.x * blockDim.x + threadIdx.x;
05     int j = blockIdx.y * blockDim.y + threadIdx.y;
06     int atomarrdim = numatoms * 4;
07     int k = z / gridspacing;
08     float y = gridspacing * (float) j;
09     float x = gridspacing * (float) i;
10     float energy = 0.0f;
       // calculate potential contribution from all atoms
11     for (int n=0; n<atomarrdim; n+=4) {
12        float dx = x - atoms[n  ];
13        float dy = y - atoms[n+1];
14        float dz = z - atoms[n+2];
15        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
16     }
17     energygrid[grid.x*grid.y*k + grid.x*j + i] += energy;
18   }
```

FIGURE 21.6

DCS Kernel using the gather approach.

set so that the array would fit in the constant memory. The kernel uses each thread to implement an iteration of the outermost loop in Fig. 21.4 and scatters the contribution of its assigned atom to all grid points. That is, each thread *scatters* the contribution of an atom to all grid points. Unfortunately, this scatter approach to parallelization requires atomic operations for updating the energy grid points (lines 17-18), which significantly reduces the speed of parallel execution.

We can instead use a *gather approach* where each thread calculates the accumulated contributions of all atoms to one grid point. That is, each thread gathers the contributions from all atoms to a grid point. This is a preferred approach since each thread will be writing into its own grid point and there is no need to use atomic operations. This, however, requires the loops to be arranged in the ordering of the unoptimized C code in Fig. 21.3. That is, we would be parallelizing a slower C implementation and exemplifies a frequently experienced dilemma in parallelizing applications: the optimized sequential code is not as amenable to parallelization as the unoptimized sequential code. The downside is that we can end up with drastically slower execution within each thread, which can reduce the speed benefit of parallelization. We will return to this point later in this chapter.

Fig. 21.6 shows a kernel based on the gather approach. The kernel is based on the unoptimized C code in Fig. 21.3. We form a two-dimensional thread grid that matches the two-dimensional potential grid point organization. To do so, we need to modify the two outer loops in lines 04-06 of Fig. 21.3 into perfectly nested loops so that we can use each thread to execute one iteration of the 2-level loop. This is done by moving the calculation of the y coordinate (Line 05 of Fig. 21.3) into the inner loop. This transformation allows all `i` and `j` iterations to be executed in parallel. The cost is that the y-coordinate calculation is performed redundantly in all inner loop iterations. This is a tradeoff between the amount of calculation done and the level of parallelism achieved.

Inside the kernel code in Fig. 21.6, the outer two levels of the loop in Fig. 21.3 have been removed and are replaced by the execution configuration parameters in the kernel invocation (lines 04-05 of Fig. 21.6). Within each thread grid, the thread blocks are organized to calculate the electrostatic potential of tiles of the grid structure. In the simplest kernel, each thread calculates the value at one grid point. In more sophisticated kernels, each thread calculates multiple grid points and exploits the redundancy between the calculations of the grid points to improve execution speed. This is an example of the thread coarsening optimization introduced in Chapter 6 and is discussed in Section 21.3.

The performance of the kernel in Fig. 21.6 is quite good since its execution speed is not hampered by atomic operations. Also, a quick glance over the code shows that each thread does 9 floating point operations for every 4 `atom[]` array elements accessed. These `atoms[]` array elements for each atom are cached in a hardware constant cache memory in each SM and are broadcast to many threads. The massive re-use of these constant memory elements across threads makes the constant cache extremely effective, eliminating the vast majority of the DRAM accesses. As a result, global memory bandwidth is not a limiting factor for this kernel.

21.3 Thread coarsening

Although the kernel in Fig. 21.6 avoids the global memory bottleneck through constant caching, it still needs to execute four constant memory access instructions for every 9 floating point operations performed. These memory access instructions consume hardware resources that could be otherwise used to increase the execution throughput of floating-point instructions. Furthermore, the execution of these memory access instructions consumes energy, an important limiting factor for many large scale parallel computing systems. This section shows that we can use the thread coarsening technique to fuse several threads together so that the `atoms[]` data can be fetched once from the constant memory, stored into registers, and used for computing multiple grid points.

Furthermore, as shown in Fig. 21.7, all energy grid points along the same row (y-dimension) have the same y-coordinate. Therefore, the difference between the y-coordinate of an atom and the y-coordinate of any grid point along a row has the same value. In the DCS Kernel in Fig. 21.6, this calculation is redundantly done by all threads for all grid points in a row when calculating the distance between the atom and the grid points. We can eliminate this redundancy and improve the execution efficiency.

The idea is to have each thread calculate the electrostatic potential for multiple energy grid points in the same row. The kernel in Fig. 21.8 has each thread calculate four grid points. For each atom, the code calculates `dy`, the difference of the y-coordinates, only once (line 21). It then calculates the expression `dy*dy + dz*dz` and saves it to the auto variable `dysqdzsq`, which is assigned to a register (line 23). This value is the same for all four grid points. Furthermore, the electrical charge information is also

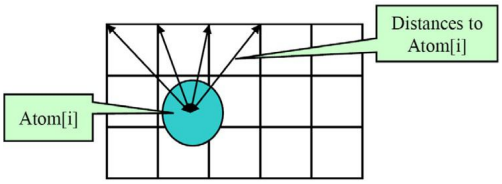


FIGURE 21.7

Reusing computation results among multiple grid points.

```
01  __constant__ float atoms[CHUNK_SIZE*4];
02  #define COARSEN_FACTOR 4
03  void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
04                          float z, int numatoms) {
05     int i = blockIdx.x * blockDim.x*COARSEN_FACTOR + threadIdx.x;
06     int j = blockIdx.y * blockDim.y + threadIdx.y;
07     int atomarrdim = numatoms * 4;
08     int k = z / gridspacing;
09     float y = gridspacing * (float) j;
10     float x = gridspacing * (float) i;
11     float energy0 = 0.0f;
12     float energy1 = 0.0f;
13     float energy2 = 0.0f;
14     float energy3 = 0.0f;
15     // calculate potential contribution from all atoms
16     for (int n=0; n<atomarrdim; n+=4) {
17        float dx0 = x - atoms[n  ];
18        float dx1 = dx0 +   gridspacing;
19        float dx2 = dx0 + 2*gridspacing;
20        float dx3 = dx0 + 3*gridspacing;
21        float dy = y - atoms[n+1];
22        float dz = z - atoms[n+2];
23        float dysqdzsq = dy*dy + dz*dz;
24        float charge = atoms[n+3];
25        energy0 += charge / sqrtf(dx0*dx0 + dysqdzsq);
26        energy1 += charge / sqrtf(dx1*dx1 + dysqdzsq);
27        energy2 += charge / sqrtf(dx2*dx2 + dysqdzsq);
28        energy3 += charge / sqrtf(dx3*dx3 + dysqdzsq);
29     }
30     energygrid[grid.x*grid.y*k + grid.x*j + i  ] += energy0;
31     energygrid[grid.x*grid.y*k + grid.x*j + i+1] += energy1;
32     energygrid[grid.x*grid.y*k + grid.x*j + i+2] += energy2;
33     energygrid[grid.x*grid.y*k + grid.x*j + i+3] += energy3;
34   }
```

FIGURE 21.8

DCS kernel with thread coarsening.

accessed from constant memory and stored in the automatic variable `charge` (line 24). Therefore, the calculation of `energy0` through `energy3` can all just use the values stored in the registers.

Similarly, the x-coordinate of the atom is also read from constant memory and used to calculate `dx0` through `dx3` (lines 17-20). Altogether, this kernel eliminates three accesses to constant memory for the y coordinate of its atom, three accesses for the x coordinate of its atom, three accesses for the charge of the atom, three floating point subtraction operations, five floating-point multiply operations, and nine floating point add operations when processing an atom for four grid points. A quick inspec-

tion of the kernel code in Fig. 21.8 shows that each iteration of the loop performs four constant memory accesses, three floating-point subtractions, eleven floating point additions, six floating-point multiplications, and four floating-point divisions for four grid points.

The reader should also verify that the version of DCS kernel in Fig. 21.6 performs 16 constant memory accesses, 12 floating point subtractions, 12 floating point additions, 12 floating point multiplications, and 12 floating-point divisions – a total of 48 floating-point operations for the same four grid points. Going from Fig. 21.6 to Fig. 21.8, there is a total reduction from 16 constant memory accesses and 48 operations down to 4 constant memory accesses and 24 floating-point operations, a sizable reduction. One can expect a sizable improvement in both the execution time and the energy consumption of the kernel.

The cost of the optimization is that more registers are used by each thread. This can potentially reduce occupancy. However, since the number of registers used by each thread stays within the allowed limit, it does not limit the occupancy in this case.

21.4 Memory coalescing

While the performance of the DCS kernel in Fig. 21.8 is quite high, a quick profiling run reveals that the threads perform memory writes inefficiently. In lines 30-33, each thread writes four neighboring grid points. Unfortunately, the write pattern of adjacent threads in each warp will result in un-coalesced global memory writes. There are two problems that cause the un-coalesced write pattern in the kernel. First, each thread calculates four adjacent neighboring grid points. Thus, for each statement that writes to the `energygrid[]` array, the threads in a warp are not accessing adjacent locations. Note that two adjacent threads access memory locations that are four elements apart. Thus, the 32 locations to be written by all the threads in a warp are spread out, with three elements in between the loaded/written locations. This problem can be solved by assigning adjacent grid points to adjacent threads in each block. We first assign `blockDim.x` consecutive grid points in the x dimension to the threads. We then assign the next `blockDim.x` consecutive grid points to the same threads. We repeat the assignment until each thread has the number of grid points desired. This assignment is illustrated in Fig. 21.9.

The kernel code with coarsened thread granularity and coalescing-aware assignment of grid points to threads is shown in Fig. 21.10. Note that the x-coordinates used to calculate the atom-to-grid-point distances for a thread's assigned grid points are offset by `blockDim.x*gridspacing`. This reflects the fact that the x-coordinates of the four grid points assigned to a thread are `blockDim.x` grid points away from each other. Also, after the end of the loop, the indices of the memory writes to the `energygrid` array are `blockDim.x` away from each other as well. Hence, all writes to the `energygrid` array are coalesced and the performance of the kernel is better than that of Fig. 21.8.

For a thread coarsening factor of four, one can also use vector stores to achieve write coalescing in the original thread-to-grid-point assignment. We leave the implementation using vector stores as an exercise for the reader.

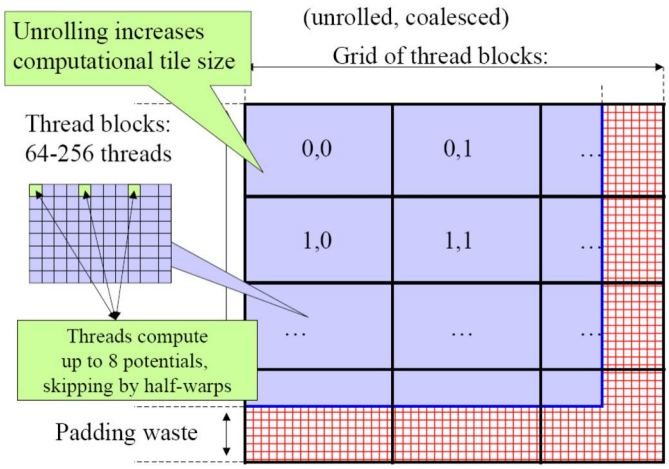


FIGURE 21.9

Organizing threads and memory layout for coalesced writes.

```
01  __constant__ float atoms[CHUNK_SIZE*4];
02  #define COARSEN_FACTOR 4
03  void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing,
04                          float z, int numatoms) {
05     int i = blockIdx.x * blockDim.x*COARSEN_FACTOR + threadIdx.x;
06     int j = blockIdx.y * blockDim.y + threadIdx.y;
07     int atomarrdim = numatoms * 4;
08     int k = z / gridspacing;
09     float y = gridspacing * (float) j;
10     float x = gridspacing * (float) i;
11     float energy0 = 0.0f;
12     float energy1 = 0.0f;
13     float energy2 = 0.0f;
14     float energy3 = 0.0f;
15     // calculate potential contribution from all atoms
16     for (int n=0; n<atomarrdim; n+=4) {
17        float dx0 = x - atoms[n  ];
18        float dx1 = dx0 +   blockDim.x * gridspacing;
19        float dx2 = dx0 + 2*blockDim.x * gridspacing;
20        float dx3 = dx0 + 3*blockDim.x * gridspacing;
21        float dy = y - atoms[n+1];
22        float dz = z - atoms[n+2];
23        float dysqdzsq = dy*dy + dz*dz;
24        float charge = atoms[n+3];
25        energy0 += charge / sqrtf(dx0*dx0 + dysqdzsq);
26        energy1 += charge / sqrtf(dx1*dx1 + dysqdzsq);
27        energy2 += charge / sqrtf(dx2*dx2 + dysqdzsq);
28        energy3 += charge / sqrtf(dx3*dx3 + dysqdzsq);
29     }
30     energygrid[grid.x*grid.y*k + grid.x*j + i              ] += energy0;
31     energygrid[grid.x*grid.y*k + grid.x*j + i +   blockDim.x] += energy1;
32     energygrid[grid.x*grid.y*k + grid.x*j + i + 2*blockDim.x] += energy2;
33     energygrid[grid.x*grid.y*k + grid.x*j + i + 3*blockDim.x] += energy3;
34  }
```

FIGURE 21.10

DCS kernel with thread coarsening and memory coalescing.

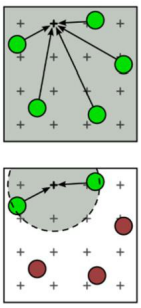


FIGURE 21.11

Cutoff summation vs. direct summation.

21.5 Cutoff binning for data size scalability

There are often multiple algorithms to solve a given problem. Some algorithms require fewer steps of computation than others, some expose a higher degree of parallel execution, some have better numerical stability than others, and some consume less memory bandwidth than others. Unfortunately, there is often not a single algorithm that is better than others in all the four aspects. Given a problem and a decomposition strategy, a parallel programmer often needs to select an algorithm that achieves the best compromise for a given hardware system.

In general, alternative algorithms for solving the same problem should reach the same solution. Under this requirement, one can optimize the computation for better efficiency and/or more parallelism. In some applications, one can often come up with even more aggressive algorithm strategies if the problem can be solved with slight variation in the final solution. An important algorithm strategy, referred to as cutoff summation, can significantly improve the execution efficiency of grid algorithms like the electrostatic potential calculation by sacrificing a small amount of accuracy. This is based on the observation that many grid calculation problems are based on physical laws where numerical contributions from particles or samples that are far away from a grid point can be collectively treated with an implicit method at much lower computational complexity.

The cut-off summation strategy is illustrated for the electrostatic potential calculation in Fig. 21.11. Fig. 21.11(a) shows the direct summation algorithms discussed in the previous sections of this chapter. Each grid point receives contributions from all atoms. While this is a very parallel approach and achieves excellent speedup, it does not scale well to very large energy-grid systems where the number of atoms increases proportional to the volume of the system. The amount of computation increases with the square of the volume. For large volume systems, such increase makes the computation excessively long even for massively parallel devices.

We know that each grid point needs to receive accurate contributions from atoms that are close to it. However, the atoms that are far away from a grid point have a

tiny contribution to the energy value at the grid point because the contribution is inversely proportional to the distance. Fig. 21.11(b) illustrates this observation with a circle drawn around a grid point. The contributions to the grid point energy from atoms outside the circle (dark atoms) are small and can be handled with a separate implicit method. If we can devise an algorithm where each grid point only receives contributions from atoms within a fixed radius of its coordinate, the computational complexity of the algorithm would be reduced to be linearly proportional to the volume of the system. This would make the computation time of the algorithm linearly proportional to the volume of the system. Such algorithms have been used extensively in sequential computation.

In sequential computing, a simple cutoff algorithm handles one atom at a time. For each atom, the algorithm iterates through the grid points that fall within a radius of the atom's coordinate. This is a straightforward procedure since the grid points are in an array that can be easily indexed as a function of their coordinates. A C implementation of the cut-off algorithm can be derived with minor modifications to the C implementation of DCS in Fig. 21.4 by restricting the range of `i` and `j` of the inner loops to the energy grid points that fall into the radius. However, this simple procedure does not carry easily to parallel execution. The reason is what we discussed in Section 21.2: the atom-centric parallelization does not work well due to it scatter memory update behavior.

Therefore, we need to find a cutoff binning algorithm based on the grid-centric decomposition: each thread calculates the energy value at one grid point. Fortunately, there is a well-known approach to adapting direct summation algorithm, such as the one in Fig. 21.10, into a cutoff binning algorithm. Rodrigues et al. present such an algorithm for the electrostatic potential problem [3].

The key idea of the algorithm is to first sort the input atoms into bins according to their coordinates. Each bin corresponds to a box in the energy grid space and it contains all atoms whose coordinates fall into the box. These bins are implemented as multi-dimensional arrays: the x, y, and z dimensions as well as the fourth dimension that is a vector of atoms in the bin.

We define a "neighborhood" of bins for a grid point to be the collection of bins that contain all the atoms that can contribute to the energy value of the grid point. Fig. 21.12 shows an example of the neighborhood bins for a grid point. Note that nine bins around the grid point overlap with the circle of cut-off distance. For correct cut-off summation, we need to make sure that all atoms in these nine bins are considered for contribution to the grid point. Note that some of the atoms in the neighborhood bins may not fall into the radius. Therefore, when processing an atom from one of the neighborhood bins, all threads need to check if the atom falls into its radius. This can cause some control divergence among threads in a warp.

Although Fig. 21.12 shows only one layer (2D) of bins that immediately surround the bin containing a grid point as its neighborhood, a real algorithm will typically have multiple 3D bins in a grid point's neighborhood. In this approach, all threads iterate through their own neighborhood. They use their block and thread indices to identify the coordinates of their assigned grid point and use these coordinates to identify the appropriate bins to examine. One can think of the neighborhood bins as a

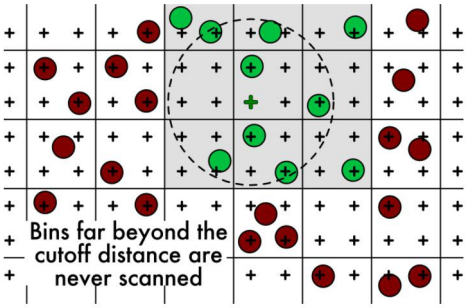


FIGURE 21.12

Neighborhood bins for a grid point.

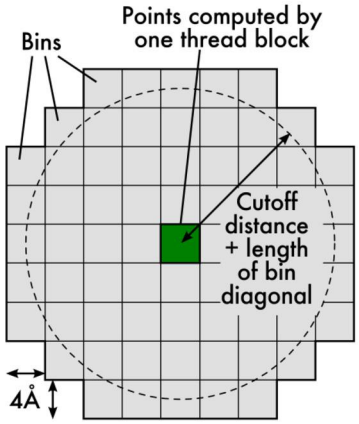


FIGURE 21.13

Identifying the neighborhood bins for all grid points processed by a block.

stencil in the energy grid space. However, the calculations involved in determining the neighborhood bins given a cut-off radius can be a complex geometric problem whose solution can be time consuming. Therefore, the neighborhood bins are usually defined for all threads in a block and are prepared before launching the grid.

Fig. 21.13 shows the per-block design of neighborhood bins. Based on the block dimensions and the grid spacing, one can calculate the area (volume in 3D) of the energy grid space covered by each block. We show the areas covered by the blocks as squares in Fig. 21.13. For simplicity, we assume that each of these areas is also covered by a bin. That is, all atoms that fall into the same square will be collected into the same bin. For example, if the grid spacing is 0.5 Å and the blocks are $8 \times 8 \times 8$, each block would cover 4 Å$\times$4 Å$\times$4 Å cube in the energy grid space. If we assume a typical cut-off distance of 12 Å for molecular-level force calculation, we need to identify all bins that can potentially be fully or partially covered by any of the 512

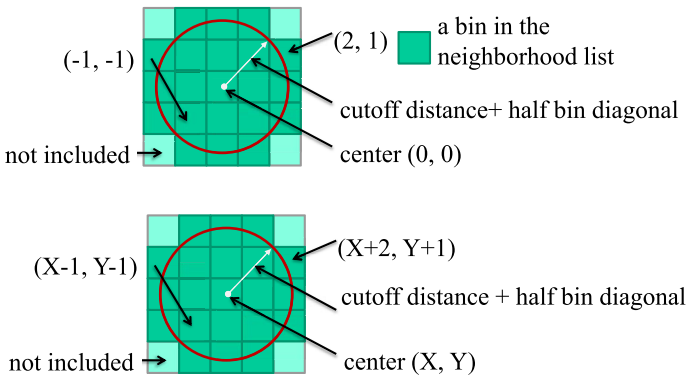


FIGURE 21.14

Neighborhood list using relative offsets.

circles each of which is centered at one of the grid points covered by one of the threads in the block. We can also use a conservative approximation by drawing a super circle that is centered at the center of the bin with a radius that is the cut-off distance plus half of the bin diagonal. The rationale is that this super circle would cover all the circles that are centered at the corners of the bin. We can simply create a list the relative positions of the bins that are fully or partially covered by the super circle.

Fig. 21.14 shows a small example of identifying neighborhood bins. We see that for each bin, there are 9 bins that are fully covered by the super circle and 12 bins that are partially covered. We can generate a list of $9 + 12 = 21$ neighborhood bins that each thread in a block needs to examine for atoms that are within the cut-off distance of the grid point covered by the thread. These bins are expressed as the relative offset of the bin coordinates. For example, the 9 bins that are fully covered by the super circle can be expressed as the list $(-1, -1)$, $(0, -1)$, (1,-1), $(-1,0)$, (0,0), (1,0), $(-1,1)$, (0, 1), (1,1), as illustrated in the top portion of Fig. 21.14. The list would be supplied to the kernel, most likely as a constant memory array. During kernel execution, all threads in a block would iterate through the neighborhood list. For each neighborhood bin, the threads apply the offsets to the coordinates of the bin covered by the block and derive the coordinate of the neighbor bin, as illustrated in the bottom portion of Fig. 21.14. They collaborate to load the atoms in the bin into the shared memory and then each thread individually checks if these atoms fall within the cut-off distance of its assigned grid point. Each thread could make different decisions on including or excluding each atom for contributing to the energy value at its assigned grid point.

The improvement in computational complexity in a cut-off binning algorithm mainly comes from the fact that each thread now examines a much smaller subset of atoms, defined by the neighborhood bins, in a large grid system. This, however, makes constant memory much less attractive for holding the atoms. Since thread

blocks will be accessing different neighborhoods, the limited sized constant memory will unlikely be able to hold all the atoms that are needed by all active thread blocks. This motivates the use of global memory to hold a much larger set of atoms. To mitigate the bandwidth consumption, threads in a block collaborate in loading the atom information in each common neighborhood bin into the shared memory. All threads then examine the atoms out of shared memory.

One subtle issue with binning is that bins may end up with different numbers of atoms. Since the atoms are statistically distributed in the grid system, some bins may have lots of atoms and some bins may end up with no atom at all. To guarantee memory coalescing, it is important that all bins are of the same size and aligned at appropriate coalescing boundaries. This would require us to fill many bins with dummy atoms whose electrical charge is 0, which causes two negative effects. First, the dummy atoms still occupy global memory and shared memory storage. They also consume data transfer bandwidth to the device. Second, the dummy atoms extend the execution time of the thread blocks whose bins have few real atoms.

A good approach is to set the bin size at a reasonable level that covers the number of atoms in the vast majority of the bins, typically much smaller than the largest possible number of atoms in a bin. The binning process maintains an overflow list. When processing an atom, if the atom's home bin is full, the atom is added to the overflow list instead. After the device completes a kernel, the resulting grid point energy values are transferred back to the host. The host executes a sequential cutoff algorithm on the atoms in the overflow list to complete the missing contributions from these overflow atoms.

As long as the overflow atoms account for only a small percentage (e.g., less than 3%) of the atoms, the additional sequential processing time of the overflow atoms is typically shorter than that of the device execution time. One can also design the kernel so that each kernel invocation calculates the energy values for a sub-volume of grid points. After each kernel completes, the host launches the next kernel and processes the overflow atoms for the completed kernel. Thus, the host processes the overflow atoms while the device executes the next kernel. This approach can hide most if not all the delays in processing overflow atoms since it is done in parallel with the execution of the next kernel.

21.6 Summary

This chapter presents a series of decisions and tradeoffs in parallelizing the calculation of electrostatic potential energy in a regularly spaced energy grid. We show that parallelizing a highly optimized sequential C implementation of the DCS method leads to a slow scatter kernel that requires heavy use of atomic operations. We then show that one can parallelize a less optimized sequential C code into a gather kernel that has much higher parallel execution speed. We also show that through thread coarsening, we can reclaim much of the efficiency of the optimized sequential execution. We further demonstrate that by carefully choosing the grid points to fold into each thread, the kernel can have completely coalesced memory write patterns.

While DCS is a highly accurate method for calculating the electrostatic potential energy map of a molecular system, it is not a scalable method. The number of operations performed by the method grows proportionally with the number of atoms and the number of grid points. When we increase the physical volume of the molecular system to be simulated, we should expect both the number of grid points and the number of atoms to increase proportionally to the physical size. As a result, the number of operations to be performed is approximately proportional to the square of the physical volume. That is, the number of operations to be performed will grow quadratically with the volume of the system being simulated. This makes the use of the DCS method for not suitable for simulating realistic biological systems. We show that by slightly reducing the accuracy, a cut-off summation method implemented with the binning technique can dramatically improve the complexity of the computation while retaining a high level of parallelism.

21.7 Exercises

1. Complete the host code for configuring the grid and calling the kernel in Fig. 21.6 with all the execution configuration parameters.
2. Compare the number of operations (memory loads, floating-point arithmetic, branches) executed in each iteration of the kernel in Fig. 21.6 to that of the kernel in Fig. 21.8 with a coarsening factor of eight. Keep in mind that each thread of the latter corresponds to eight threads of the former.
3. Give two potential disadvantages associated with increasing the amount of work done in each CUDA thread, as shown in Section 21.3.
4. Revised the kernel in Fig. 21.8 so that vector stores can be used to improve memory coalescing when updating the energy grid points.
5. Use Fig. 21.13 to explain how control divergence can arise when threads in a block process a bin in the neighborhood list.

References

[1] W. Humphrey, A. Dalke, K. Schulten, Vmd: visual molecular dynamics, Journal of Molecular Graphics 14 (1) (1996) 33–38.

[2] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (16) (2007) 2618–2640.

[3] C.I. Rodrigues, D.J. Hardy, J.E. Stone, K. Schulten, W.-M.W. Hwu, Gpu acceleration of cutoff pair potentials for molecular modeling applications, in: Proceedings of the 5th Conference on Computing Frontiers, 2008, pp. 273–282.

CHAPTER

22 Algorithm selection, problem decomposition, and problem formulation

CONTENTS

22.1 Algorithm selection 530
22.2 Problem decomposition 532
22.3 Application level considerations – Amdahl's law 537
22.4 Problem formulation 538
22.5 Batching: latency vs. throughput 539
22.6 Summary 540
References 540

We have so far concentrated on practical knowledge in parallel programming, which consists of CUDA programming features, the GPU architecture, performance optimization techniques, parallel patterns, and application case studies. In this chapter, we switch our discussions to concepts that are more abstract. We first generalize parallel programming into a thinking process of designing/selecting parallel algorithms and decomposing a domain problem into well-defined and coordinated parallel work units that can each be efficiently performed by the selected algorithms. With good algorithm selection and problem decomposition, the programmer can achieve an appropriate compromise between parallelism, work efficiency, and resource consumption.

A strong combination of domain knowledge and parallel thinking skills is often needed for creating successful computational solutions to challenging domain problems. With strong parallel thinking skills, a parallel programmer does not need to be confined to the role of solving domain problems handed down by domain scientists. Rather, the programmer can collaborate with domain scientists to analyze and transform the structure of a domain problem: which parts are inherently serial, which parts are amenable to high-performance parallel execution, and the domain-specific tradeoffs involved in moving parts from the former category to the latter. This chapter gives the reader more insight into parallel programming and parallel thinking in general.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00033-0

Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

22.1 Algorithm selection

An algorithm is a step-by-step procedure where each step is precisely stated and can be carried out by a computer. An algorithm must exhibit three essential properties: definiteness, effective computability, and finiteness. Definiteness refers to the notion that each step is precisely stated; there is no room for ambiguity as to what is to be done. Effective computability refers to the fact that each step can be carried out by a computer. Finiteness means that the algorithm must be guaranteed to terminate.

Given a problem, one can typically come up with multiple algorithms to solve the problem. Some require fewer steps of computation than others (i.e., have lower algorithmic complexity), some expose a higher degree of parallel execution than others, some are more generally applicable than others, and some have better accuracy or numerical stability than others. Unfortunately, there is often not a single algorithm that is better than others in all these aspects. A parallel programmer often needs to select an algorithm that achieves the best compromise for a given hardware system.

We have seen examples of assessing the trade-offs between different algorithms for a computation in several chapters throughout this book. For the prefix sum computation in Chapter 11, we compared two different algorithms for performing parallel prefix sum, namely the Kogge-Stone algorithm and the Brent-Kung algorithm. Our analysis showed that the Brent-Kung algorithm has lower algorithmic complexity. It requires fewer operations to perform the same computation, which makes it more work efficient. However, we also showed that the Kogge-Stone algorithm exposes more parallelism than the Brent-Kung algorithm, allowing it to complete in fewer iterations. This trade-off between algorithmic complexity and the amount of parallelism exposed by an algorithm is a classic trade-off that is encountered by parallel programmers. The best algorithm usually depends on the characteristics of the parallel hardware being targeted, as well as the extent to which the higher complexity of an algorithm can be mitigated with hybrid approaches that combine two parallel algorithms or combine a parallel algorithm with a lower-complexity sequential algorithm via thread coarsening.

For the sorting pattern in Chapter 14, we compared three different algorithms for performing parallel sorting: odd-even sort, merge sort, and radix sort. Odd-even sort has the highest algorithmic complexity, but is the most trivial to parallelize. Radix sort can achieve lower algorithmic complexity than merge sort because it is a non-comparison sorting algorithm. It is also highly amenable to parallelization. However, radix sort is not generally applicable because it can only be used with certain types of keys. Merge sort, which is a comparison-based sort, is more generally applicable than radix sort and can be used with any type of key that has a well-define comparison operator. The trade-off between generality and parallel execution efficiency is yet another trade-off that parallel programmers encounter when selecting a parallel algorithm.

For the electrostatic potential map computation in Chapter 21, we compared two different algorithms for performing the computation, namely Direct Coulomb Summation (DCS) and cutoff summation. These two approaches both expose ample

parallelism. However, they present the classic trade-off between algorithmic complexity and accuracy. In Chapter 21, we showed that the cutoff summation algorithm can significantly improve the execution efficiency of grid computation by sacrificing a small amount of accuracy. The challenge addressed by this technique is that the amount of computation performed by fully accurate methods like DCS increases with the square of the volume. For large volume systems, such increase makes the computation excessively long even for massively parallel devices. The cut-off summation method is based on the observation that many grid calculation problems are based on physical laws where numerical contributions from particles or samples that are far away from a grid point are tiny and can be collectively treated with an implicit method at much lower algorithmic complexity. Although this trade-off between algorithmic complexity and accuracy is not unique to parallel programming and is also encountered with sequential implementations, it often presents additional challenges to parallel programmers. We discuss an example of these additional challenges for the cutoff summation algorithm in the rest of this section.

In sequential computing, a simple cutoff algorithm handles one atom at a time. For each atom, the algorithm iterates through the grid points that fall within a radius of the atom's coordinate. This is a straightforward procedure since the grid points are in an array that can be easily indexed as a function of their coordinates. However, this simple procedure does not carry easily to parallel execution. The reason is that the atom-centric decomposition does not work well due to it scatter memory access behavior. Therefore, we need to use a cutoff binning algorithm based on the grid-centric decomposition: each thread calculates the energy value at one grid point. The key idea of the algorithm is to first sort the input atoms into bins according to their coordinates. Each bin corresponds to a box in the grid space and it contains all atoms whose coordinates fall into the box. We define a "neighborhood" of bins for a grid point to be the collection of bins that contain all the atoms that can contribute to the energy value of a grid point according to the cut-off distance threshold. We describe an efficient way of managing neighborhood bins for all grid points in Chapter 21. In this algorithm, all blocks iterate through their own neighborhood bins while all threads in a block scan through the atoms in their combined neighborhood bins together but make individual decisions on whether each atom falls within its cut-off radius.

One subtle issue with binning is that bins may end up with different numbers of atoms. Some bins may have many atoms and some bins may have no atom at all. Declaring the size (capacity) of the bins to accommodate the largest possible number of atoms can be prohibitively expensive. A well-known solution is to set the bin size at a reasonable level, typically much smaller than the largest possible number of atoms in a bin. The binning process maintains an overflow list. When sorting atoms into bins, if an atom's home bin is full, the atom is added to the overflow list instead. After the device completes executing a cut-off summation kernel for calculating an electrostatic potential map, atoms in the overflow list need to be processed to complete the missing contributions.

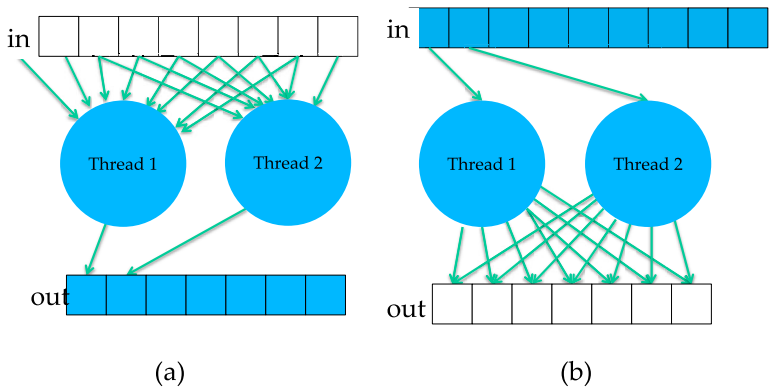


FIGURE 22.1

Problem decomposition strategies: (a) output-centric and (b) input-centric.

22.2 Problem decomposition

After selecting an appropriate algorithm, for a problem to be solved with parallel computing, the problem must be formulated in such a way that it can be decomposed into sub-problems that can be safely solved at the same time. Under such formulation and decomposition, the programmer writes code and organizes data to solve these sub-problems concurrently. Finding parallelism in large computational problems is often conceptually simple, but can be challenging in practice. The key is to identify the units of work to be performed, or the subproblem to be solved, by each thread or collection so that the inherent parallelism in the problem is well utilized.

The two most common strategies for decomposing a problem for parallel execution are output-centric and input-centric decompositions. These decomposition strategies are illustrated in Fig. 22.1. As the names imply, with output-centric decomposition, solving each sub-problem produces the result for one or more output elements. An implementation based on output-centric decomposition assigns a thread or a collection of threads to solve one or more sub-problems and produce one or more output elements. The solution of each sub-problem may require all input elements (e.g., grid-centric DCS in Chapter 21) or a subset of the input elements (e.g., matrix multiplication in Chapter 5 and grid-centric cut-off-summation with binning in Chapter 21).

In contrast, with input-centric decomposition, solving each sub-problem derives the contribution of one or more input elements to the final output. An implementation based on input-centric decomposition assigns a thread or a collection of threads to process one or more input elements and apply their contributions to the output elements. The solution of each sub-problem may update one output element (e.g., histogram in Chapter 9), a subset of output elements (e.g., hypothetically, atom-centric cut-off summation in Chapter 21) or all output elements (e.g., atom-centric DCS in Chapter 21).

For some problems, input-centric and output-centric decomposition may lead to the same sub-problems. For example, in odd-even sort in Chapter 14, each thread is responsible for reading a pair of input elements to produce the result values for the corresponding pair of output elements. In this case, since the input elements and output elements have simple direct relations, it does not matter whether we view the decomposition as input-centric or output-centric.

While both decomposition strategies lead to the same execution results, they can exhibit very different performance in a given hardware system. The output-centric decomposition usually exhibits the gather memory access behavior, where each thread gathers or collects the effect of input values into an output value. Fig. 22.1(a) illustrates the gather access behavior. Gather-based access patterns are usually more desirable in CUDA devices because the threads can accumulate their results in their private registers. Also, multiple threads can share input values, and can effectively use the GPU caches or shared memory to conserve global memory bandwidth.

The input-centric decomposition, on the other hand, usually exhibits the scatter memory access behavior, where each thread scatters or distributes the effect of an input value into the output values. The scatter behavior is illustrated in Fig. 22.1(b). Scatter-based access patterns are usually undesirable in CUDA devices because multiple threads can update the same output value at the same time. The output values must be stored in a memory that can be written by all the threads involved. Atomic operations must be used to prevent race conditions and loss of values during simultaneous writes to an output value by multiple threads. These atomic operations are significantly slower than the register accesses used in the output-centric decomposition.

However, aside from gather versus scatter access patterns, other considerations may go into deciding whether an output-centric or input-centric decomposition (or another decomposition) is more suitable for a particular application. These considerations include: the amount of parallelism exposed by the decomposition, the ease of identifying which input data contributes to which output data, the load balance induced by the decomposition, and others depending on the application.

The comparison between input-centric and output-centric decompositions was most evident in Chapter 21 where the two strategies were both implemented and explicitly compared. In particular, the atom-centric decomposition was an input-centric one, and the grid-centric decomposition was an output-centric one. However, aside from Chapter 21, problem decomposition has been an implicit design decision that was made throughout many computations discussed in this book. We revisit these computations to highlight the problem decomposition chosen in each case and why it is advantageous over alternative decompositions.

The image processing (Chapter 3), matrix multiplication (Chapter 3, Chapter 5, and Chapter 15), convolution (Chapter 7), and stencil (Chapter 8) computations were all parallelized using an output-centric decomposition. That is, the threads in these computations are assigned to output elements (image pixels, matrix entries, or grid points) and iterate over the input elements that contribute to them. Alternatively, an input-centric decomposition would assign threads to the input elements and have

each thread iterate over the output elements that its input element contributes to and make an update using atomic operations. The clear advantage of the output-centric decomposition over the input-centric decomposition in these cases is avoiding atomic operations by using a gather access pattern instead of a scatter access pattern. On the other hand, none of the other considerations make the input-centric decomposition favorable. There are enough output elements to expose a high degree of parallelism; it is straightforward to identify which input elements are needed by each output element; and all output elements require the same amount of work to be computed so there is no load imbalance.

The histogram computation (Chapter 9) was parallelized using an input-centric decomposition. That is, each thread is assigned to an input element (or chunk of elements) and updates the output bins based on the input value(s). Since multiple threads may update the same output bin, atomic operations are needed (scatter access pattern). Alternatively, an output-centric decomposition for the histogram computation would assign threads to output bins and have each thread search for the inputs that map to the bin and update the bin accordingly. This decomposition would eliminate the atomic operations because each bin would be updated by a single thread (gather access pattern). However, the output-centric decomposition would create many problems. First, the number of output bins is typically much smaller than the number of input values so parallelism is substantially reduced. Second, a thread cannot know which input values map to its output bin without inspecting those input values, so each thread will need to iterate over every input value which is not work efficient. Third, even if threads had some way of quickly identifying which input values map to their output bins, each bin will have a different number of input values mapping to it, which would create load imbalance across threads. All these considerations make the input-centric decomposition more favorable for the histogram computation.

The reduction (Chapter 10) and scan (Chapter 12) patterns both involve multiple iterations. If one considers problem decomposition on an iteration-by-iteration basis, both reduction and scan are parallelized using an output-centric decomposition. Each thread is responsible for updating one element of the iteration's output array by gathering and adding two elements from the iteration's input array. An input-centric decomposition of a reduction or scan iteration would assign each thread to an element of the iteration's input array, and that thread would scatter that element's value to the elements that need it in the iteration's output array. Since multiple input elements contribute to the same output element, such a decomposition would require atomic operations to avoid race conditions which would significantly impede performance.

The unstable and stable filter patterns (Chapter 12) were both parallelized using an input-centric decomposition. Each thread is assigned to an input element and is responsible for identifying if that input element should be retained, and if yes, where it should be placed in the output array. An efficient output-centric decomposition of the unstable or stable filter patterns would be quite difficult. It would entail assigning a thread to an element of the output array and having that thread identify which input element should be placed there. However, doing so would require the thread to inspect a large number of other elements which would be inefficient and cause redundant work across threads. The input-centric decomposition is naturally a better fit. It

is noteworthy that while the overall decomposition of the stable filter computation is input-centric, the computation includes a scan operation which itself is output-centric. It is not unusual for complex computations to include multiple operations that are parallelized with different decomposition strategies.

The merge operation (Chapter 13) was parallelized using an output-centric decomposition. That is, each thread is assigned to a chunk of elements in the output array and performs a binary search to find the corresponding input element(s) to be copied into these output elements. While many other computations favor the output-centric decomposition to avoid atomic operations, this consideration is not relevant for the merge operation. Every output element in the merge operation receives contribution from only one input element, so an input-centric merge operation would not need to use atomic operations. The main reason for merge to prefer an output-centric merge over an input-centric merge is to achieve load balance when thread coarsening is applied. In an input-centric merge, we would partition one of the input arrays across the threads and have each thread find the corresponding elements in the other input array and merge them into the output array. However, different threads could end up merging very different numbers of input elements, resulting in control divergence and load imbalance across threads. For this reason, the output-centric decomposition for parallel merge is preferred.

Sorting (Chapter 14) uses different problem decompositions depending on the algorithm used. For odd-even sort, the input-centric and output-centric decompositions are equivalent. In every iteration of odd-even sort, each thread is assigned to a pair of elements and decides whether or not to swap it. In other words, each thread is assigned to one input pair and processes it to produce one output pair. There is no scattering or gathering involved. For merge sort, each level consists primarily of a merge operation, which uses an output-centric decomposition as we have seen. For radix sort, each iteration uses an input-centric decomposition where each thread is assigned to an input element and identifies its new position in the output array. Recall that the partition operation used in radix sort is a generalization of the stable filter pattern, which also uses an input-centric decomposition as we have seen.

The wavefront pattern (Chapter 16) is parallelized using an output-centric decomposition. To compute each (output) element of the dynamic programming matrix, the thread assigned to that element gathers the values of multiple previously computed (input) elements. Alternatively, an input-centric decomposition of the wavefront pattern would entail having each thread scatter the value of a previously computed (input) element to all the succeeding (output) elements that depend on that value. Since multiple elements of the dynamic programming matrix can contribute to the same succeeding element, such a decomposition would require atomic operations making it less favorable.

For sparse matrix computations (Chapter 17), the SpMV/CSR, SpMV/ELL, and SpMV/JDS kernels all use output-centric decomposition where each thread is assigned to an output vector element and iterates over the corresponding input row of non-zeros to gather their contribution. On the other hand, the SpMV/COO kernel uses input-centric decomposition where each thread is assigned to a non-zero element in the input matrix and atomically updates the corresponding element in the

output vector, resulting in a scatter access pattern. Compared to the output-centric kernels, the input-centric SpMV/COO kernel has the disadvantage of performing atomic operations, but it has other advantages such as extracting more parallelism and better load balance. An output-centric decomposition for SpMV/COO would require threads to search through all input elements to find the ones that contribute to their output vector element, which would drastically increase the computation complexity. The SpMV/CSC kernel also uses input-centric decomposition, where each thread is assigned to an element of the input vector and iterates through a column of the input matrix, scattering its contribution to the relevant output vector elements using atomic operations. The CSC format does not provide convenient access for an output-centric decomposition to iterate through all non-zero elements of a row. Finding all input elements that contribute to an output element would require each thread to scan through all columns to check for any input element from the row that contributes to its assigned output element. Ultimately, the best decomposition for this computation depends on the input dataset. Moreover, the hybrid ELL-COO format showcases an example of where a hybrid output-centric and input-centric decomposition may be beneficial.

For graph traversal (Chapter 18), the vertex-centric push implementation and the edge-centric implementation use input-centric decomposition. That is, threads are assigned to vertices or edges, and update the levels of neighboring vertices. In contrast, the vertex-centric pull implementation uses output-centric decomposition where each thread only updates the level of the vertex that it is assigned to. The trade-off between the scatter and gather access patterns is not a concern when selecting between different decompositions because the updates to the level values are idempotent and do not use atomic operations. However, the amount of parallelism extracted and the load balance achieved play an important role in deciding which decomposition is more favorable, and the best decomposition ultimately depends on the input dataset. The reader is encouraged to review Chapter 18 for a more detailed handling of this topic.

The electrostatic potential map calculation problem (Chapter 21) favors the output-centric decomposition because of the benefit over the gather pattern of the scatter pattern in avoiding atomic operations. The chapter already handles the distinction between the two decomposition strategies in depth. The computation involves the calculation of the contribution of many input atoms to the potential energy at a large number of output grid points. A realistic molecular system model typically involves at least hundreds of thousands of atoms and millions of energy grid points. The electrostatic charge information of each atom is used many times in calculating its contributions to the energy grid points. We show that the decomposition of the electrostatic potential map calculation problem can be atom-centric (i.e., input-centric) or grid-centric (i.e., output-centric). In an atom-centric decomposition, each thread is responsible for calculating the effect of one atom on all grid points. In contrast, a grid-centric decomposition uses each thread to calculate the effect of all atoms on a grid point. The grid-centric (i.e., output-centric) decomposition proved to be better because it uses the more favorable gather access pattern. The amount of parallelism exposed by both decompositions is sufficient. For the cutoff summation algorithm, the difficulty of mapping input data to output data in the grid-centric decomposition is overcome with the use of binning.

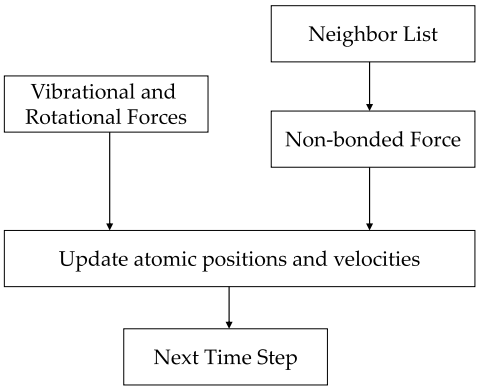


FIGURE 22.2

Major tasks of a molecular dynamics application.

22.3 Application level considerations – Amdahl's law

A real application often consists of multiple modules that work together. Fig. 22.2 shows an overview of the major modules of a molecular dynamics application. For each atom in the system, the application needs to calculate the various forms of forces – e.g., vibrational, rotational, and non-bonded – that are exerted on the atom. Each form of forces is calculated with a different method. At the high level, a programmer needs to decide how the work is organized. Note that the amount of work can vary dramatically between these modules. The non-bonded force calculation typically involves interactions among many atoms and incurs much more calculations than the vibrational and rotational forces. Therefore, these modules tend to be realized as separate passes over the force data structure.

The programmer needs to decide if each pass is worth implementing for a CUDA device. For example, the programmer may decide that the vibrational and rotational force calculations do not involve sufficient amount of work to warrant execution on a GPU device. Such a decision would lead to a CUDA program that launches a kernel that calculates non-bonding force fields for all the grid points while continuing to calculate the vibrational and rotational forces for the grid points on the host. The module that updates atom positions and velocities may also run on the host. It first combines the vibrational and rotational forces from the host and the non-bonding forces from the device. It then uses the combined forces to calculate the new atomic positions and velocities.

The portion of work done by the GPU device ultimately decides the application-level speedup achieved by parallelization. For example, assume that the non-bonding force calculation accounts for 95% of the original sequential execution time and it is accelerated by 100× using a GPU. Further assume that the rest of the application remains on the host and receives no speedup. The application-level speed up is $1/(5\% + 95\%/100) = 1/(5\% + 0.95\%) = 1/(5.95\%) = 17\times$. In the case where the execution

of the host and the CUDA device can be overlapped, the execution time of the parallel section is completely hidden in the host execution time. The application-level speedup would be $1/(5\%) = 20\times$. This is a demonstration of Amdahl's Law that we saw in Chapter 1: the application speedup due to parallel computing is limited by the sequential portion of the application. In this case, even though the sequential portion of the application is quite small (5%), it limits the application-level speedup to $20\times$ despite the non-bonding force calculation having a speedup of $100\times$ and being completely hidden in the shadow of the host execution of the sequential portion. This example illustrates a major challenge in accelerating large applications: the accumulated execution time of small activities that are not worth parallel execution on a CUDA device can become a limiting factor in the speedup seen by the end users.

Amdahl's Law often motivates task-level parallelization. Although some of these smaller activities do not warrant fine-grained massive parallel execution, it may be desirable to execute some of these activities in parallel with each other when the data set is large enough. This could be achieved by using a multi-core host to execute such tasks in parallel. Alternatively, we could try to simultaneously execute multiple small kernels, each corresponding to one task. Although each kernel may not be able to make good use of the GPU and achieve significant speedup over CPU execution, executing them simultaneously on a GPU may overlap their execution with each other and achieve good speedup overall. CUDA devices support task parallelism with streams, which are discussed in Chapter 23.

22.4 Problem formulation

Problem formulation is arguably the most consequential aspect of parallel application development [1]. The "best" approach to parallelizing an application is not using generic numerical methods to map domain problems to computing steps that can be implemented using the skills you have learned from this book. Such approach often results in high computational complexity and/or limited parallelism. Rather, one often needs to rethink numerical methods to relax some of the constraints on the implementation.

The cutoff binning approach in Chapter 21 is a good example. The approach requires domain expertise to trade accuracy for dramatically reduced algorithmic complexity, and requires problem decomposition and optimization skills to use the grid-centric decomposition with the binning techniques designed by computer scientists. In this approach, there is the potential for the biggest performance advantage and fundamental new discoveries and capabilities. For example, one may be able to perform high-fidelity simulation of a biochemical system whose size is considered beyond reach with traditional methods. This approach is interdisciplinary and requires both computer science and domain insights, but the payoff is worth the effort.

22.5 Batching: latency vs. throughput

The parallel programming optimization techniques that we have discussed so far are designed to make the parallel program run faster given the same problem. For example, in Chapter 11, techniques such as double buffering, warp-level shuffling register tiling, and look-back help the code run faster, i.e., with shorter latency, when processing the same input. The process of making parallel code run faster for the same input data is often referred to as latency optimization. While it is desirable to make the parallel code run faster on the same input, it is not the only motivation for optimizing a parallel program. In general, there are at least two motivations for optimizing parallel programs.

The first and most intuitive motivation is to solve a given problem, i.e. processing a given input data set, in less time, which has been our focus in this book. For example, an investment firm may need to perform a financial portfolio risk analysis on all its portfolios during after-trading hours. Such an analysis may take 10 hours with a current parallel program. However, the portfolio management team may request that the analysis be completed in four hours in order to make more timely decisions. When the parallel program fails to meet the new latency requirement, optimization techniques can further enhance the speed and allow the analysis to finish within the time window. In Chapter 20, the use of flash attention technique reduces the transformer latency for a given input by eliminating redundant accesses to global memory.

The second, and somewhat less intuitive, motivation of using parallel computing is to improve the total latency for solving many problems. In our financial portfolio analysis example, the investment firm may decide to run multiple instances of the analysis with different parameters for statistical reasons. However, the current parallel program may not be able to run all the instances within the time window. Optimizing the current parallel program to reduce the total latency for solving multiple instances of the problem involves an additional tradeoff: latency for solving each individual problem vs. latency for solving all problem instances.

In Chapter 20, processing a query from a user can be viewed as an instance of the inference problem. When there are many users making queries, there are many instances of inference problems. Batching queries from different users allows these queries to be processed by the transformer layers together to increase the arithmetic intensity of the attention heads. For example, batching allows the `QKV` projection vector-matrix multiplication operations to be converted into matrix multiplications. Although a matrix multiplication takes more time to execute than a vector-matrix multiplication, it takes much shorter time than the sum of the execution time for all the vector-matrix multiplications in the batch. This is because reusing the original matrix elements in matrix multiplication eliminates redundant global memory accesses across vector-matrix multiplications and greatly improves arithmetic intensity.

That is, individual queries experience longer `QKV` projection latency, but the whole group of queries finish with shorter total latency. Thus, the throughput of `QKV` projection is higher with batching ($\frac{total\ number\ of\ queries}{latency\ for\ matrix\ multiplication}$) than without batching ($\frac{total\ number\ of\ queries}{total\ latency\ for\ all\ matrix-vector\ multiplications}$). Such tradeoff between latency for individual problem instances vs. the throughput of a batch of problem instances is an

important aspect of optimizing parallel programs. The reader should also recognize that the same tradeoff has been discussed in Chapter 1. The CPU design has shorter latency for processing each individual data element, whereas the GPU design has higher throughput for processing a large number of data elements.

There is a very large volume of literature on a wide range of algorithms, problem decompositions, problem formulation, and optimization strategies. It is beyond the scope of this book to have a comprehensive coverage of all the available concepts and techniques. We discuss a substantial set of techniques in each of the algorithm, problem decomposition, problem formulation, and optimization steps that have broad applicability. Although these techniques are demonstrated using CUDA implementations, they help the readers build up the foundation for parallel thinking in general. We believe that humans understand best when we learn from the bottom up. That is, we first learn concepts in the context of a particular programming model, which provides us with solid footing before we generalize our knowledge. An in-depth experience with the CUDA implementations also enables us to gain maturity, which helps us learn parallel programming and thinking concepts that may not even be pertinent to GPUs.

22.6 Summary

In summary, we have discussed the main steps of parallel programming and thinking, namely: problem formulation, algorithm selection, problem decomposition, and optimization. Given a computational problem, the programmer will typically have to select from a variety of algorithms. Some of these algorithms achieve different tradeoffs while maintaining the same numerical accuracy. Others involve sacrificing some level of accuracy to achieve much more scalable running times. For the selected algorithm, different choices of problem decomposition can result in different levels of interference between threads, level of exposed parallelism, load imbalance, and other performance considerations during parallel execution. Batching techniques can also be used to trade increased individual latency for increased throughput for groups of problem instances. Parallel thinking skills allow a parallel programmer to work around roadblocks and reach a good solution.

References

[1] J.M. Wing, Computational thinking, Communications of the ACM 49 (3) (2006) 33–35.

CHAPTER

Multi-GPU programming

And an introduction to MPI, NCCL, and NVSHMEM

23

With special contributions from Jiri Kraus and Isaac Gelado

CONTENTS

23.1 Stencil as a running example 542
23.2 Multi-GPU stencil with MPI 545
23.3 Overlapping computation and communication 553
23.4 Multi-GPU stencil with NCCL 563
23.5 Multi-GPU stencil with NVSHMEM 568
23.6 Summary 574
23.7 Exercises 576
References 576

So far, we have focused on programming a heterogeneous computing system with one host and one device. However, many applications require the aggregate computing power of multiple hosts and devices. Many clusters and data centers today have one or more hosts and one or more devices attached to each host. In this chapter, we look at three programming models for programming clusters with multiple GPUs: the Message Passing Interface (MPI) [1], the NVIDIA Collective Communications Library (NCCL) [2], and the NVIDIA OpenSHMEM Library (NVSHMEM) [3]. We only present the key concepts from each programming model that a programmer needs to understand in order to scale their heterogeneous applications to multiple GPUs and nodes in a cluster environment. We focus on domain partitioning, point-to-point communication, collective communication, and overlapping computation and communication in the context of scaling a CUDA stencil computation across multiple GPU nodes. The implementations presented are intended to illustrate the core principles behind each programming model and can each be further optimized within their respective programming models. Readers are encouraged to refer to tutorials and application case studies of each of the three programming models for a more extensive coverage of the full capabilities of each model once they master the concepts in this chapter.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00034-2
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

```
01    __global__ void jacobi_kernel(float* out, float* in, float* l2norm,
02        unsigned int ny, unsigned int nx) {
03        int y = blockIdx.y*blockDim.y + threadIdx.y + 1;
04        int x = blockIdx.x*blockDim.x + threadIdx.x + 1;
05        // Compute stencil and residue
06        float residue = 0.0f;
07        if(y < ny - 1 && x < nx - 1) {
08            float val = 0.25*(in[y*nx + (x + 1)]
09                            + in[y*nx + (x - 1)]
10                            + in[(y + 1)*nx + x]
11                            + in[(y - 1)*nx + x]);
12            out[y*nx + x] = val;
13            residue = val - in[y*nx + x];
14        }
15        // Reduce L2 norm
16        float blockL2norm = blockReduce(residue*residue);
17        if (threadIdx.y == 0 && threadIdx.x == 0) {
18            atomicAdd(l2norm, blockL2norm);
19        }
20    }
```

FIGURE 23.1

Modified stencil kernel for the Jacobi iterative method in 2D.

23.1 Stencil as a running example

Throughout this chapter, we use the stencil computation introduced in Chapter 8 as a running example. In particular, we use the Jacobi Iterative Method where, in each iteration or time step, the value of a point in a structured grid[1] is calculated as a weighted sum of its neighbors from the previous time step. The stencil kernel is called for multiple iterations until the grid point values stop changing significantly. To determine if the values have stopped changing, we calculate the L2 norm of the residual values, i.e., the differences between the old value and new value at each grid point.

The stencil kernel in Fig. 23.1 embodies two modifications [4] to those from Chapter 8. The first modification is that we compute a 2D stencil, as opposed to a 3D stencil, to simplify the illustrations and the code examples. The second modification is that the stencil kernel also computes the residual and the L2 norm. As shown in Fig. 23.1, each thread identifies the `x` and `y` coordinates of the grid point it is responsible for computing (lines 03-04). A boundary check makes sure that only threads computing grid points that are within bounds are active (line 07). Note that the points at $x = 0$, $x = nx - 1$, $y = 0$, and $y = ny - 1$ are skipped because they are either boundary points or halo points, as we will see later. The thread calculates the new value at its grid point as the average of the old values at the neighboring grid points (lines 08-12). The code so far should be familiar from the kernels in Chapter 8.

[1] Throughout this chapter, we use the word "grid" to refer to structured grid of data values in a modeled system as introduced in Section 8.1. To avoid confusion, we use the term "thread grid" to refer to a grid of GPU threads.

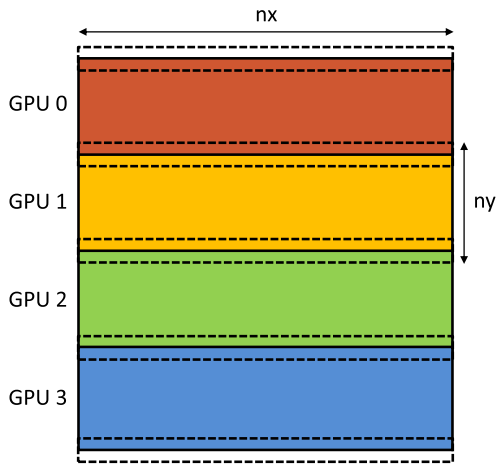


FIGURE 23.2

2D grid array for the modeling system partitioned across 4 GPUs.

After computing the new value at its grid point, each thread calculates its grid point's residue as the difference between the new value and the old value (line 13). To calculate the L2 norm, the squares of the residues need to be summed up across all threads in the grid. A block-wide reduction, such as that discussed in Chapter 10, is performed to sum up the squares of the residues across the entire block (line 16). After that, one thread in the block atomically accumulates the block-level sum to the grid-wide L2 norm (lines 17-19).

The kernel in Fig. 23.1 can be further optimized. For example, shared memory tiling and thread coarsening can be used to reduce global memory accesses, as described in Chapter 8. Furthermore, many optimizations can be applied to the reduction operation, as described in Chapter 10, or the block-wide reduction function from the CUB [5] library can be called. However, these optimizations are not the focus of this chapter. This chapter focuses on how the stencil computation can be distributed across multiple GPUs.

Fig. 23.2 illustrates the modeled system and how it can be partitioned across multiple GPUs. We assume that the system is modeled as a structured grid, where the spacing between grid points is constant in each direction. This assumption allows us to use a 2D array to represent the system, where each element stores the state of a grid point, as we discussed in Chapter 8. The physical distance between adjacent elements in each dimension can be represented by a grid spacing variable. In the 2D array, we assume that the data is placed in the memory space in the row-major layout, where x is the lowest dimension followed by y. That is, all elements with $y = 0$ are placed in consecutive memory locations according to their x coordinate, followed by all elements with $y = 1$, etc.

When one uses multiple GPUs, it is common to divide the input data into several partitions, called domain partitions, and assign each partition to a different GPU

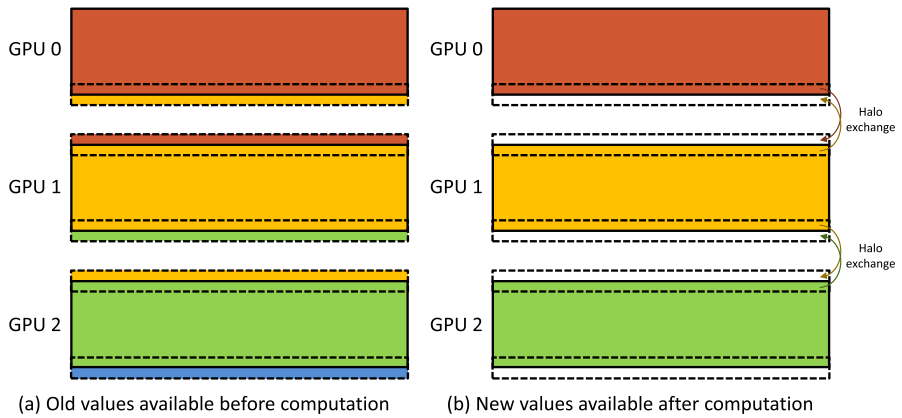


FIGURE 23.3

Values available to each GPU before and after computation.

in the cluster. With the row major layout, it is natural to partition the system across GPUs along the y-dimension, giving each GPU a contiguous set of rows. This partitioning is illustrated in Fig. 23.2 where the grid points that each GPU is responsible for computing are indicated by different colors and separated by solid lines. Such partitioning ensures that each GPU's partition is contiguous in memory according to the row-major layout.

Recall that to compute the new value at each grid point, the old values at the neighboring points are needed. Hence, for a GPU to compute the new grid point values at its partition's edge, it needs the old values of the grid points at the edges of the adjacent partitions. These grid points, called *halo* points, are indicated in Fig. 23.2 by the dotted lines. For example, for GPU 1 to compute the new values of the grid points in its partition, it needs the old values of all these grid points as well as the old values of the bottom row of GPU 0's partition and the top row of GPU 2's partition. The span of input rows needed as input by GPU 1 is indicated by ny. The first and last rows in the span are the *halo* points, used only as input, whereas the middle $ny - 2$ rows are both inputs and outputs.

Fig. 23.3 further illustrates the values available to each GPU before and after computing the grid point values in its partition. Before computing, the GPU needs the values of the halo rows as shown in Fig. 23.3(a). After computing, the values at its halo rows become out of date as shown in Fig. 23.3(b). The new values at the halo rows are computed by the other GPUs that are responsible these rows. Hence, before proceeding to the next iteration, a GPU needs to obtain the updated values of the halo rows from the GPUs computing the adjacent partitions. For example, GPU 1 needs to copy its top halo row from GPU 0 and its bottom halo row from GPU 2. Simultaneously, GPU 1 needs to send its top row to GPU 0 and its bottom row to GPU 2. This sending and receiving of halo rows between adjacent GPUs is referred to as *halo exchange*. Here, we observe another benefit of applying a 1D

domain decomposition across the y-dimension when using the row-major layout. The halo rows that need to be copied from one GPU to another are contiguous in memory, which simplifies the copy operation during halo exchange.

One drawback of this 1D domain decomposition scheme is that depending on the shape of the domain and the number of GPUs, large volumes of data may need to be communicated between GPUs. For example, if the domain is very wide, each GPU would need to communicate an entire row of the domain which can be expensive. Alternatively, one could use a 2D domain decomposition where each GPU takes a 2D tile of the domain. A 2D domain decomposition reduces the surface to volume ratio, thereby reducing the amount of data that needs to be communicated. However, a 2D domain decomposition would require each GPU to communicate with four other neighboring GPUs instead of just two, and some of the data that needs to be communicated would not be contiguous. To keep the examples in this chapter simple, we focus on the described 1D domain decomposition.

In summary, every iteration of the multi-GPU Jacobi iterative method consists of two phases: a computation phase, where the new values at the grid points are computed, and a communication phase, where the new values at the halo rows are exchanged between GPUs computing adjacent partitions. Moreover, since the L2 norm needs to encompass all the grid points in all the partitions, the communication phase additionally includes a reduction operation that reduces the grid-wide L2 norms computed by each GPU. Each of MPI, NCCL, and NVSHMEM provide their own mechanisms for exchanging data between GPUs. We will see how these operations can be done with each programming model throughout this chapter.

23.2 Multi-GPU stencil with MPI

There are two main categories of parallel programming models: shared memory models and distributed memory models. In *shared memory models*, different threads or processes have access to the same memory space, which simplifies data sharing. For example, CUDA kernels follow a shared memory model because different CUDA threads can access the same global memory variable using the same pointer. In distributed memory models, different threads or processes have access to distinct memory spaces. In this case, the sharing of data must be done explicitly, such as by passing messages.

The dominating programming interface for computing clusters today is Message Passing Interface (MPI) [1], which is a set of API functions for communication between processes running in a computing cluster. MPI assumes a distributed memory model where processes exchange information by sending messages to each other. When an application uses API communication functions, it does not need to deal with the details of the interconnect network. The MPI implementation allows the processes to address each other using logical numbers, much the same way as using phone numbers in a telephone system: telephone users can dial each other using

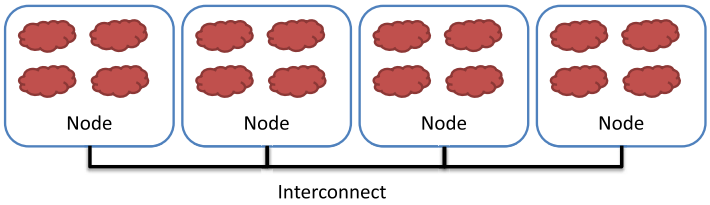


FIGURE 23.4

Programmer's view of MPI processes.

- `int MPI_Init(int *argc, char ***argv)`
  - Initialize MPI
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`
  - Rank of the calling process in group of comm
- `int MPI_Comm_size (MPI_Comm comm, int *size)`
  - Number of processes in the group of comm
- `int MPI_Finalize ()`
  - `Ending an MPI application, close all resources`

FIGURE 23.5

Basic MPI API functions for establishing and closing a communication system.

phone numbers without knowing exactly where the called person is and how the call is routed.

In a typical MPI application, data and work are partitioned among processes. As shown in Fig. 23.4, each node can contain one or more processes, shown as clouds within nodes. As these processes progress, they may need data from each other, such as when we need to perform halo exchange in the Jacobi iterative method. This need is satisfied by sending and receiving messages. In some cases, the processes also need to synchronize with each other and generate collective results, such as when reducing the L2 norm. This is done with collective communication API functions.

Like CUDA, MPI programs are based on the SPMD parallel programming model. All MPI processes execute the same program. The MPI system provides a set of API functions to establish communication systems that allow the processes to communicate with each other. Fig. 23.5 shows five essential MPI functions that set up and tear down the communication system for an MPI application. Fig. 23.6 shows how an MPI application typically uses these functions.

To launch an MPI application in a cluster, a user needs to supply the executable file of the program to the `mpirun` command or the `mpiexec` command, which creates the processes that run executable file.[2] Each process starts by initializing the MPI runtime

[2] While the MPI standard suggests `mpiexec` as the standard name for launching MPI processes, implementations such as Open MPI offer both as synonymous commands.

```
01    #include "mpi.h"
02    int main(int argc, char *argv[]) {
03        MPI_Init(&argc, &argv);
04        int rank;
05        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
06        int numRanks;
07        MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
08        ...
09        MPI_Finalize();
10        return 0;
11    }
```

FIGURE 23.6

A simple MPI main program.

with a `MPI_Init()` call (line 03). This call initializes the communication system for all the processes running the application.

Next, the `MPI_Comm_rank()` function is called (line 05), which returns a unique number to each calling process called the MPI rank of the process. The numbers received by the processes vary from 0 to the number of processes minus 1. The MPI rank for a process uniquely identifies the process in a communication, which is equivalent to the phone number in a telephone system. It is analogous to the expression `blockIdx.x*blockDim.x + threadIdx.x` for a CUDA thread. Recall that a CUDA thread can have up to three dimensions. Similarly, MPI ranks can be organized into multiple dimensions using MPI Cartesian communicators.

The `MPI_Comm_rank()` function takes two parameters. The first one is an MPI built-in type `MPI_Comm` that specifies the scope of the request, i.e., the collection of processes that form the group identified by a `MPI_Comm` variable. Each variable of the `MPI_Comm` type is commonly referred to as a communicator. `MPI_Comm` and other MPI built-in types are defined in the `"mpi.h"` header file (line 01) that should be included in all program files that use MPI. An MPI application can create one or more communicators each of which specifies a group of MPI processes to be included when calling a communication function. `MPI_Comm_rank()` assigns a unique id to each process within the group specified by the communicator parameter when it is called. In Fig. 23.6, the parameter value passed is `MPI_COMM_WORLD`, which is used as a default and means that the communicator includes all MPI processes running the application.[3]

The second parameter to the `MPI_Comm_rank()` function is a pointer to an integer variable into which the function will deposit the returned rank value. In Fig. 23.6, a variable `rank` is declared for this purpose. After the `MPI_Comm_rank()` returns, the `rank` variable will contain the unique id for the calling process within the group of processes specified by the given communicator. In the MPI literature, the terms MPI process and *MPI rank* are used interchangeably.

[3] Interested readers should refer to the MPI reference manual [1] for details on creating and using multiple communicators in an application, in particular the definition and use of intracommunicators and intercommunicators.

Next, the `MPI_Comm_size()` function is called (line 07), which returns the total number of MPI processes running in the communicator. The total number of MPI processes specified by a communicator is referred to as the *number of ranks* for the communicator. The `MPI_Comm_size()` function takes two parameters. The first parameter is of `MPI_Comm` type that gives the scope of the request. In Fig. 23.6, the parameter value passed in is `MPI_COMM_WORLD`, which means the scope of the `MPI_Comm_size()` is all the processes of the application. The second parameter is a pointer to an integer variable into which the `MPI_Comm_size()` function will deposit the return value. In Fig. 23.6, a variable `numRanks` is declared for this purpose. Since the scope of the call is all MPI processes, after the function returns, the variable `numRanks` contains the number of MPI processes running the application. This value is configured by the user when the application is executed using the `mpirun` command or the `mpiexec` command. Querying the value from the MPI runtime using `MPI_Comm_size()` avoids making implicit assumptions about how the application was launched when determining how many partitions need to be created.

Next, the application program goes on to perform the main computation (line 08). We will discuss this part of the code in the rest of this section. After the application completes the main computation, it notifies the MPI runtime with a call to `MPI_Finalize()`, which frees all MPI communication resources allocated to the application (line 09). The application can then exit with a return value of 0, which indicates that no error has occurred (line 10).

We now turn our attention to the main computation performed by each process in the MPI application. With each process knowing its rank and the number of ranks in the application, each process can identify what partition of the modeled system it is responsible for. The process then allocates GPU memory for its partition using the familiar `cudaMalloc` function, initializes that memory using the familiar `cudaMemset` function, and launches a simple kernel to initialize its partition's boundary elements based on the boundary conditions of the system being modeled.

We omit this initialization code and focus our attention on the main computation host code that iteratively performs the stencil computation, halo exchange, and L2 norm reduction. The code assumes a process `P` executes the host code in Fig. 23.7 on a host CPU, it allocates the `output` and `input` arrays in the global memory of the GPU connected to the host CPU. Each of these arrays contains `ny` × `nx` elements, as illustrated in Fig. 23.3. Note that the first row (elements 0 through `nx-1`) of the array consists of the halo elements from the top neighbor. Similarly, the last row (elements `(ny-1)*nx` through `ny*nx-1`) consists of the halo elements from the bottom neighbor.

The main loop of the MPI-based multi-GPU host code is shown in Fig. 23.7. The loop iterates until the L2 norm drops below the minimum tolerance threshold or until the maximum number of allowed iterations is exceeded (line 01). In each iteration, the L2 norm is first initialized to 0 with a call to `cudaMemset` (line 04). Next, a function is called (line 07) that launches the Jacobi kernel implemented in Fig. 23.1. The Jacobi kernel performs one iteration of the stencil computation, reading the old grid point values in the `input` array and writing the new grid point values to the `output` array, while also updating the L2 norm. These arrays are swapped in each iteration in

```
01    for(unsigned int iter = 0; l2norm > tol && iter < iter_max; ++iter) {
02
03        // Reset L2 norm
04        cudaMemset(l2norm_d, 0, sizeof(float));
05
06        // Compute stencil
07        launch_jacobi_kernel(output, input, l2norm_d, ny, nx);
08        cudaDeviceSynchronize();
09
10        // Halo exchange
11        const int top = rank > 0 ? rank - 1 : (numRanks - 1);
12        const int bottom = (rank + 1) % numRanks;
13        MPI_Sendrecv(output + nx, nx, MPI_FLOAT, top, 0,
14                     output + (ny - 1)*nx, nx, MPI_FLOAT, bottom, 0,
15                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
16        MPI_Sendrecv(output + (ny - 2)*nx, nx, MPI_FLOAT, bottom, 0,
17                     output, nx, MPI_FLOAT, top, 0,
18                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
19
20        // Copy L2 norm to host
21        cudaMemcpy(l2norm_h, l2norm_d, sizeof(float), cudaMemcpyDeviceToHost);
22
23        // Reduce L2 norm
24        MPI_Allreduce(l2norm_h, &l2norm, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
25        l2norm = std::sqrt(l2norm);
26
27        std::swap(output, input);
28
29    }
```

FIGURE 23.7

Host code for MPI-based multi-GPU implementation of the Jacobi iterative method.

a double-buffering fashion (line 27). We wait for the Jacobi kernel to finish execution (line 08) to ensure that the new grid point values are computed before we begin the halo exchange.

The next step is to perform the halo exchange. The halo exchange requires the MPI processes to communicate with their neighbors to send their new boundary grid point values to each other. Such communication can be done with MPI *point-to-point* communication, which involves exchanging message between two communicating processes. MPI provides many different types of point-to-point communication routines. In our code example, we use the `MPI_Sendrecv()` routine. To understand what `MPI_Sendrecv()` does, let us first look at the more basic point-to-point communication routines that MPI provides.

The most basic point-to-point communication routines in MPI are `MPI_Send()` and `MPI_Recv()`, where a source process calls the `MPI_Send()` function to send data to a destination process, and the destination process calls the `MPI_Recv()` function to receive that data from the source process. This is analogous to a caller dialing a call and a receiver answering a call in a telephone system. Because both sides of the communication must participate for the communication to take place, such point-to-point communication is referred to as *two-sided communication*.

Fig. 23.8 shows the signature of the `MPI_Send()` function. The first parameter is a pointer to the starting location of the memory area where the data to be sent can be found. The second parameter is an integer that gives that number of data elements to be sent. The third parameter is of an MPI built-in type `MPI_Datatype`. It

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Buf: starting address of send buffer (pointer)
  - Count: Number of elements in send buffer (nonnegative integer)
  - Datatype: Datatype of each send buffer element (MPI_Datatype)
  - Dest: Rank of destination (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)

FIGURE 23.8

Signature of the `MPI_Send()` function.

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - buf: starting address of receive buffer (pointer)
  - Count: Maximum number of elements in receive buffer (integer)
  - Datatype: Datatype of each receive buffer element (MPI_Datatype)
  - Source: Rank of source (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)
  - Status: Status object (Status)

FIGURE 23.9

Signature of the `MPI_Recv()` function.

specifies the type of each data element being sent as far as the MPI library implementation is concerned. The values that can be held by a variable or argument of the `MPI_Datatype` are defined in `"mpi.h"` and include `MPI_DOUBLE` (double-precision floating point), `MPI_FLOAT` (single-precision floating point), `MPI_INT` (integer), and `MPI_CHAR` (character). The exact sizes of these types depend on the size of the corresponding C types in the host processor. See the MPI reference manual for more sophisticated use of MPI types [1]. The fourth parameter for `MPI_Send()` is an integer that gives the MPI rank of the destination process. The fifth parameter gives a tag that can be used to classify the messages sent by the same process. The sixth parameter is a communicator that specifies the group of processes in which the destination MPI rank is defined.

Fig. 23.9 shows the signature of the `MPI_Recv()` function. The first parameter is a pointer to the area in memory where the received data should be deposited. The second parameter is an integer that gives the maximum number of elements that the `MPI_Recv()` function is allowed to receive. The third parameter is an `MPI_Datatype` that specifies the type of each element to be received. The fourth parameter is an integer that gives the rank of the source of the message. The fifth parameter is an integer that specifies the tag value expected by the destination process. If the destination process does not want to be limited to a particular tag value, it can use `MPI_ANY_TAG`, which means that the receiver is willing to accept messages of any tag value from the source.

For halo exchange, when a process sends its boundary data to its neighbor, it also needs to receive halo data from its other neighbor simultaneously. For example, in

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - Sendbuf: Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

FIGURE 23.10

Signature of the `MPI_Sendrecv()` function.

Fig. 23.3, when process 1 (i.e., the process controlling GPU 1) sends its top boundary grid point values to process 0 (i.e., the process controlling GPU 0), process 2 (i.e., the process controlling GPU 2) will also be sending its top boundary grid point values to process 1. All processes need to send their boundary data to their top neighbors and receive halo data from their bottom neighbors simultaneously. For such situations, MPI provides the `MPI_Sendrecv()` function that sends data to a destination and receives data from a source simultaneously. The signature of the `MPI_Sendrecv()` function is shown in Fig. 23.10. It is a combination of `MPI_Send()` and `MPI_Recv()` functions so we will not further elaborate on the meaning of the parameters.

The same objective can also be accomplished using MPI non-blocking communication. In particular, we could initiate a non-blocking send using the `MPI_Isend` function and a non-blocking receive using the `MPI_Irecv` function, and then wait for the two operations to finish using the `MPI_Wait` function. These non-blocking communication functions enable expressing more complex communication patterns that requires concurrent sending and receiving to and from different neighbors. However, the `MPI_Sendrecv()` function provides a convenient way of expressing the particular concurrent communication pattern that we are interested in using just a single API call. Interested readers can refer to the MPI documentation for more details on how to use non-blocking communication functions [1]

We now turn our attention back to the code in Fig. 23.7, particularly the code that performs the halo exchange (lines 11-18). First, each process identifies the neighbor process at its top (line 11) and the one at its bottom (line 12). Rather than simply computing `rank - 1` and `rank + 1`, the code uses a wrap-around strategy where the topmost rank treats the bottommost rank as its top neighbor, and vice versa. This wrap-around strategy is often referred to as the *periodic boundary condition technique* for modeling a large system by simulating only a small portion of it without introducing a disproportional amount of edge effects. Such boundary conditions require special handling of the top and the bottom ranks since they would not have a top neighbor and bottom neighbor, respectively, to communicate with.

Next, we have two calls to `MPI_Sendrecv` that perform the halo exchange. The first call (lines 13-15) sends the top boundary row, which starts at `output + nx`, to the top neighbor, and receives the bottom halo row, which starts at `output + (ny - 1)*nx`, from the bottom neighbor. The second call (lines 16-18) sends the bottom boundary row, which starts at `output + (ny - 2)*nx`, to the bottom neighbor, and receives the top halo row, which starts at `output`, from the top neighbor. In all cases, the number of elements communicated is `nx`, the size of the row, and the datatype of `MPI_FLOAT`, which corresponds to the `float` datatype in C and C++.

It is important to note that the `output` and `input` pointers passed to the `MPI_Sendrecv` function are pointers to the GPU device memory. Passing GPU device memory pointers to MPI functions is possible with CUDA-aware MPI. If the MPI implementation used is not CUDA-aware, the programmer would have needed to copy the data from device memory to host memory and invoke the MPI communication function on the host memory array. In contrast, CUDA-aware MPI implementations are aware of the CUDA programming model and are designed to allow communication functions to be invoked on CUDA global memory arrays directly. Allowing send and receive functions to work directly on the device arrays eliminates the need for `cudaMemcpy` calls in the source code and thus minimizes the bloat to the host code size as well as the added communication latency. Most MPI implementations (e.g., MPICH [6], Open-MPI [7], MVAPICH2 [8]) are designed to be CUDA-aware and are built with this feature enabled.

After the halo exchange, the host code needs to calculate the L2 norm to identify if the system has significantly changed and we need to continue iterating. Each GPU has already computed the square of the L2 norm of its own partition's residual. However, we still need to sum up these L2 norm squares across all the partitions. In other words, we need to perform a reduction operation across all the MPI ranks, where each rank contributes its own partition's L2 norm square. Such a reduction across ranks can be done with MPI collective communication.

In contrast with MPI point-to-point communication, such as `MPI_Sendrecv`, which involves communicating between two ranks, MPI *collective communication* involves communicating between all ranks in the communicator. The commonly used types of MPI collective communication are barrier, broadcast, reduce, gather, and scatter. We will not elaborate on all types of MPI collective communication functions but encourage the reader to read the MPI programming guide [1] for the details of these functions. In general, collective communication functions are highly optimized by the MPI runtime developers and system vendors. Using the collective communication functions usually leads to better performance as well as readability and productivity than trying to achieve the same functionality with sequences of send and receive calls.

The MPI collective communication function that we are interested in for computing the L2 norm is `MPI_Allreduce`. The `MPI_Allreduce` function takes a buffer of values from each rank, reduces the buffers into a single buffer, and gives a copy of that buffer to each rank. Fig. 23.11 shows the signature of the `MPI_Allreduce` function. The first parameter, `sendbuf`, specifies the input buffer that each rank contributes to the reduction operation. The second parameter, `recvbuf`, specifies where the final

- `int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
  - sendbuf: starting address of the send buffer
  - recvbuf: starting address of the receive buffer
  - count: number of elements in the send buffer
  - datatype: datatype of elements of the send buffer
  - op: operation
  - comm: communicator

FIGURE 23.11

Signature of the `MPI_Allreduce()` function.

buffer containing the result of the reduction should be placed. That is, every element in the receive buffer contains the reduced value of the elements at the same index in the send buffers of all the ranks. The third parameter, `count`, specifies the number of elements in the send buffer, which is equal to the number of elements in the receive buffer as well. The fourth parameter, `datatype`, specifies the type of the elements being reduced, such as integer, floating point, etc. The fifth parameter, `op`, specifies the reduction operation being performed. MPI provides many built-in reduction operations such as sum, product, minimum, maximum, logic operations, etc. The sixth and final parameter, `comm`, specifies the communicator whose ranks are participating in the reduction operation.

We now turn our attention back to the code in Fig. 23.7, particularly the code that reduces the L2 norm (lines 20-25). First, the partition's square of the L2 norm is copied from the device to the host (line 21). Next, the `MPI_Allreduce` collective is used to reduce the L2 norm squares across all ranks (line 24). The number of elements in the buffers (third parameter) in this case is 1 since each rank contributes one value, which is its partition's L2 norm square. The reduction operation (fifth parameter) is `MPI_SUM`, which indicates that we would like to calculate the sum of the L2 norms across all the ranks. After `MPI_Allreduce` returns, every rank now has the sum of all the L2 norm squares. The square root of this value is calculated yielding the ultimate L2 norm across the entire system (line 25), which is used to check whether or not the loop should continue iterating (line 01).

The reader may wonder why the square of the L2 norm was copied from the device to the host before invoking `MPI_Allreduce`. We could have invoked `MPI_Allreduce` directly on the GPU buffer since it is CUDA-aware. The reason is that the result of `MPI_Allreduce` is needed on the host to check for convergence, so it needs to be copied to the host anyway. In this case, it is more efficient to copy it to the host before invoking `MPI_Allreduce`.

23.3 Overlapping computation and communication

In our multi-GPU stencil implementation in Section 23.2, each process computes the new values of its entire partition, exchanges the halo data with the top and bottom

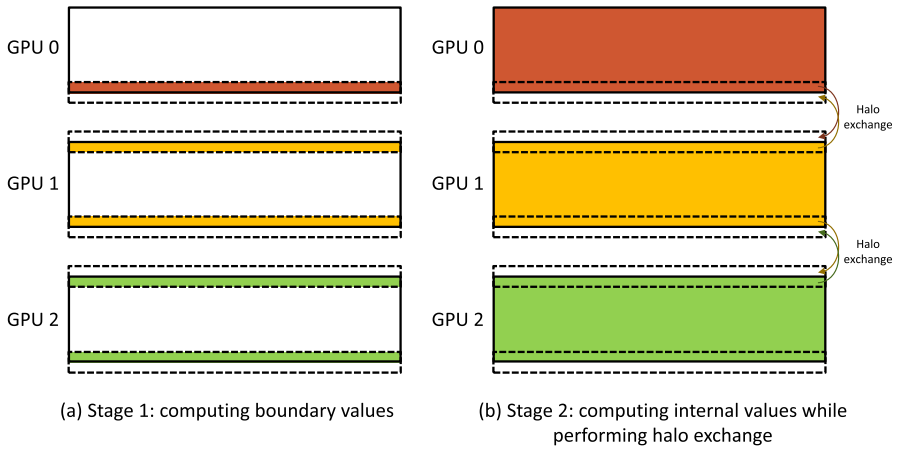


FIGURE 23.12

A two-stage strategy for overlapping computation with communication.

neighbors, and repeats. While this strategy is very simple, it is not very efficient. The reason is that this strategy forces the system to be in one of two modes, where each mode only utilizes part of the system. In the first mode, all processes are performing computation using the GPUs. During this time, the communication network is not used. In the second mode, all processes are exchanging halo data with their top and bottom neighbors using the communication network. During this time, the GPU compute hardware is not well utilized. Ideally, we would like to achieve better performance by utilizing both the communication network and GPU compute hardware all the time. This improved utilization can be achieved by dividing the computation tasks of each process into two stages, as illustrated in Fig. 23.12.

During the first stage (Stage 1), each process calculates its boundary rows that are needed as halo cells by its neighbors in the next iteration. Fig. 23.12 shows the boundary rows computed in this stage as colored rows. For example, GPU 1 calculates its top and bottom boundary rows, which are the halo rows of GPU 0 and GPU 2, respectively. The rationale is that these boundary rows are needed by the neighbors for the next iteration. By calculating these boundary rows first, the data can be communicated to the neighbors in the second stage while the rest of their internal grid points are calculated.

During the second stage (Stage 2), each process performs two activities in parallel. The first is to communicate its new boundary values to its neighbor processes. The second activity is to calculate the rest of the data in the partition. If the communication activity takes a shorter amount of time than the calculation activity, we can hide the communication delay and fully utilize the computing hardware most of the time. This is usually achieved by having enough rows in the internal part of each partition to allow each process to perform enough computation to hide the communication.

Fig. 23.13 compares the timeline of execution without and with overlapping computation and communication. In Fig. 23.13(a), when computation and communication

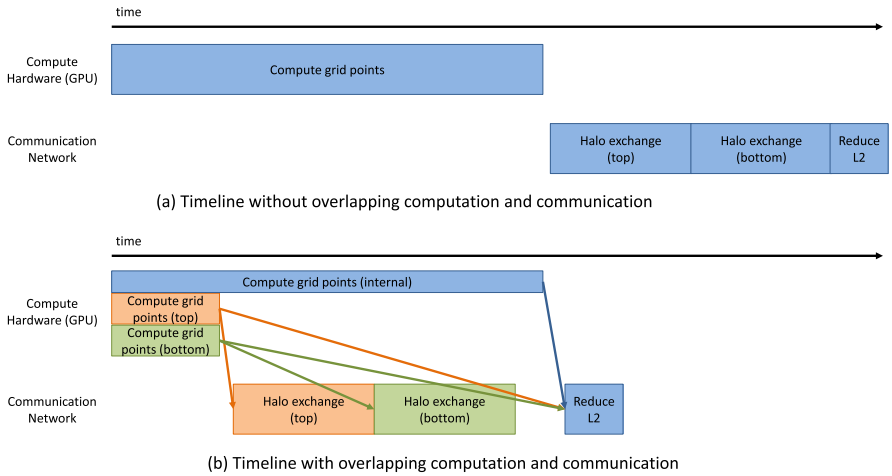


FIGURE 23.13

Comparing timelines without and with overlapping computation and communication.

are not overlapped, the entire computation is performed by the compute hardware while the communication network is idle. Subsequently, the top and bottom halos are exchanged over the communication network while the compute hardware is idle. The L2 norm reduction is also performed. In contrast, in Fig. 23.13(b), the top and bottom grid points are computed distinctly from the internal grid points, and the top and bottom halo exchanges are initiated as soon as the corresponding grid points are computed. For GPUs with a sufficient amount of compute resources, grid points in the three categories can be computed concurrently. However, priority should be given to the boundary grid points in order to initiate the halo exchange as soon as possible. Importantly, the internal grid points continue to be computed while the halo exchanges are performed. This overlap between computing the grid points and performing the halo exchanges ensures that both the compute hardware and communication network are utilized simultaneously, resulting in the overall execution to finish sooner. Inevitably, reducing the L2 norm cannot be overlapped because it depends on all the grid points being computed.

Accomplishing this kind of overlap between communication and computation requires splitting the Jacobi kernel call into separate kernel calls for computing the boundary elements and the internal elements. For example, we can split the kernel call into three different calls: one for the top boundary row, one for the bottom boundary row, and one for the internal elements. In this way, the halo exchanges can be initiated as soon as the kernels processing the boundary rows are done. To execute these three kernels concurrently and synchronize the communication activities with each of them independently, we need to use an important feature of the CUDA programming model which is CUDA streams.

In CUDA, a *stream* is an ordered sequence of CUDA operations. These operations could include kernel calls and memory copies, among other things. All operations in

the same stream are done sequentially according to the order in which they are placed into that stream. On the other hand, operations in different streams may be executed concurrently without any ordering constraint. Hence, to get the three aforementioned Jacobi kernels to execute concurrently, we should insert them into three different streams, which starts with the creation of stream objects in the following code:

```
cudaStream_t topStream, bottomStream, internalStream;
cudaStreamCreate(&internalStream);
cudaStreamCreate(&topStream);
cudaStreamCreate(&bottomStream);
```

The first line declares three stream objects of type `cudaStream_t`. The stream objects are named `topStream`, `bottomStream`, and `internalStream`, referring to the streams that will contain the calls to the kernels that process the top boundary row, the bottom boundary row, and the internal rows, respectively. The next three lines create these streams by calling `cudaStreamCreate` and passing to it a pointer to the stream object. Each call to `cudaStreamCreate()` creates a new stream and initializes the state of that stream.

To insert a kernel call into a particular stream, the stream object can be passed as a configuration parameter for the kernel call. Recall from Chapter 2 that a kernel call takes two required configuration parameters in triangle brackets, the first being the grid dimension and the second being the block dimension. Also recall from Chapter 5 that the kernel call takes an optional third configuration parameter which is the size of the dynamically allocated shared memory. If not specified, the value is 0 by default. Similarly, the kernel call also takes an optional fourth configuration parameter which is the stream into which the kernel call is to be inserted. If not specified, the kernel call is inserted into a *default stream*. The following code shows how a kernel call can be inserted into a specific stream:

```
kernelName<<<dimGrid, dimBlock, 0, myStream>>>(...)
```

This code inserts a call to kernel `kernelName` with a grid dimension specified in `dimGrid`, a block dimension specified in `dimBlock`, and zero dynamically allocated shared memory, and inserts itself into the stream specified in `myStream`. Note that returning from the call does not mean that the kernel has been executed. It simply means that the kernel has been inserted into the designated stream. The execution of the host code will continue with the following code as soon as the kernel has been inserted into the stream. The fact that a kernel call in the host code inserts the kernel into a stream, rather than actually executing the kernel, makes the CUDA kernel calls *asynchronous*. Furthermore, the continued execution of the host code after the kernel call does not mean that the corresponding thread grid has completed executing the kernel. It does not even mean that the thread grid has started executing the kernel. It simply means that the kernel has been inserted into the designated stream and a grid will eventually be launched to execute the kernel after all preceding operations in the stream have completed their execution.

```
01    for(unsigned int iter = 0; l2norm > tol && iter < iter_max; ++iter) {
02
03        // Reset L2 norm
04        cudaMemsetAsync(l2norm_d, 0, sizeof(float), internalStream);
05        cudaEventRecord(resetL2, internalStream);
06        cudaStreamWaitEvent(topStream, resetL2, 0);
07        cudaStreamWaitEvent(bottomStream, resetL2, 0);
08
09        // Compute stencil
10        launch_jacobi_kernel(output, input, l2norm_d, 3, nx, topStream);
11        cudaEventRecord(computeTop, topStream);
12        launch_jacobi_kernel(output + (ny - 3)*nx, input + (ny - 3)*nx, l2norm_d, 3,
13                             nx, bottomStream);
14        cudaEventRecord(computeBottom, bottomStream);
15        launch_jacobi_kernel(output + nx, input + nx, l2norm_d, ny - 2, nx,
16                             internalStream);
17
18        // Copy L2 norm to host
19        cudaStreamWaitEvent(internalStream, computeTop, 0));
20        cudaStreamWaitEvent(internalStream, computeBottom, 0);
21        cudaMemcpyAsync(l2norm_h, l2norm_d, sizeof(float), cudaMemcpyDeviceToHost,
22                        internalStream);
23
24        // Halo exchange
25        const int top = rank > 0 ? rank - 1 : (numRanks - 1);
26        const int bottom = (rank + 1) % numRanks;
27        cudaStreamSynchronize(topStream);
28        MPI_Sendrecv(output + nx, nx, MPI_FLOAT, top, 0,
29                     output + (ny - 1)*nx, nx, MPI_FLOAT, bottom, 0,
30                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
31        cudaStreamSynchronize(bottomStream);
32        MPI_Sendrecv(output + (ny - 2) * nx, nx, MPI_FLOAT, bottom, 0,
33                     output, nx, MPI_FLOAT, top, 0,
34                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
35
36        // Reduce L2 norm
37        cudaStreamSynchronize(internalStream);
38        MPI_Allreduce(l2norm_h, &l2norm, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
39        l2norm = std::sqrt(l2norm);
40
41        std::swap(output, input);
42
43    }
```

FIGURE 23.14

Host code for MPI-based multi-GPU implementation of the Jacobi iterative method that overlaps computation and communication.

Having seen how to create streams and insert kernel calls into them, we now proceed to split our Jacobi kernel into three calls to be inserted into three different streams. Fig. 23.14 shows the host code for the MPI-based multi-GPU implementation of the Jacobi iterative method that overlaps computation and communication. Compared to the code in Fig. 23.7 which calls the `launch_jacobi_kernel` a single time launching a single kernel per iteration, the code in Fig. 23.14 makes three different calls to compute the top boundary row (line 10), the bottom boundary row (line 12-13), and the internal grid points (line 15-16).

To compute the top boundary row (line 10), the function needs the first three rows (top halo row, top boundary row, and top internal row) as an input so we pass the `output` and `input` pointers as they are, and pass `3` as the fourth parameter indicating that there are three input rows of interest. Importantly, we also pass `topStream` as an

additional parameter, into which the `launch_jacobi_kernel` function inserts the kernel call.

As for the bottom boundary row (line 12-13), the function needs the last three rows (bottom halo row, bottom boundary row, and bottom internal row) as an input so we pass the `output + (ny - 3)*nx` and `input + (ny - 3)*nx` pointers which point to the beginning of the last three rows, and pass `3` as the fourth parameter indicating that there are three input rows of interest. We also pass `bottomStream` as the stream into which the kernel call is inserted.

Finally, to calculate the internal grid points (line 15-16), the function needs all the rows except the top halo and bottom halo rows so we pass `output + nx` and `input + nx` pointers which skip the first halo row, and pass `ny - 2` as the fourth parameter omitting the two halo rows. We also pass `internalStream` as the stream into which the kernel call is inserted.

One important consideration to keep in mind when executing the three kernels concurrently in three different streams is that these kernels may execute in any order. The likely scenario is that the kernels computing the boundary grid point values will execute first because they were called first, but this ordering is not guaranteed. It is possible for the kernel computing the internal grid point values to execute first and occupy all the GPU resources, preventing the kernels computing the boundary grid point values and the subsequent halo exchange from executing, and thereby not achieving the desired overlap between computation and communication. To avoid such a situation, we need to give higher priority for the kernel computing the boundary grid point values than we give to the kernel computing the internal grid point values. In other words, we need to give higher priority to the `topStream` and `bottomStream` than we do to the `internalStream`.

By default, streams created with `cudaStreamCreate` have the same priority. To assign different priorities to different streams, we need to use different CUDA API functions. The following code shows how streams can be created with different priorities:

```
int low, high;
cudaDeviceGetStreamPriorityRange(&low, &high);
cudaStreamCreateWithPriority(&internalStream, cudaStreamDefault, low);
cudaStreamCreateWithPriority(&topStream, cudaStreamDefault, high);
cudaStreamCreateWithPriority(&bottomStream, cudaStreamDefault, high);
```

On the second line, the function `cudaDeviceGetStreamPriorityRange` returns the range (high value and low value) of stream priority values supported by the CUDA runtime. The third and fourth/fifth lines then use `cudaStreamCreateWithPriority` to create the `internalStream` and `topStream/bottomStream` with the lowest and highest priority, respectively. In our example, we are only interested in distinguishing between two kinds of streams, so two priority values are sufficient. In other cases, when may be interested in more than two priority levels and use intermediate values as well. The absolute values of the priority values are not important. Only their relative values matter.

Having inserted the three Jacobi kernel calls into three different streams so they may execute concurrently, we now turn our attention to how we can wait for specific kernels to finish before initiating subsequent tasks such as halo exchange. CUDA provides a function `cudaStreamSynchronize` that waits for all preceding (existing) tasks in a specific stream to finish executing. We can use this function to wait for the kernels that compute the top and bottom boundary rows to finish executing before initiating the top and bottom halo exchanges, as shown in Fig. 23.14. Compared to the code in Fig. 23.7, Fig. 23.14 adds a call that waits for the kernel calculating the top boundary row in `topStream` to finish (line 27) before calling `MPI_Sendrecv` to perform the halo exchange of that top boundary row. Similarly, Fig. 23.14 adds a call that waits for the kernel calculating the bottom boundary row in `bottomStream` to finish (line 32) before calling `MPI_Sendrecv` to perform the halo exchange of that bottom boundary row. In the meantime, since the kernel solving the internal rows is in `internalStream` which we have not synchronized on, that kernel may proceed to execute concurrently as the `MPI_Sendrecv` halo exchanges are being performed. In this way, we have enabled overlapping computation of the internal rows with communication of the halo rows.

Although splitting the kernel call into three different calls enables synchronizing on the three calls independently for the purpose of halo exchange, it introduces a challenge when it comes to computing the L2 norm. All three kernels update the same L2 norm, which means that all three kernels must wait for the L2 norm to be initialized to 0, and all three kernels must complete before the L2 norm can be copied to the host and reduced across ranks. However, the kernel calls are placed in three different streams, but the calls to initialize and copy the L2 norm can only go in a single stream. To overcome this challenge, we need to be able to have operations in one stream wait for operations in another stream to complete. Such interaction between streams can be accomplished with CUDA events.

In CUDA, an *event* is a marker that marks a certain point in a stream. CUDA allows programmers to synchronize on specific events, i.e., wait for the execution in a stream to reach the point marked by the event. Such synchronization can be done by the host thread, or a synchronization operation on an event can be placed in another stream. The latter capability is the one we are interested in for the purpose of initializing and copying the L2 norm.

To initialize the L2 norm to zero in the GPU's global memory, the code in Fig. 23.7 uses `cudaMemset` which is a blocking function call that does not return until the initialization operation is complete. In contrast, the code in Fig. 23.14 uses a variant of `cudaMemset`, `cudaMemsetAsync` (line 04), that is non-blocking. Like kernel calls, the `cudaMemsetAsync` function inserts the initialization operation into the stream specified by an additional parameter. In our case, we place `cudaMemsetAsync` in `internalStream` which ensures that subsequent operations in `internalStream` (i.e., the kernel call that solves the internal rows) do not execute before `cudaMemsetAsync` completes.

However, we also need to make sure that the other two kernels that calculate the top and bottom boundary rows do not execute before `cudaMemsetAsync` completes. This

can be done by inserting an event in `internalStream` after `cudaMemsetAsync`, and inserting waiting (i.e., synchronization) operations that wait for that event in `topStream` and in `bottomStream` before the kernels.

The following code shows how an event can be created for the aforementioned synchronization purpose:

```
cudaEvent_t resetL2;
cudaEventCreateWithFlags(&resetL2, cudaEventDisableTiming);
```

The first line declares an event object of type `cudaEvent_t`. The second line creates the event by calling `cudaEventCreateWithFlags` with a pointer to the event object as its first parameter. The second parameter, `cudaEventDisableTiming`, is a compile-time constant indicating that we do not wish to use this event for timing purposes, which allows synchronization activities based on the event to be implemented more efficiently. With the event created, we can place it in the `internalStream` after the call to `cudaMemsetAsync` using the function `cudaEventRecord` as shown in Fig. 23.14 (line 05). In this case, the event `resetL2` now marks the completion of the `cudaMemsetAsync` operation.

To insert waiting operations for that event in another stream, namely `topStream` and `bottomStream`, we can use the function `cudaStreamWaitEvent` as shown in Fig. 23.14 (lines 06-07). The first parameter specifies the stream into which the waiting operation should be inserted, namely `topStream` (line 6) and `bottomStream` (line 7). The second parameter specifies the event that the operation is waiting for, namely `resetL2` for both. The third parameter indicates if any special options should be used for the waiting operation. The 0 value for the third parameter indicates that the default option should be used.

We now turn our attention to copying back the L2 norm after all three kernels have completed execution. In Fig. 23.7, this copy was done using `cudaMemcpy`. In contrast, the code in Fig. 23.14 uses a variant of `cudaMemcpy`, `cudaMemcpyAsync` (line 21), that is non-blocking and takes an additional parameter for the stream it is to be placed in. In our case, we place `cudaMemcpyAsync` in `internalStream` which ensures that it waits for preceding operations in `internalStream` (i.e., the kernel call that calculates the internal rows).

However, we also want it to wait for the other two kernels that solve the top and bottom boundary rows. This can be done by inserting events in `topStream` and `bottomStream` after the kernel calls, and inserting waiting operations for them in `internalStream` before `cudaMemcpyAsync`. Indeed, the code in Fig. 23.14 uses `cudaEventRecord` to insert such events in `topStream` (line 11) and `bottomStream` (line 14), and inserts waiting operations for these events in `internalStream` (lines 19-20) before the memory copy operation.

One further optimization applied to our host code in Fig. 23.14, based on the fact that `cudaMemcpyAsync` is non-blocking, is that we can use `cudaMemcpyAsync` to insert the memory copy of the l2norm from the global memory to the host memory before the calls to `MPI_Sendrecv` as shown in Fig. 23.14. Doing so overlaps the

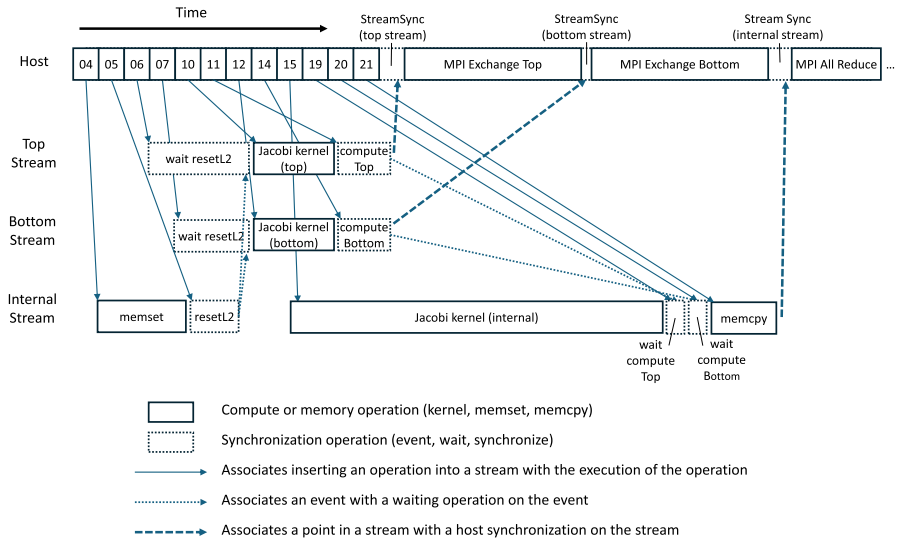


FIGURE 23.15

An example timeline for overlapping computation and MPI communication using CUDA streams and events.

memory copy from the GPU to the host CPU with the network communication performing the halo exchange if the kernel computing the internal grid points finishes before the halo exchange does. However, we must ensure that the memory copy completes before the call to `MPI_Allreduce` is performed. This can be done with a call to `cudaStreamSynchronize` which waits for the `internalStream` where `cudaMemcpyAsync` is executing (line 37).

Fig. 23.15 illustrates how the host code in Fig. 23.14 accomplishes the goal of overlapping the computation of the internal grid points and the exchange of halos within each iteration of the Jacobi solver (line 04 through line 41). The four timelines show the activities of the host CPU, `topStream`, `bottomStream`, and `internalStream`. For the host CPU, each slot is either an insertion into one of the streams (marked with the line number of the host code in Fig. 23.14) or the nature of the activities performed by the host CPU.

For each operation in one of the streams, a slot in the host timeline marks the time of insertion whereas a slot in the stream shows the time of execution by the GPU. A solid line associates the insertion with the execution. For example, the host timeline slot for line 10 of the host code inserts the Jacobi kernel (for calculating the top grid points) into `topStream`. Note that the kernel call has to wait for the `resetL2` event so its execution is delayed. If the reset operation for `l2Norm` takes less (more) time, the Jacobi kernel would be executed earlier (later).

A dotted arrow represents how an event allows a waiting operation to complete its execution. For example, a dotted arrow goes from the event `resetL2` to the wait operation on `resetL2` in `topStream`. Soon after the `cudaMemset` operation is complete,

the `resetL2` event is reached, allowing the wait operation on `resetL2` to complete. The completion of the wait operation, in turn, allows the Jacobi kernel call for the top grid points in the same stream to proceed.

A thick dotted line shows how reaching a point in a stream allows the corresponding `cudaStreamSynchronize` function called on the host to return, thereby allowing the host to proceed to subsequent activities. For example, the thick dotted line immediately after the bottom event of `bottomStream` indicates the point in time when `cudaStreamsynchronize` on `bottomStream` can return. Since `cudaStreamSynchronize` function starts execution after this point, it will immediately return and unblock the halo exchange for the bottom grid points.

An important observation from Fig. 23.15 is that host synchronization can cause significant latency between the completion of a compute operation, e.g., the Jacobi kernel in the Top Stream, and dependent data transfer operations, e.g., MPI Exchange Top by the host. This extra latency is commonly referred to as synchronization overhead and can significantly impact the achieved overlap between operations, e.g., the Jacobi kernel in the Internal Stream and the MPI Exchange operations in the Host. For example, because the MPI Exchange call is blocking, the host's stream synchronization for the Bottom Stream and MPI Exchange Bottom cannot start until the MPI Exchange Top is complete. It should be clear that this blocking causes additional latencies between the completion of the Jacobi kernel in the Bottom stream and the MPI Exchange for the bottom cells, thus reducing the amount of overlap between the MPI Exchange operations and the Jacobi kernel in the Internal stream in Fig. 23.15. Reducing such synchronization overhead is an important design goal of NCCL and NVSHMEM, which will be presented in Sections 23.4 and 23.5, respectively.

One important detail to note is that since the host memory pointed to by `l2norm_h` is involved in asynchronous memory operations such as `cudaMemcpyAsync`, it cannot be allocated using the standard host memory allocation methods such as `malloc()`. Instead, it must be allocated in *pinned memory*, sometimes also referred to as *page locked* memory. Pinned memory allocation can be performed using the `cudaMallocHost` function as follows:

```
float* l2norm_h;
cudaMallocHost(&l2norm_h, sizeof(float));
```

To fully understand the concept of pinned memory and why it is needed for asynchronous copies, we first need to know a little more background on the memory management in operating systems.

In a modern computer system, the operating system manages the memory address spaces of many applications simultaneously. Each application has access to a large and logically consecutive address space called a virtual address space. In reality, the system only has a limited amount of physical memory that needs to be shared among all running applications. This sharing is done by partitioning the virtual address spaces of applications into virtual pages and mapping these pages to physical pages in the physical memory. Mapping virtual pages to physical pages achieves multiple goals. One goal is to ensure that processes cannot access each other's pages in

physical memory without explicit sharing arrangements. They can only access the physical pages that are mapped into their own virtual address space. Another goal is to accommodate situations when there is higher demand for memory than the amount of physical memory available. In such cases, only the most actively used virtual pages are mapped to physical pages while the others are placed on disk. As part of this process, the operating system may need to "page out" some of the pages from the physical memory to mass storage such as disks to make room for other pages. Therefore, an application may have its data paged out of physical memory at any time during its execution.

The implementation of `cudaMemcpy()` uses a type of hardware called a *Direct Memory Access* (DMA) device. When a `cudaMemcpy()` function is called to copy between the host and device memories, its implementation uses a DMA to complete the task. On the host memory side, the DMA hardware operates on physical addresses. That is, the operating system needs to give a translated physical address to the DMA. However, there is a chance that the data may be paged out before the DMA operation is complete. The physical memory locations for the data may be re-assigned to other data corresponding to a different virtual memory location. In this case, the DMA operation can be potentially corrupted since its data can be overwritten by the paging activity.

A common solution to this data corruption problem is for the CUDA runtime to perform the copy operation in two steps. For a host-to-device copy, the CUDA runtime first copies the source host memory data into a "pinned" memory buffer, which means that the memory locations are marked so that the paging mechanism will not page out the data. It then uses the DMA device to copy the data from the pinned memory buffer to the device memory. For a device-to-host copy, the CUDA runtime first uses a DMA device to copy the data from the device memory into a pinned memory buffer. It then copies the data from the pinned memory buffer to the destination host memory location. By using an extra pinned memory buffer, the DMA copy will be safe from any paging activities.

There are two problems with this approach. The first problem is that the extra copy adds delay to the `cudaMemcpy()` operation. The second problem is that the extra complexity involved leads to a synchronous implementation of the `cudaMemcpy()` function. That is, the host cannot continue to execute until the `cudaMemcpy()` function completes its operation and returns because the host needs to be involved in the memory copy operation. In order to support asynchronous copies such as `cudaMemcpyAsync()`, we need to eliminate the step where the host copies the data from a regular memory buffer to a pinned memory buffer. For this reason, the `cudaMemcpyAsync()` function requires that the host memory buffer passed to it be already allocated as a pinned memory buffer.

23.4 Multi-GPU stencil with NCCL

The multi-GPU implementation of the Jacobi iterative method described in Section 23.3 is effective at overlapping computation and communication by using concurrent asynchronous kernel calls, memory copies, and CUDA streams. However,

the implementation still has some synchronization points that introduce inefficiency. If we look back to the code in Fig. 23.14, we observe that the host thread calls `cudaStreamSynchronize` (lines 27 and 31) before the calls to `MPI_Sendrecv` to ensure that the kernels computing the top and bottom boundary rows are complete before initiating the network communication for the halo exchange. During this time, the `cudaStreamSynchronize` function blocks the host thread from executing any of the subsequent statements in the host code. This need to involve the host thread in the synchronization means that the host cannot be used to perform other useful work.

This inefficiency can be overcome if we free the host thread from the burden of synchronizing between CUDA operations such as kernel calls and communication operations. In other words, if the communication operations themselves could be placed inside the streams, then the stream mechanism can simply be used to initiate the communication operations as soon as the operations that they depend on are complete, and without involving the host thread. This desire to perform the communication operations within the context of the CUDA streams is one motivation behind the NVIDIA Collective Communications Library (NCCL) [2].

NCCL is a library that provides various inter-GPU communication primitives. It supports both point-to-point communication, such as send and receive primitives, and collective communication, such as reduce, broadcast, scatter, and gather primitives. While the communication primitives in NCCL resemble those in MPI, an important difference is that NCCL communication primitives run on the GPU, not the host CPU, and can be placed inside of CUDA streams. Hence, using NCCL frees the host thread from having to serve as a synchronizing intermediary between kernel calls and communication primitives. Another benefit of NCCL communication primitives is that they are highly optimized for multi-GPU communication, both within and across nodes, in a topology-aware manner. NCCL thus frees developers from the burden of optimizing their communication routines for specific systems and interconnects. NCCL supports multiple interconnect technologies such as PCIe, NVLINK, InfiniBand Verbs, and sockets.

NCCL is not a complete replacement for MPI, but rather, a library that can be used in conjunction with MPI to enhance multi-GPU communication. To use NCCL in conjunction with MPI, we must set up and tear down the NCCL communication system similar to how we set up and tear down the MPI communication system. To set up and tear down the NCCL communication system, there are three important NCCL API functions that we need to be aware of: `ncclGetUniqueId`, `ncclCommInitRank`, and `ncclCommDestroy`. The signatures and descriptions of these three functions are shown in Fig. 23.16.

Fig. 23.17 shows how the API functions in Fig. 23.16 are typically used for setting up and tearing down the NCCL communication system in conjunction with doing so for an MPI communication system. The code for setting up and tearing down the MPI communication system is the same as that in Fig. 23.6. Compared to Fig. 23.6, Fig. 23.17 additionally includes a header file to access the NCCL API functions (line 02) and uses the API functions to set up (lines 09-14) and tear down (line 16) the NCCL communication system.

- `ncclResult_t ncclGetUniqueId(ncclUniqueId *uniqueId)`
  - Creates a unique ID to be passed to `ncclCommInitRank` by all ranks in the communicator. `ncclGetUniqueId` should be called once (i.e., by one rank) and the ID should be distributed to all ranks.
  - `uniqueId`: pointer to where the created unique ID should be deposited
- `ncclResult_t ncclCommInitRank(ncclComm_t *comm, int nranks, ncclUniqueId commId, int rank)`
  - Create a new communicator
  - `comm`: pointer to where the created communicator handle should be deposited
  - `nranks`: number of ranks in the communicator
  - `commId`: unique ID created by `ncclGetUniqueId`
  - `rank`: the rank of the calling process in the communicator
- `ncclResult_t ncclCommDestroy(ncclComm_t comm)`
  - Destroys a communicator object
  - `comm`: the communicator object to be destroyed

FIGURE 23.16

Basic NCCL API functions for establishing and closing a communication system.

```
01    #include "mpi.h"
02    #include <nccl.h>
03    int main(int argc, char *argv[]) {
04        MPI_Init(&argc, &argv);
05        int rank;
06        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
07        int numRanks;
08        MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
09        ncclUniqueId ncclID;
10        if(rank == 0) ncclGetUniqueId(&ncclID);
11        MPI_Bcast(&ncclID, sizeof(ncclUniqueId), MPI_BYTE, 0, MPI_COMM_WORLD);
12        MPI_Barrier(MPI_COMM_WORLD);
13        ncclComm_t ncclComm;
14        ncclCommInitRank(&ncclComm, numRanks, ncclID, rank);
15        ...
16        ncclCommDestroy(ncclComm);
17        MPI_Finalize();
18        return 0;
19    }
```

FIGURE 23.17

A simple MPI + NCCL main program.

Similarly to MPI, NCCL communication operations are based on the concept of communicators that specify all the entities involved in each operation. Thus, one must create a NCCL communicator before a group of n CUDA devices can communicate with each other using NCCL. When creating a NCCL communicator, a unique rank between 0 and $n - 1$ has to be assigned to each CUDA device which is part of the communicator. If there is a 1-to-1 mapping between MPI processes/threads and CUDA devices in an application, one can simply use their MPI ranks for the NCCL ranks, which is the case in the example host code in Fig. 23.17.

Given assigned ranks, the next step is to create a unique object which will be used by all processes and threads to synchronize and understand that they are part of the same communicator. This should be done by only one of the ranks calling the `ncclGetUniqueId` function. In Fig. 23.17, rank 0 calls `ncclGetUniqueId` to create a unique ID (line 10). The `MPI_Bcast` function is then used to send this unique object from rank 0 to all other MPI ranks (line 11). Note that all receiving ranks must

- `ncclResult_t ncclSend(const void *sendbuff, size_t count, ncclDataType_t datatype, int peer, ncclComm_t comm, cudaStream_t stream)`
  - Sends data to another rank
  - `sendbuff`: starting address of send buffer
  - `count`: number of elements to be sent
  - `datatype`: datatype of each send buffer element
  - `peer`: rank of destination
  - `comm`: communicator
  - `stream`: CUDA stream for the send operation to be executed in
- `ncclResult_t ncclRecv(void *recvbuff, size_t count, ncclDataType_t datatype, int peer, ncclComm_t comm, cudaStream_t stream)`
  - Receives data from another rank
  - `recvbuff`: starting address of receive buffer
  - `count`: number of elements to be received
  - `datatype`: datatype of each receive buffer element
  - `peer`: rank of the source
  - `comm`: communicator
  - `stream`: CUDA stream for the receive operation to be executed in

FIGURE 23.18

NCCL API functions for sending and receiving data.

also execute the `MPI_Bcast` function to receive the broadcasted object. The reader is referred to the MPI user guide for details of the `MPI_Bcast` function. Briefly, the parameters specify the address of the data to be broadcasted, its size, its type, the rank that is doing the broadcast (rank 0 in this case), and the scope of the broadcast which is all the MPI ranks in the application. A call to `MPI_Barrier` forces all the ranks to wait for each other to receive the ID object before creating the communicator (line 12).

Next, all ranks call `ncclCommInitRank` to intialize the NCCL communicator object in each rank and the communication system (line 14). They all pass the same `ncclID` to `ncclCommInitRank` which registers the participating ranks as part of the same communicator that is replicated into each rank. Finally, at the end of the main function, the call to `ncclCommDestroy` tears down the NCCL communication system (line 16). We only need to use one communicator in our examples, however, interested readers should refer to the NCCL user guide for details on using multiple NCCL communicators and multiple NCCL communication systems in an application.

Having seen how to set up and tear down a NCCL communication system, we now turn our attention to how we can use NCCL primitives, i.e., API functions, instead of `MPI_Sendrecv` for performing the halo exchange. For this purpose, the communication primitives we are particularly interested in are `ncclSend` and `ncclRecv`. The signatures and descriptions of these two functions are shown in Fig. 23.18. Note that the functions resemble the `MPI_Send` and `MPI_Recv` functions, with one major difference. The difference is that they are asynchronous functions that accept a CUDA stream object for the stream in which they should be executed.

Recall from Section 23.2 that we could not use `MPI_Send` and `MPI_Recv` for halo exchange because each rank needs to simultaneously send data to one rank and receive data from another rank. For this reason, we resorted to `MPI_Sendrecv` which fuses the

- `ncclResult_t ncclGroupStart()`
  - Indicates that subsequent calls to NCCL communication primitives before the next call to `ncclGroupEnd` will not block
- `ncclResult_t ncclGroupEnd()`
  - Waits until all communication primitives since the last call to `ncclGroupStart` have been submitted to the specified streams.

FIGURE 23.19

NCCL API functions for grouping communication primitives together.

two operations in one. However, NCCL does not provide a similar function that consists of a fused send and receive operation. Instead, NCCL provides a feature for fusing communication primitives which is *group calls*. In NCCL, a set of communication primitives can be fused together by sandwiching them in between calls to two API functions: `ncclGroupStart` and `ncclGroupEnd`. The descriptions of these two functions are shown in Fig. 23.19. Accordingly, a fused send and receive operation can be achieved by placing calls to `ncclSend` and `ncclRecv` in between calls to `ncclGroupStart` and `ncclGroupEnd`. NCCL primitives will run concurrently on each CUDA device, ensuring that each device will be able to send its boundary grid points to one of its neighbors while simultaneously receiving its halo grid points for the other neighbor.

We are now ready to show how the halo exchange that overlaps computation and communication can be achieved with NCCL. Fig. 23.20 shows the host code for such a NCCL-based implementation. Compared to the MPI-based host code in Fig. 23.14, the main change in Fig. 23.20 is in the halo exchange (lines 25-36). Here, each call to `MPI_Sendrecv` is replaced with a call to `ncclSend` and a call to `ncclRecv` sandwiched between calls to `ncclGroupStart` and `ncclGroupEnd`. Importantly, instead of synchronizing on `topStream` and `bottomStream` before the calls using `cudaStreamSynchronize()`, we pass the streams to `ncclSend` and `ncclRecv` as their last parameter. The CUDA runtime ensures that these communication primitives are initiated after the preceding kernels in the streams finish computing the corresponding halo rows.

An important distinction between the call to `MPI_Sendrecv` and the group call to `ncclSend` and `ncclRecv` is that `MPI_Sendrecv` is blocking. In other words, `MPI_Sendrecv` does not return until the communication is complete. On the other hand, the group call to `ncclSend` and `ncclRecv` is non-blocking. Even after the call to `ncclGroupEnd`, the only guarantee is that `ncclSend` and `ncclRecv` have been inserted into the specified stream. For this reason, we need to take further action to ensure that we do not begin the next iteration before the halo exchange communication is actually complete. This can be done by recording the events `exchangeTop` and `exchangeBottom` in the `topStream` and the `bottomStream` after their respective calls to `ncclSend` and `ncclRecv` (line 31 and 36), and synchronizing on these events in the `internalStream` before proceeding to the next iteration (line 43-44).

Fig. 23.21 illustrates the timelines achieved by the host code in Fig. 23.20 while overlapping computation and communication using NCCL. Compared to the timelines in Fig. 23.15, the key difference is that the halo exchange has moved from being MPI calls in the host timeline to NCCL group calls in the `topStream` and `bottomStream` timelines. Moreover, the synchronization on these two streams has moved from being a stream synchronization in the host timeline to event synchronizations in the

```
01    for(unsigned int iter = 0; l2norm > tol && iter < iter_max; ++iter) {
02
03        // Reset L2 norm
04        cudaMemsetAsync(l2norm_d, 0, sizeof(float), internalStream);
05        cudaEventRecord(resetL2, internalStream);
06        cudaStreamWaitEvent(topStream, resetL2, 0);
07        cudaStreamWaitEvent(bottomStream, resetL2, 0);
08
09        // Compute stencil
10        launch_jacobi_kernel(output, input, l2norm_d, 3, nx, topStream);
11        cudaEventRecord(computeTop, topStream);
12        launch_jacobi_kernel(output + (ny - 3)*nx, input + (ny - 3)*nx, l2norm_d, 3,
13                             nx, bottomStream);
14        cudaEventRecord(computeBottom, bottomStream);
15        launch_jacobi_kernel(output + nx, input + nx, l2norm_d, ny - 2, nx,
16                             internalStream);
17
18        // Copy L2 norm to host
19        cudaStreamWaitEvent(internalStream, computeTop, 0);
20        cudaStreamWaitEvent(internalStream, computeBottom, 0);
21        cudaMemcpyAsync(l2norm_h, l2norm_d, sizeof(float), cudaMemcpyDeviceToHost,
22                        internalStream);
23
24        // Halo exchange
25        const int top = rank > 0 ? rank - 1 : (numRanks - 1);
26        const int bottom = (rank + 1) % numRanks;
27        ncclGroupStart();
28        ncclSend(output + nx,          nx, ncclFloat, top,    nccl_comm, topStream);
29        ncclRecv(output + (ny - 1)*nx, nx, ncclFloat, bottom, nccl_comm, topStream);
30        ncclGroupEnd();
31        cudaEventRecord(exchangeTop, topStream);
32        ncclGroupStart();
33        ncclSend(output + (ny - 2)*nx, nx, ncclFloat, bottom, nccl_comm, bottomStream);
34        ncclRecv(output,               nx, ncclFloat, top,    nccl_comm, bottomStream);
35        ncclGroupEnd();
36        cudaEventRecord(exchangeBottom, bottomStream);
37
38        // Reduce L2 norm
39        cudaStreamSynchronize(internalStream);
40        MPI_Allreduce(l2norm_h, &l2norm, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
41        l2norm = std::sqrt(l2norm);
42
43        cudaStreamWaitEvent(internalStream, exchangeTop, 0);
44        cudaStreamWaitEvent(internalStream, exchangeBottom, 0);
45
46        std::swap(output, input);
47
48    }
```

FIGURE 23.20

Host code for NCCL-based multi-GPU implementation of the Jacobi iterative method that overlaps computation and communication.

`internalStream` timeline. Consequently, the host is now free from the communication and synchronization operations, except for the synchronization on the internal stream before the call to `MPI_Allreduce`.

23.5 Multi-GPU stencil with NVSHMEM

So far, we have seen how to use MPI and NCCL to perform data communication for the purpose of halo exchange. The style of communication used by both MPI and

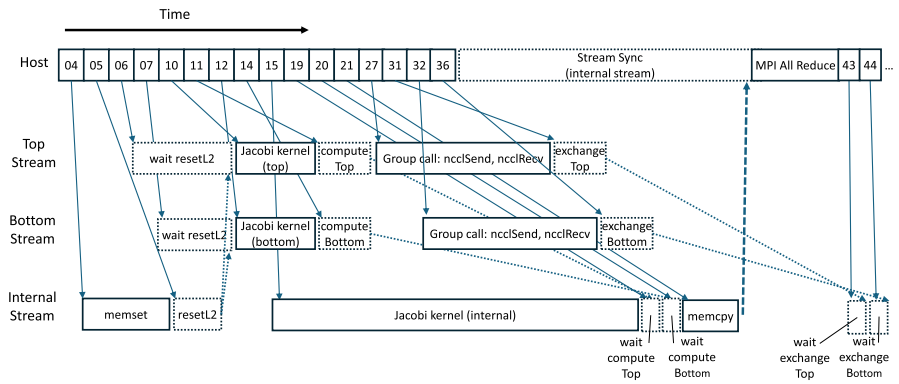


FIGURE 23.21

An example timeline for overlapping computation and NCCL communication using CUDA streams and events.

NCCL is referred to as *two-sided communication*. The communication is two-sided because for a process to send data to or receive data from another process, the two processes have to both be executing corresponding primitives. In the case of MPI, the call to `MPI_Sendrecv` in the sending process must have a corresponding call to `MPI_Sendrecv` in the receiving process. In the case of NCCL, the call to `ncclSend` in the sending process must have a corresponding call to `ncclRecv` in the receiving process. The reason that both processes have to be involved is that the sending process specifies where the communicated data is coming from, whereas the receiving process specifies where the communicated data should be placed after being received. During actual execution, the sending and receiving processes need to wait for each other to reach these calls to initiate the communication, which incurs additional latency.

An alternative form of communication that is commonly used in parallel computing is *one-sided communication*. In one-sided communication, a process can send data to or receive data from another process without involving the other process. The process directly accesses the other process' address space with *put* operations that store data to the other process' address space, or *get* operations that load data from the other process' address space. The ability to directly access parts of another process' address space without waiting for the other process to become available reduces the latency of data communication. It also enables the communication to be initiated by threads executing on the device since matching send and receive calls is not scalable to hundreds of thousands of threads executing on the device, but one-sided put and get operations are.

A commonly used library for one-sided communication is OpenSHMEM [9]. NVIDIA provides an implementation of OpenSHMEM for clusters of GPUs called the NVIDIA OpenSHMEM Library (NVSHMEM) [3]. NVSHMEM allows put and get operations to be initiated by the host threads or by individual device threads of a

- `void* nvshmem_malloc(size_t size)`
  - Allocates at least `size` bytes in the symmetric heap and returns the address of the allocated bytes
- `void nvshmem_free(void *ptr)`
  - Deallocates the data pointed to by `ptr` from the symmetric heap
- `__device__ float nvshmem_float_g(const float* source, int pe)`
  - Gets a value at the symmetric address `source` in the memory of processing element pe and returns it
- `__device__ void nvshmem_float_p(float* dest, float val, int pe)`
  - Puts a value `val` at the symmetric address `dest` in the memory of processing element pe

FIGURE 23.22

NVSHMEM API functions for allocating, putting, and getting data in the symmetric heaps.

kernel. That is, any thread running on the CPU or GPU in one process can directly load or store data to part of the memory space of another GPU in another process. In NVSHMEM terminology, a process or rank is referred to as a *processing element* (PE).

The reader may wonder how threads in a process initiating a put or get operation can know the address of the accessed data object in the target process' address space. In two-sided communication, this address is specified by the matching call executed by the target process, whereas in one-sided communication, the initiating process needs to specify it. To enable the initiating process to specify addresses in the target process' address space, a *symmetric heap* is used. Each process allocates a portion of its address space to locally accommodate the symmetric heap, and all processes allocate corresponding memory objects within their address spaces at the same offset in their respective local instances of the symmetric heap. Hence, for an initiating process to access an object in a target process' address space, it can simply provide the address of the corresponding object in its own address space. The runtime can then infer the object's offset, and consequently, the address of the object in the target process' address space.

Fig. 23.22 shows the NVSHMEM API functions for allocating objects in the symmetric heap (`nvshmem_malloc` and `nvshmem_free`) and accessing them from within kernels (`nvshmem_float_g` and `nvshmem_float_p`). The following code shows an example of how `nvshmem_malloc` can be used to perform symmetric memory allocation of the arrays holding the input and output grid point values:

```
input  = (float*) nvshmem_malloc(nx*ny*sizeof(float));
output = (float*) nvshmem_malloc(nx*ny*sizeof(float));
```

The following code shows how `nvshmem_free` can subsequently be used to free these arrays:

```
nvshmem_free(input);
nvshmem_free(output);
```

All PEs must participate when calling these routines. Note that all PEs only allocate their own partition of the input and output grids. The assumption is that the layout of

```
01    #include <nvshmem.h>
02    __global__ void jacobi_kernel(float* out, float* in, float* l2norm,
03        unsigned int ny, unsigned int nx, int top, int bottom) {
04        int y = blockIdx.y*blockDim.y + threadIdx.y + 1;
05        int x = blockIdx.x*blockDim.x + threadIdx.x + 1;
06        // Compute stencil and residue
07        float residue = 0.0f;
08        if (y < ny - 1 && x < nx - 1) {
09            float val = 0.25*(in[y*nx + (x + 1)]
10                            + in[y*nx + (x - 1)]
11                            + in[(y + 1)*nx + x]
12                            + in[(y - 1)*nx + x]);
13            out[y*nx + x] = val;
14            residue = val - in[y*nx + x];
15            if (y == 1) {
16                nvshmem_float_p(out + (ny - 1)*nx + x, val, top);
17            }
18            if (y == ny - 2) {
19                nvshmem_float_p(out + x, val, bottom);
20            }
21        }
22        // Reduce L2 norm
23        float blockL2norm = blockReduce(residue*residue);
24        if (threadIdx.y == 0 && threadIdx.x == 0) {
25            atomicAdd(l2norm, blockL2norm);
26        }
27    }
```

FIGURE 23.23

Stencil kernel for the Jacobi iterative method that uses NVSHMEM put functions for halo exchange.

the partitions, i.e., the offsets of the top and bottom grid points, is symmetrical, i.e., identical, across all partitions.

Fig. 23.23 shows how the kernel in Fig. 23.1 can be modified to use NVSHMEM to exchange halo values within the kernel. The key changes from Fig. 23.1 are the inclusion of the NVSHMEM library header file (line 01), the passing of the top and bottom PE's IDs as parameters to the kernel (line 03), and the calls to `nvshmem_float_p` to send the boundary values to the top and bottom PEs (lines 15-20). In particular, each thread checks if it is computing a value in the top boundary row of its PE, i.e., if `y == 1` (line 15). If so, the thread uses `nvshmem_float_p` to send that value to its top PE, passing the value as the second parameter and the ID of the top PE as the third parameter (line 16). Since this value needs to be placed in the bottom halo row of the top PE, the thread passes the address of the symmetric halo element in its own bottom halo row as the first parameter. Similarly, each thread checks if it is computing a value in the bottom boundary row of its PE, i.e., if `y == ny - 2` (line 18). If so, the thread uses `nvshmem_float_p` to send that value to its bottom PE, passing the value as the second parameter and the ID of the bottom PE as the third parameter (line 19). Since this value needs to be placed in the top halo row of the bottom PE, the thread passes the address of the symmetric halo element in its own top halo row as the first parameter.

```
01    #include "mpi.h"
02    #include <nvshmem.h>
03    #include <nvshmemx.h>
04    int main(int argc, char *argv[]) {
05        MPI_Init(&argc, &argv);
06        int rank;
07        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
08        int numRanks;
09        MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
10        nvshmemx_init_attr_t attr;
11        MPI_Comm mpi_comm = MPI_COMM_WORLD;
12        attr.mpi_comm= &mpi_comm;
13        nvshmemx_init_attr(NVSHMEMX_INIT_WITH_MPI_COMM, &attr);
14        ...
15        nvshmem_finalize();
16        MPI_Finalize();
17        return 0;
18    }
```

FIGURE 23.24

A simple MPI + NVSHMEM main program.

Having seen how to implement a kernel that uses NVSHMEM, we now turn our attention to the host code that calls this kernel. Similarly to NCCL, NVSHMEM can be used in conjunction with MPI. Fig. 23.24 shows the typical host code for setting up NVSHMEM in conjunction with MPI. The code for setting up and tearing down the MPI communication system is the same as that in Fig. 23.6. Compared to Fig. 23.6, Fig. 23.24 additionally includes header files to access the NVSHMEM API functions (lines 02-03) and uses these API functions to allocate and initialize (lines 10-13) as well as release (line 15) the resources needed by the NVSHMEM library.

In particular, an object of type `nvshmemx_init_attr_t` is created (line 10) to provide configuration details for initializing the NVSHMEM communication system. Interested readers are referred to the NVSHMEM user guide for all the configuration attribute fields supported by the type. In our example, we focus on one of the configuration attribute fields, `mpi_comm`, that points to an MPI communicator object specifying the scope of the processes involved in the NVSHMEM communication. In our stencil example, the scope is all the ranks of the MPI communication system (lines 11-12). The `nvshmemx_init_attr` function is then called to initialize the NVSHMEM library resources using the object, which is passed as the second parameter (line 13). The first parameter, `NVSHMEMX_INIT_WITH_MPI_COMM`, indicates that we are initializing NVSHMEM using an MPI communicator. Finally, after the host code completes the Jacobi method computation, it calls `nvshmem_finalize` releases the NVSHMEM library resources (line 15). Note that NVSHMEM can also be initialized without an MPI communicator. We refer the reader to the NVSHMEM documentation [3] for instructions on how to initialize NVSHMEM when not used in conjunction with MPI.

Fig. 23.25 shows the host code for the main loop that executes the Jacobi iterative method using the NVSHMEM-based kernel. The code for resetting the L2 norm (line 04) and reducing it (lines 12-16) is the same as previous examples. The main difference is in the code for launching the compute kernel and performing the halo exchange (lines 07-09). In contrast with the MPI example in Fig. 23.14 and the NCCL

```
01    for(unsigned int iter = 0; l2norm > tol && iter < iter_max; ++iter) {
02
03        // Reset L2 norm
04        cudaMemsetAsync(l2norm_d, 0, sizeof(float), stream);
05
06        // Compute stencil and exchange halos
07        const int top = PE > 0 ? PE - 1 : (numPEs - 1);
08        const int bottom = (PE + 1) % numPEs;
09        launch_jacobi_kernel(output, input, l2norm_d, ny, nx, top, bottom, stream);
10
11        // Copy and reduce L2 norm
12        cudaMemcpyAsync(l2norm_h, l2norm_d, sizeof(float), cudaMemcpyDeviceToHost,
13                        stream);
14        cudaStreamSynchronize(stream);
15        MPI_Allreduce(l2norm_h, &l2norm, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
16        l2norm = std::sqrt(l2norm);
17
18        nvshmemx_barrier_all_on_stream(stream);
19
20        std::swap(output, input);
21
22    }
```

FIGURE 23.25

Host code for NVSHMEM-based multi-GPU implementation of the Jacobi iterative method that overlaps computation and communication.

example in Fig. 23.20 where three different kernels are launched into three different streams, only a single kernel is launched into a single stream called `stream` to compute the entire partition (line 09). The kernel takes two additional parameters, `top` and `bottom` to tell the kernel to which PEs to send the boundary values. The values of `top` and `bottom` are computed (lines 07-08) in the same manner as the previous examples. We comment on the implications of launching only one kernel for both the computation and communication in each Jacobi iteration later in this section.

After inserting the kernel that performs the computation and the halo exchange into the stream, we need to guarantee that the boundary values have arrived at the target process before proceeding to the next iteration. In the MPI-based implementation in Fig. 23.14, this guarantee comes from the fact that the calls to `MPI_Sendrecv` are blocking, which means they do not return until the data have arrived. Therefore, by the time all processes have reached the `MPI_Allreduce` in the timeline in Fig. 23.15, we know for sure that all halo exchange data have been received.

In the NCCL-based implementation in Fig. 23.20, this guarantee comes from the fact that we synchronize on the `ncclSend` and `ncclRecv` events in the appropriate streams. The `cudaStreamWaitEvent()` calls before the end of the iteration guarantee that the halo exchange data have arrived for each process before the L2 norm is reset for the next iteration.

In the NVSHMEM-based implementation in Fig. 23.25, the boundary values are sent by the same kernel that calculates the grid points. The kernel will complete as soon as all the launched threads complete their work in computing the grid point values and sending the halo data. Synchronizing on the stream only guarantees that the kernel has finished its computation and sending the data; it does not guarantee that the data has been received. To ensure that data have been received, we need to call

`nvshmemx_barrier_all_on_stream`, the NVSHMEM API function that provides such guarantee, on the same stream as the kernel (line 18). This call makes all PEs wait at the barrier until all the NVSHMEM memory access operations initiated by any preceding kernels in the stream have completed. Therefore, when all the processes proceed to the next iteration, we know for sure that all halo exchange data have been received and it is safe to proceed.

Using NVSHMEM to perform put operations of the halo values directly from the kernel has two key advantages. The first advantage is that it automatically overlaps computation and communication. With MPI and NCCL, the kernel needs to be terminated before the communication could be started. For this reason, we have to separate the computation of the boundary values and the internal values into different kernels so that we can wait for only the kernel computing the boundary values before initiating the communication in parallel with the kernel computing the internal values. In NVSHMEM, it is unnecessary to launch separate kernels and orchestrate the concurrency between them. NVSHMEM allows the halo values to be sent from within some of the GPU threads using the put operation as soon as they are computed, and allows other threads to continue to compute the internal grid point values while the communication initiated by the put operation happens in the background.

The second advantage of performing put and get operations directly from within the kernel is that it allows multiple operations to be fused into a single kernel. This advantage is less apparent with our Jacobi iterative method example because there is only one operation, i.e., the stencil operation. However, if the computation consisted of multiple computations with different communications happening in between these computations, then MPI and NCCL would require each of these computations to be executed by separate kernels. On the other hand, NVSHMEM would enable fusing these operations in the same kernel and communicating from within the kernel. Such kernel fusion could reduce the overhead of kernel launches for small kernels, as well as enable keeping data in shared memory and registers if this data is used across the different operations being fused.

One subject for potential further study is that the fine-grain manner in which we have used the put operations in the kernel in Fig. 23.23 would perform reasonably well for GPUs connected over NVLINK, which supports low latency and high bandwidth communication. However, for slower interconnects and communication over a network, it would be more efficient to perform fewer and more coarse-grain put operations. NVSHMEM provides an alternative function, `nvshmemx_float_put_block` where threads in the same thread block can collaborate to execute one coarse-grain put operation. We refer the reader to the NVSHMEM documentation for more information on how to use such more advanced techniques [3].

23.6 Summary

We have covered basic patterns of multi-GPU programming for clusters of GPUs using three programming models: MPI, NCCL, and NVSHMEM. We use the stencil

pattern to show how multiple processes exchange data throughout the execution. Our first implementation uses MPI alone and has each process compute all grid point values in its partition then send the boundary values to neighboring processes after all the computation is done. We then improve on this MPI implementation by computing the boundary values first, then sending the boundary values to neighboring processing while computing the internal grid point values, thereby overlapping computation and communication. Achieving this overlap required the use of CUDA streams to execute different kernels concurrently.

We then looked at how the same overlap between computation and communication can be achieved with using NCCL in conjunction with MPI. The advantage of NCCL is that it allows communication operations to be placed inside of CUDA streams, freeing the host from the burden of synchronizing between GPU kernels and MPI communication primitives. Furthermore, the NCCL library is well-optimized for multi-GPU communication, freeing the programmer from the burden of customizing their communication primitives to the underlying system organization and network topology.

After that, we looked at how the same overlap between computation and communication can be achieved with NVSHMEM. Unlike MPI and NCCL which rely on two-sided communication, NVSHMEM relies on one-sided communication, which removes the latency of send operations needed to match with corresponding receive operations. Moreover, NVSHMEM put and get operations can be executed from within GPU kernels, which allows kernels to be fused and achieves the overlap between computation and communication transparently.

Our comparison of NCCL and NVSHMEM focused on contrasting the host-initiated two-sided communication in NCCL with the device-initiated one-sided communication in NVSHMEM. However, it is worth noting that these multi-GPU programming APIs continue to evolve and have more and more features in common. For example, NCCL recently added support for symmetric memory and device-initiated communication, and may add support for one-sided communication soon. On the other hand, NVSHMEM supports host-initiated communication as well, not just device-initiated communication. Host-initiated communication works well when the data to be communicated are of coarse granularity and can be determined independently of the computation that overlaps with the communication. On the other hand, device-initiated communication is preferred when the data to be communicated are of fine granularity and are determined as part of the computation that overlaps with the communication.

While multi-GPU programming is very different from programming GPU kernels, the major programming concepts that we covered (SPMD, ranks, and barriers) all have counterparts in the CUDA programming model. This confirms our belief that by teaching parallel programming with one model well, students can pick up other programming models quickly and easily. We encourage the reader to build on the foundation from this chapter and study more advanced features of MPI, NCCL, and NVSHMEM, and apply them to other important patterns.

23.7 Exercises

1. Assume that the 5-point stencil computation in this chapter is applied to a grid whose size is 64 grid points in the x-dimension and 512 grid points in the y-dimension. The computation is divided across 16 MPI ranks.
   a. How many output grid point values are computed by each process?
   b. How many halo grid points are needed by each process?
   c. How many boundary grid points are computed by each process in Stage 1 of Fig. 23.12?
   d. How many internal grid points are computed by each process in Stage 2 of Fig. 23.12?
   e. How many bytes are sent by each process in Stage 2 of Fig. 23.12?
2. If the MPI call `MPI_Send(ptr_a, 1000, MPI_FLOAT, 2000, 4, MPI_COMM_WORLD)` resulted in a data transfer of 4,000 bytes, what is the size of each data element being sent?
   a. 1 byte
   b. 2 bytes
   c. 4 bytes
   d. 8 bytes
3. Which of the following statements is true?
   a. `MPI_Send()` is blocking by default.
   b. `MPI_Recv()` is blocking by default.
   c. MPI messages must be at least 128 bytes.

References

[1] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, vol. 1, MIT Press, 1999.
[2] NVIDIA, Overview of NCCL, https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html, 2024.
[3] NVIDIA, NVIDIA OpenSHMEM library (NVSHMEM) documentation, https://docs.nvidia.com/nvshmem/api/index.html, 2024.
[4] NVIDIA, Multi GPU programming models, https://github.com/NVIDIA/multi-gpu-programming-models, 2025.
[5] NVIDIA, CUB, https://nvidia.github.io/cccl/cub/index.html, 2025.
[6] MPICH, MPICH, https://www.mpich.org, 2025.
[7] OpenMPI, Open MPI: open source high performance computing, https://www.open-mpi.org, 2025.
[8] MVAPICH, MVAPICH: MPI over InfiniBand, omni-path, Ethernet/iWARP, RoCE, and slingshot, https://mvapich.cse.ohio-state.edu, 2025.
[9] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, L. Smith, Introducing openshmem: shmem for the pgas community, in: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, 2010, pp. 1–3.

CHAPTER

Conclusion and future outlook

24

CONTENTS

24.1 Goals revisited 577
24.2 Future outlook 578

You made it! We have arrived at the finishing line. In this final chapter, we will briefly review the learning goals that you have achieved through this book. Instead of drawing a conclusion, we will offer our vision for the future of massively parallel computing and how its advancements will impact the future course of science and technology.

24.1 Goals revisited

As we stated in the Introduction, our primary goal is to teach you, the reader, how to program massively parallel processors. We promised that it would become easy once you develop the right intuition and go about it the right way. In particular, we promised to focus on computational thinking skills that would enable you to think about problems in ways that are amenable to parallel computing. We also promised to focus on the factors that limit the performance of parallel applications and provide a systematic approach for optimizing code to overcome these factors.

We delivered on these promises through three steps. In step one, Chapter 2 to Chapter 6 introduced the essential concepts of parallel computing and CUDA C++, and the key performance considerations in developing massively parallel code in CUDA. They also introduce the pertinent computer architecture concepts needed to understand the hardware limitations that must be addressed in high-performance parallel programming. With this knowledge, a developer can be confident in writing their parallel code and reasoning about the relative merit of alternative threading arrangements, loop structures, data arrangements, and coding styles. We conclude this part with a checklist of common optimization techniques that we apply throughout the rest of the book to optimize the performance of various parallel patterns and applications.

In the second step, we introduce major parallel patterns (Chapter 7 to Chapter 15) that have been proven useful in introducing parallelism into many applications. These chapters cover the concepts behind the most useful patterns of parallel computation.

Programming Massively Parallel Processors. https://doi.org/10.1016/B978-0-44-343900-1.00035-4
Copyright © 2027 Elsevier Inc. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

	G80 (2007)	B200 (2025)	B200/G80 Ratio
Compute Throughput	0.345 TFLOPS (FP32)	80 TFLOPS / 2200 TFLOPS (FP32 / Tensor Core FP32)	232x / 638x
Memory Bandwidth	86.4 GB/s	8,000 GB/s	93x
Memory Capacity	1.5GB	192 GB	128x
PCIe Bandwidth	8 GB/s (Gen2 x16, each way)	64 GB/s (Gen5 x16, each way)	8x
NVLink Bandwidth	N/A	1,800 TB/s	-

FIGURE 24.1

From G80 to B200, a 18-year comparison.

Each pattern is illustrated with concrete code examples. Each pattern is also used to introduce important techniques for overcoming frequently encountered parallelization and performance obstacles in parallel programming. We also use these patterns to showcase how the optimizations introduced in the first step can be applied in a wide variety of scenarios.

In the third step, we introduce additional advanced patterns and applications (Chapter 16 to Chapter 23) to reinforce the knowledge and skills gained in the previous step. While in this step, we continue to apply the optimizations that we practiced in the previous step, we put more emphasis on exploring alternative forms of problem decomposition to enable parallelization and analyze the trade-offs between different decompositions and their associated data structures. Chapter 22 in this step is dedicated to recap the computational thinking skills that help the reader to generalize the concepts learned in the previous chapters into the high-level thinking required to tackle a new problem. With these insights, high-performance parallel programming becomes a reasoning process, rather than a black art.

We hope that you have enjoyed the book and agree with us that you are now well equipped for programming massively parallel computing systems.

24.2 Future outlook

Since the introduction of the first CUDA-enabled GPU, the G80, in 2007, the capability of GPUs as massively computing devices has improved at an amazing 638× in computing throughput and 93× in memory bandwidth, as shown in Fig. 24.1. These advancements have stimulated tremendous progress in HPC, AI, and data analytics for both science and engineering, with significant impact on many vertical areas such as finance, manufacturing, and medicine. For example, as we have seen in Chapter 19 and Chapter 20, GPUs have ignited a revolution in deep learning from very large data sets, with applications in image recognition, speech recognition, and video analytics.

While the performance growth of individual GPUs has been extremely impressive, the number of GPUs that collaborate through network communication to complete a single job has been increasing at an even more impressive rate since 2017. The extreme computation complexity for training Large Language Models has motivated leading tech companies to use tens or even hundreds of thousands of GPUs in each training job. These leading tech companies have been referred to as the *hyperscalers* because of the extreme scale of their data centers. This explosive growth of the number of GPUs that work together in data centers has driven fast advancement of networking hardware and software. For example, we introduced NCCL in Chapter 23, a new communication library that has been used extensively in training LLMs in hyperscaler data centers.

Since the first edition of this book in 2010, the field of parallel computing has also advanced at an amazing pace. The spectrum of problems that can be solved with scalable algorithms has broadened significantly. While the use of GPUs was initially concentrated on regular dense matrix computation and Monte Carlo methods, their use has quickly expanded into sparse methods and graph computation. In many areas, there has also been fast advancement in algorithms. Some of the algorithms presented in the parallel pattern, advanced patterns, and applications chapters represent significant recent advancements.

It is only natural for some to wonder if we have reached the end of the fast advancement in parallel computing. From all indications, the answer is a definitely no. We are most likely in the middle of the parallel computing revolution. The amazing advancement in computing in the past three decades has triggered a paradigm shift in the industry. The major innovations used to be driven by physical instruments assisted by computing devices. They are now driven by computing assisted by physical instruments.

For example, two decades ago, GPS revolutionized the way we drive. GPS is primarily based on satellite signal sensing assisted by computing methods that determine the shortest path between two locations. More recently, peer reporting and computing through the map apps in the phones and data services in the cloud have further enabled drivers to change their routes based on the traffic condition. Today, the most exciting revolution in the automobile industry is self-driving cars, which is primarily based on machine-learning computing methods assisted by physical sensors.

For another example, MRI and PET revolutionized medicine in the past two decades. These technologies are primarily based on electromagnetic and light sensors assisted by computational image reconstruction methods. They allowed doctors to see the pathology inside human bodies without surgery. Today, the field of medicine is going through the revolution of individualized medicine, which is primarily driven by deep learning models and computational genomics methods assisted by medical and genomic instruments.

For yet another example, the semiconductor industry used to rely on advancement in physical light sources assisted by computing methods that enforce design rules in their push to reduce the device feature size in the manufacturing process. Today, the advancement in physical light sources has practically stopped. The advancement in

feature size reduction is primarily driven by lithography masks that are computationally designed to orchestrate the interference of light waves to result in extremely precise etching patterns on the chips.

The same kind of paradigm shift has been taking place in many other areas such as content authoring, drug discovery, and weather/climate forecasting. Computing has become the primary driving force for virtually all exciting innovations in our society. This has created an insatiable demand for faster computing systems. As we discussed in Chapter 1, parallel computing is the only viable approach to the growth of computing performance. This powerful demand will continue to motivate the industry to innovate and create more powerful parallel computing devices. One of the highest potential areas for improvement is the level of parallelism in accessing storage data, which has been done with a very low level of parallelism in history. This advancement can accelerate databases and other data-intensive applications that have been lagging in terms of innovation. Radical improvements in the precision and speed in extracting relevant portions of massive storage data will likely inspire a whole generation of applications that we currently cannot even imagine.

In conclusion, we are in the middle of a golden age of computing. The industry will continue to recruit and reward highly skilled parallel programmers. Your work will make a real difference in the field of your choice.

Enjoy the ride!

PART

Appendices

4

This page intentionally left blank

APPENDIX

Numerical considerations A

With special contributions from Federico Busato

In the early days of computing, floating-point arithmetic capability was found only in mainframes and supercomputers. Although many microprocessors designed in the 1980's started to have floating-point coprocessors, their floating-point arithmetic speed was extremely slow, about three orders of magnitude slower than that of mainframes and supercomputers. With advances in microprocessor technology, many microprocessors designed in the 1990's, such as Intel Pentium III and AMD Athlon, started to have high performance floating-point capabilities that rival supercomputers. High speed floating-point arithmetic has become a standard feature for microprocessors and GPUs today. More recently, in response to the unsatisfiable demand for high-speed floating-point arithmetic from LLM training and inference use cases, GPUs have dramatically increased the throughput of their floating-point throughput. Floating-point representation allows for larger dynamic range of representable data values and more precise representation of tiny data values than its integer counterpart. These desirable properties make floating-point preferred data representative for modeling the physical and artificial phenomena, such as combustion, aerodynamics, light illumination, and financial risks [1]. Large scale evaluation of these models has been driving the need for parallel computing. As a result, it is important for application programmers to understand the nature of floating-point arithmetic in developing their parallel applications. In particular, we will focus on the accuracy of floating-point arithmetic operations, the precision of floating-point number representation, the stability of numerical algorithms and how they should be taken into consideration in parallel programming.

A.1 Floating-point data representation

The IEEE-754 Floating-Point Standard is an effort for the computer manufacturers to conform to a common representation and arithmetic behavior for floating-point data [2]. Most, if not all, of the computer manufacturers in the world have accepted this standard. In particular, virtually all microprocessors designed in the future will either fully conform to or almost fully conform to the IEEE-754 Floating-Point Standard and its more recent IEEE-754 2019 revision [2].[1] Therefore, it is important for

[1] There are emerging alternatives to the IEEE Floating-Point Standard, especially in the context of AI. For example, the OCP Microscaling Formats (MX) have been adopted by recent GPUs [3].

application developers to understand the concepts and practical considerations of this standard.

A floating-point number system starts with the representation of a numerical value as bit patterns. In the IEEE Floating-Point Standard, a numerical value is represented in three groups of bits: sign (S), exponent (E), and mantissa (M). With some exceptions that will be detailed later, each (S, E, M) pattern uniquely identifies a numeric value according to the following formula:

$$value = (-1)^S \cdot 1.M \cdot 2^{E-bias} \tag{A.1}$$

The interpretation of S is simple: S = 0 means a positive number and S = 1 a negative number. Mathematically, any number, including −1, when raised to the power of 0, results in 1. Thus the value is positive. On the other hand, when −1 is raised to the power of 1, it is −1 itself. With a multiplication by −1, the value becomes negative. The interpretation of M and E bits are, however, much more complex. We will use the following example to help explain the interpretation of M and E bits.

Assume for the sake of simplicity that each floating-point number consists of a 1-bit sign, 3-bit exponent, and 2-bit mantissa. We will use this hypothetical 6-bit format to illustrate the challenges involved in encoding E and M. As we discuss numeric values, we will sometimes need to express a number either in decimal place value or in binary place value. Numbers expressed in decimal place value will have subscript D and those as binary place value will have subscript B. For example, 0.5_D is equal to $5 * 10^{-1}$ since the place to the right of "." in a decimal system carries a weight of 10^{-1}. The same number can be expressed as 0.1_B which is equal to $1 * 2^{-1}$ since the place to the right of "." carries a weight of 2^{-1}.

Normalized representation of M

Eq. (A.1) requires that all values are derived by treating the mantissa value as 1.M, which makes the mantissa bit pattern for each floating point number unique. For example, under this interpretation of the M bits, the only mantissa bit pattern allowed for 0.5_D is the one where all bits that represent M are 0s:

$$0.5_D = 1.0_B \cdot 2^{-1}$$

Other potential candidates would be $0.1_B * 2^0$ and $10.0_B * 2^{-2}$, but neither fits the form of 1.M. The numbers that satisfy this restriction will be referred to as normalized numbers. Because all mantissa values that satisfy the restriction are of the form 1.XX, we can omit the "1." part from the representation. Therefore, the mantissa value of 0.5 in a 2-bit mantissa representation is 00, which is derived by omitting "1." from 1.00. This makes a 2-bit mantissa effectively a 3-bit mantissa. In general, with IEEE format, an m-bit mantissa is effectively an (m + 1)-bit mantissa.

Excess encoding of E

The number of bits used to represent E determines the range of numbers that can be represented. Large positive E values result in very large floating-point absolute

2's complement	Decimal value	Excess-3
101	-3	000
110	-2	001
111	-1	010
000	0	011
001	1	100
010	2	101
011	3	110
100	Reserved pattern	111

FIGURE A.1

Excess-3 encoding, sorted by excess-3 ordering.

values. For example, if the value of E is 64, the floating point number being represented is between 2^{64} (which is greater than 10^{18}) and 2^{65}. You would be extremely happy if this was the balance of your savings account! Large negative E values result in very small floating values. For example, if E value is -64, the number being represented is between 2^{-64} (which is smaller than 10^{-18}) and 2^{-63}. This is a very tiny fractional number. The E field allows a floating-point number format to represent a wider range of numbers than integer number formats. We will come back to this point when we look at the representable numbers of a format.

The IEEE standard adopts an excess or biased encoding convention for E. If e bits are used to represent the exponent E, $2^{e-1} - 1$ is added to the two's complement representation for the exponent to form its excess representation. A two's complement representation is a system where the negative value of a number can be derived by first complementing every bit of the value and adding one to the result. In our 3-bit exponent representation, there are three bits in the exponent ($e = 3$). Therefore, the value $2^{3-1} - 1 = 011$ will be added to the 2's complement representation of the exponent value.

Excess representation allows an unsigned comparator to be used to compare two exponents as if they were unsigned numbers. As shown in Fig. A.1, in our 3-bit exponent representation, the excess-3 bit patterns increase monotonically from -3 to 3 when view numbers. We will refer to each of these bit patterns as the code for the corresponding value. For example, the code for -3 is 000 and that for 3 is 110. Thus, if one uses an unsigned number comparator to compare the excess-3 codes of any pair of exponent values from -3 to 3, the comparator gives the correct comparison result in terms of which number is larger, smaller, etc. For another example, if one compares excess-3 codes 001 and 100 with an unsigned comparator, 001 is smaller than 100. This is the right conclusion, since the values that they represent, -2 and 1, have exactly the same relation.

Fig. A.1 also shows that the pattern of all 1's in the excess representation is a reserved pattern. Note that a 0 value and an equal number of positive and negative values result in an odd number of patterns. Having the pattern 111 as either even number or odd number would result in unbalanced number of positive and negative numbers. The IEEE standard uses this special bit pattern in special ways that will be discussed later.

Now we are ready to represent 0.5D with our 6-bit format:

$$0.5_D = 0\ 010\ 00, \text{ where S} = 0, \text{ E} = 010, \text{ and M} = (1.)00$$

That is, the 6-bit representation for 0.5_D is 001000.

In general, with normalized mantissa and excess-coded exponent, the value of a number represented as (S, E, M) with an n-bit E is:

$$(-1)^S \cdot 1.M \cdot 2^{E-(2^{n-1}-1)}$$

A.2 Representable numbers

The representable numbers of a representation format are the numbers that can be exactly represented in the format. For example, if one uses a 3-bit unsigned integer format, the representable numbers are shown in Fig. A.2.

Neither −1 nor 9 can be represented in the format given above. We can draw a number line to identify all the representable numbers, as shown in Fig. A.3 where all representable numbers of the 3-bit unsigned integer format are marked with stars.

The representable numbers of a floating point format can be visualized in a similar manner. In Fig. A.4, we show all the representable numbers of what we have so far and two variations. We use a 5-bit format to keep the size of the table manageable. The format consists of 1-bit S, 2-bit E (excess-1 coded), and 2-bit M (with "1." part omitted). The no-zero column gives the representable numbers of the format we discussed thus far. The reader is encouraged to generate at least part of the no-zero column using the formula given in Section A.1. Note that with this format, 0 is not one of the representable numbers.

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

FIGURE A.2

Representable numbers of a 3-bit unsigned integer format.

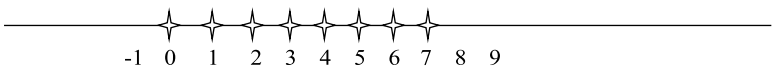


FIGURE A.3

Representable numbers of a 3-bit unsigned integer format.

		No-zero		Abrupt underflow		Denorm	
E	M	S=0	S=1	S=0	S=1	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	0	0	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern						

FIGURE A.4

Representable numbers of no-zero, abrupt underflow, and denorm formats.

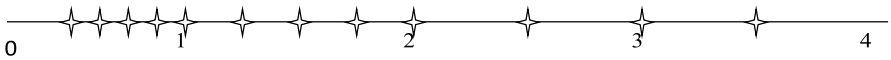


FIGURE A.5

Representable positive numbers of the no-zero representation.

A quick look at how these representable numbers populate the number line, as shown in Fig. A.5, provides further insights about these representable numbers. In Fig. A.5, we show only the positive representable numbers. The negative numbers are symmetric to their positive counterparts on the other side of 0.

We can make five observations. First, the exponent bits define the major intervals of representable numbers. In Fig. A.5, there are three major intervals on each side of 0 because there are two exponent bits. Basically, the major intervals are between powers of 2s. With two bits of exponents and one reserved bit pattern (11), there are three powers of two ($2^{-1} = 0.5_D$, $2^0 = 1.0_D$, $2^1 = 2.0_D$), each starts an interval of representable numbers. Keep in mind that there are also three powers of two ($-2^{-1} = -0.5_D$, $-2^0 = -1.0_D$, $-2^1 = -2.0_D$) on the left side of zero and are not shown in Fig. A.5.

The second observation is that the mantissa bits define the number of representable numbers in each interval. With two mantissa bits, we have four representable numbers in each interval. In general, with N mantissa bits, we have 2^N representable numbers in each interval. If a value to be represented falls between two representable numbers, it will be rounded to one of these representable numbers. Obviously, the larger the number of representable numbers in each interval, the more precisely we can represent a value in the region. Therefore, the number of mantissa bits determines the precision of the representation.

The third observation is that 0 is not representable in this format. It is missing from the representable numbers in the no-zero column of Fig. A.5. Because 0 is one of the most important numbers, not being able to represent 0 in a number representation system is a serious deficiency. We will address this deficiency soon.

The fourth observation is that the representable numbers become closer to each other towards the neighborhood of 0. Each interval is half the size of the previous interval as we move towards zero. In Fig. A.5, the rightmost interval is of width 2, the next one is of width 1, and the next one is of width 0.5. While not shown in Fig. A.5, there are three intervals on the left of zero. They contain the representable negative numbers. The leftmost interval is of width 2, the next one is of width 1 and the next one is width 0.5. Since every interval has the same representable numbers, four in Fig. A.5, the representable numbers become closer to each other as we move towards zero. In other words, the representative numbers become closer as their absolute values become smaller. This is a desirable trend because as the absolute values of these numbers become smaller, it is more important to represent them more precisely. The distance between representable numbers determines the maximal rounding error for a value that falls into the interval. For example, if you have one billion dollars in your bank account, you may not even notice that there is a one dollar rounding error in calculating your balance. However, if the total balance is ten dollars, having a one dollar rounding error would be much more noticeable!

The fifth observation is that, unfortunately, the trend of increasing density of representable numbers and thus increasing precision of representing numbers in the intervals as we move towards zero does not hold for the very vicinity of 0. That is, there is a gap of representable numbers in the immediate vicinity of 0. This is because the range of normalized mantissa precludes 0. This is another serious deficiency. The representation introduces significantly larger (4×) errors when representing numbers between 0 and 0.5 compared to the errors for the larger numbers between 0.5 and 1.0. In general, with m bits in the mantissa, this style of representation would introduce 2^m times more error in the interval closest to zero than the next interval. For numerical methods that rely on accurate detection of convergence conditions based on very small data values, such deficiency can cause instability in execution time and accuracy of results. Furthermore, some algorithms generate small numbers and eventually use them as denominators. The errors in representing these small numbers can be greatly magnified in the division process and cause numerical instability in these algorithms.

One method that can accommodate 0 into a normalized floating-point number system is the abrupt underflow convention, which is illustrated in the second column of Fig. A.4. Whenever E is 0, the number is interpreted as 0. In our 5-bit format, this method takes away eight representable numbers (four positive and four negative) in the vicinity of 0 (between −1.0 and +1.0) and makes them all 0. Due to its simplicity, some mini-computers in the 1980s used abrupt underflow. Even to this day, some arithmetic units that need to operate in high speed still use the abrupt underflow convention. Although this method makes 0 a representable number, it creates an even larger gap between representable numbers in 0's vicinity, as shown in Fig. A.6. It is



FIGURE A.6

Representable numbers of the abrupt underflow format.

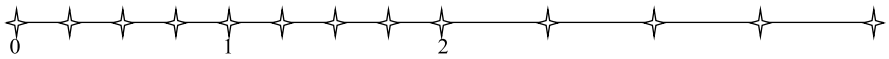


FIGURE A.7

Representable numbers of a denormalization format.

obvious, when compared to Fig. A.5, that the gap of representable numbers has been enlarged significantly (by 2×) from 0.5 to 1.0. As we explained earlier, this gap is problematic for many numerical algorithms whose correctness relies on an accurate representation of small numbers near zero.

The actual method adopted by the IEEE standard is called denormalization. The method relaxes the normalization requirement for numbers very close to 0. As shown in Fig. A.6, whenever E = 0, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. The value of the exponent is assumed to be the same as the previous interval. For example, in Fig. A.4, the denormalized representation 00001 has exponent value 00 and mantissa value 01. The mantissa is assumed to be 0.01 and the exponent value is assumed to be the same as that of the previous interval: 0 rather than −1. That is, the value that 00001 represents is now $0.01 * 2^0 = 2^{-2}$. Fig. A.7 shows the representable numbers for the denormalized format. The representation now has a uniformly spaced representable numbers in the close vicinity of 0. Intuitively, the denormalized convention takes the four numbers in the last interval of representable numbers of a no-zero representation and spread them out to cover the gap area. This eliminates the undesirable gap in the previous two methods. Note that the distances between representable numbers in the last two intervals are actually identical. In general, if the n-bit exponent is 0, the value is:

$$0.M \cdot 2^{-2^{n-1}+2}$$

As we can see, the denormalization formula is quite complex. The hardware also needs to be able to detect whether a number falls into the denormalized interval and choose the appropriate representation for that number. The amount of hardware required to implement denormalization in high speed can be significant. Implementations that use moderate amount of hardware often introduce thousands of clock cycles of delay whenever a denormalized number needs to be generated or used. This was the reason why early generations of CUDA devices did not support denormalization. However, virtually all recent generations of CUDA devices, thanks to the increasing number of available transistors of more recent fabrication processes, support denormalization [4]. CUDA programmers can turn on or off denormalization in the GPU arithmetic units by setting the NVCC compiler flag ftz = true (on) or ftz = false (off).

In summary, the precision of a floating point representation is measured by the maximal error that we can introduce to a floating point number by representing that number as one of the representable numbers. The smaller the error is, the higher the precision. The precision of a floating point representation can be improved by adding more bits to mantissa. Adding one bit to the representation of the mantissa improves the precision by reducing the maximal error by half. Thus, a number system has higher precision when it uses more bits for mantissa. This is reflected in double precision vs. single precision numbers in the IEEE standard.

A.3 Precision and special bit patterns in IEEE format

We now turn to more specific details of the actual IEEE format. There are two main precision levels in the IEEE format. A single-precision (sp) number consists of 32 bits that are divided into 1-bit S, 8-bit E, and 23-bit M. A double-precision (dp) number consists of 64 bits that are divided into 1-bit S, 11-bit E, and 52-bit M. Since a double precision number has 29 more bits for mantissa, the largest rounding error for representing a number is reduced to $\frac{1}{2^{29}}$ of that of the single precision format! With the additional three bits of exponent, the double precision format also extends the number of intervals of representable numbers by $2^{2^3} = 256$ times. This extends the range of representable numbers to very large as well as very small values. For example, the total number of atoms in the universe is estimated at 10^{80}, whereas the double-precision format can represent up to 10^{308}!

The special bit patterns of the IEEE floating-point format are described in Fig. A.8. When all exponent bits are 1's, the number represented is an infinity value if the mantissa is 0. All representable numbers fall between $-\infty$ (negative infinity) and $+\infty$ (positive infinite). An ∞ can be created by overflow, e.g., a large number divided by a very small number. That is, when a floating-point arithmetic operation generates a value that exceeds the range of representable numbers, the output value *saturates* at positive or negative infinity. This is in contrast to the integer arithmetic operations that generate wrapped-around or undefined values when their results exceed the range of representable numbers. Any representable number divided by $+\infty$ or $-\infty$ results in 0.

When all exponent bits are 1's, the number is a Not a Number (NaN) if the mantissa is not 0. NaN (Not a Number) is generated by operations when their input values

exponent	mantissa	meaning
11…1	≠ 0	NaN
11…1	=0	$(-1)^S * \infty$
00…0	≠0	denormalized
00…0	=0	0

FIGURE A.8

Special bit patterns in the IEEE standard format.

are undefined mathematically: $0/0$, $0 * \infty$, ∞/∞, $\infty - \infty$. They are also used for data that have not been properly initialized in a program. There are two types of NaN's in the IEEE standard: signaling and quiet. Signaling NaN's (SNaNs) should be represented with the most significant mantissa bit cleared whereas Quiet NaN's are represented with most significant mantissa bit set.

Signaling NaN causes an exception when used as input to arithmetic operations. For example the operation (1.0 + signaling NaN) raises an exception signal to the operating system. Signaling NaN's are used in situations where the programmer would like to make sure that the program execution be interrupted whenever any NaN values are used in floating-point computation. These situations usually mean that there is something wrong with the execution of the program. In mission critical applications, the execution cannot continue until the validity of the execution can be verified with a separate means. For example, software engineers often represent all the uninitialized data as signaling NaN. This practice ensures the detection of using uninitialized data during program execution. The current generation of GPU hardware does not support signaling NaN. This is due to the difficulty of supporting accurate signaling of exceptions during massively parallel execution.

Quiet NaN generates another quiet NaN without causing an exception when used as input to arithmetic operations. For example, the operation (1.0 + quiet NaN) generates a quiet NaN. Quiet NaN's are typically used in applications where the user can review the output and decide if the application should be re-run with a different input for more valid results. When the results are printed, Quite NaN's are printed as "NaN" so that the user can spot them in the output file easily.

Note that there are many possible bit patterns for NaN. For example, in the single-precision format, there are $2^{22} - 1$ possible NaN bit patterns given that the most significant mantissa bit is reserved for signaling/quiet. Some of these bit patterns can potentially be used to indicate the cause for generating the NaN.

A.4 Arithmetic accuracy and rounding

Now that we have a good understanding of the IEEE floating-point format, we are ready to discuss the concept of arithmetic accuracy. While the precision is determined by the number of mantissa bits used in a floating-point number format, the accuracy is determined by the operations performed on floating-point numbers. The accuracy of a floating-point arithmetic operation is measured by the maximal error introduced by the operation. The smaller the error is, the higher the accuracy. The most common source of error in floating-point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding. Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly. For example, a multiplication generates a product value that consists of twice the number of bits than either of the input values. For another example, adding two floating-point numbers can be done by adding their mantissa values together if the

two floating-point values identical exponents. When two input operands to a floating-point addition have different exponents, the one with the smaller exponent needs to be *aligned* to the other operand by repeatedly dividing its mantissa by 2 (that is, all the mantissa bits are shifted to the right by one bit position) and incrementing its exponent by 1 until the exponents are equal. As a result, the final result can have more bits than the format can accommodate.

Alignment shifting of operands can be illustrated with a simple example based on the 5-bit representation in Fig. A.4. Assume that we need to add $1.00_B * 2^{-2}$ (0, 00, 01) to $1.00_B * 2^1$ (0, 10, 00), that is, we need to perform $1.00_B * 2^1 + 1.00_B * 2^{-2}$. Due to the difference in exponent values, the mantissa value of the second number needs to be right shifted by 3 bit positions before it is added to the first mantissa value. That is, the addition becomes $1.00_B * 2^1 + 0.001_B * 2^1$. The addition can now be performed by adding the mantissa values together. The ideal result would be $1.001_B * 2^1$. However, we can see that this ideal result is not a representable number in a 5-bit representation. It would have required three bits of mantissa and there are only two mantissa bits in the format. Thus, the best one can do is to generate one of the closest representable numbers, which is either $1.01_B * 2^1$ or $1.00_B * 2^1$. By doing so, we introduce an error, $0.001_B * 2^1$, which is half the place value of the least significant place. We refer to this as 0.5_D ULP (Units in the Last Place). One can also view ULP as the gap between two representable numbers in a given format. If the hardware is designed to perform arithmetic and rounding operations perfectly, the error that it introduces should be no more than 0.5_D ULP. To our knowledge, this is the accuracy achieved by all CUDA devices today.

In practice, some of the more complex arithmetic hardware units, such as division and transcendental functions, are typically implemented with polynomial approximation algorithms. If the hardware does not use sufficient number of terms in the approximation, the result may have an error larger than 0.5_D ULP. For example, if the ideal result of an inversion operation is $1.00_B * 2^1$ but the hardware generates a $1.10_B * 2^1$ due to the use of an approximation algorithm, the error is 2_D ULP since the error ($1.10_B - 1.00_B = 0.10_B$) is two times bigger than the units in the last place (0.01_B). In practice, the hardware inversion operations in some early devices introduce an error that is twice the place value of the least place of the mantissa, or 2 ULP. Thanks to the more abundant transistors in more recent generations of CUDA devices, their hardware arithmetic operations are mathematically accurate and conform to the IEEE standard [4].

A.5 Lower-precision formats

Some applications intrinsically operate on smaller input data types (4-bit, 6-bit, 8-bit, and 16-bit). Applications such as medical imaging, remote sensing, radio astronomy, seismic analysis, and other natural data frequently fit into this category. More recently, deep learning models such as LLMs have been adapted to use smaller floating-point data types. Although these applications need only small numbers of significant

bits in their data representation, they benefit from the larger range of representable numbers of floating-point formats. Thus, they motivate smaller floating-point formats than single precision. Using smaller formats would improve the compute intensity of these applications by reducing the number of bytes fetched for each arithmetic operation.

Recent GPU architectures introduced hardware support for computing with 16-bit, 8-bit, and 4-bit floating point numbers to improve the performance and energy efficiency for applications that can live with lower-precision arithmetic [3]. For example, H100's 8-bit floating point (FP8) arithmetic throughput using tensor cores is 3,958 TFLOPS, a significant improvement compared to its 989 TFLOPS single-precision throughput using tensor cores.

A.6 Algorithm considerations

Numerical algorithms often need to sum up a large number of values. For example, the dot product in matrix multiplication needs to sum up pair-wise products of input matrix elements. Ideally, the order of summing these values should not affect the final total since addition is an associative operation. However, with finite precision, the order of summing these values can affect the accuracy of the final result. For example, if we need to perform a sum reduction on four numbers in our 5-bit representation: $1.00_B * 2^0 + 1.00_B * 2^0 + 1.00_B * 2^{-2} + 1.00_B * 2^{-2}$.

If we add up the numbers in strict sequential order, we have the following sequence of operations:

$$\begin{aligned} &1.00_B * 2^0 + 1.00_B * 2^0 + 1.00_B * 2^{-2} + 1.00_B * 2^{-2} \\ &= 1.00_B * 2^1 + 1.00_B * 2^{-2} + 1.00_B * 2^{-2} \\ &= 1.00_B * 2^1 + 1.00_B * 2^{-2} = 1.00_B * 2^1 \end{aligned}$$

Note that in the second step and third step, the smaller operand simply disappears because they are too small compared to the larger operand.

Now, let's consider a parallel algorithm where the first two values are added and the second two operands are added in parallel. The algorithm then adds up the pair-wise sum:

$$\begin{aligned} &(1.00_B * 2^0 + 1.00_B * 2^0) + (1.00_B * 2^{-2} + 1.00_B * 2^{-2}) \\ &= 1.00_B * 2^1 + 1.00_B * 2^{-1} = 1.01_B * 2^1 \end{aligned}$$

The reader should recognize that this algorithm implements a reduction tree as discussed in Chapter 10 and it assumes the associativity of the add operation. Note that the results are different from the sequential result! This is because the sum of the third and fourth values is large enough that it now affects the addition result. This discrepancy between sequential algorithms and parallel algorithms often surprises application developers who are not familiar with floating-point precision and accuracy considerations.

Although we showed a scenario where a parallel algorithm produced a more accurate result than a sequential algorithm, the reader should be able to come up with a slightly different scenario where the parallel algorithm produces a less accurate result than a sequential algorithm. That is, the associativity of the add operation is not strictly true for floating-point numbers. Experienced application developers either make sure that the variation in the final result can be tolerated or to ensure that the data is sorted or grouped in a way that the parallel algorithm results in the most accurate results.

A common technique to maximize floating-point arithmetic accuracy is to pre-sort data before a reduction computation. In our sum reduction example, if we pre-sort the data according to ascending numerical order, we will have the following:

$$1.00_B * 2^{-2} + 1.00_B * 2^{-2} + 1.00_B * 2^{0} + 1.00_B * 2^{0}$$

When we divide up the numbers into groups in a parallel algorithm, say the first pair in one group and the second pair in another group, numbers with numerical values close to each other are in the same group. Obviously, the signs of the numbers need to be taken into account during the pre-sorting process. Therefore, when we perform addition in these groups, we will likely have accurate results. Furthermore, some parallel algorithms use each thread to sequentially reduce values within each group. Having the numbers sorted in ascending order allows a sequential addition to get higher accuracy. This is a reason why sorting is frequently used in massively parallel numerical algorithms. Interested readers should study more advanced techniques such as compensated summation algorithm, also known as Kahan's Summation Algorithm, for getting even more robust approach to accurate summation of floating point values [5].

A.7 Linear solvers and numerical stability

While the order of operations may cause variation in the numerical outcome of reduction operations, it may have even more serious implications on some types of computation such as solvers for linear systems of equations. In these solvers, different numerical values of input may require different ordering of operations in order to find a solution. If an algorithm fails to follow a desired order of operations for an input, it may fail to find a solution even though the solution exists. Algorithms that can always find an appropriate operation order, and thus finding a solution to the problem as long as it exists for any given input values are called numerically stable. Algorithms that fall short are referred to as numerically unstable.

In some cases, numerical stability considerations can make it more difficult to find efficient parallel algorithms for a computational problem. We can illustrate this phenomenon with a solver that is based on Gaussian Elimination. Consider the following system of linear equations:

$$3X + 5Y + 2Z = 19 \text{ (Equation 1)}$$

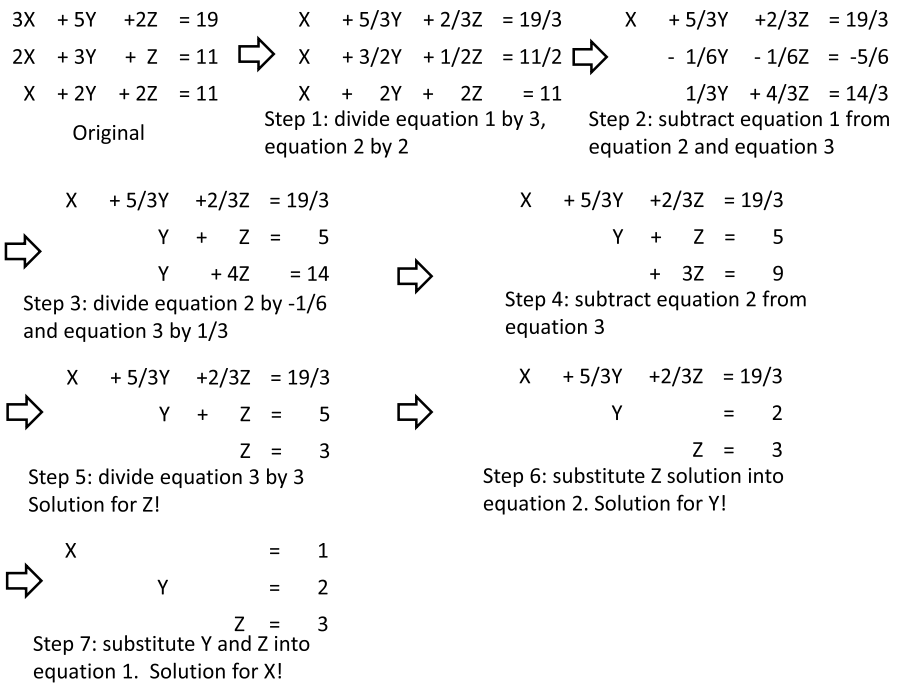


FIGURE A.9

Gaussian elimination and backward substitution for solving systems of linear equations.

$$2X + 3Y + Z = 11 \text{ (Equation 2)}$$

$$X + 2Y + 2Z = 11 \text{ (Equation 3)}$$

As long as the three planes represented by these equations have an intersection point, we can use Gaussian elimination to derive the solution that gives the coordinate of the intersection point. We show the process of applying Gaussian elimination to this system in Fig. A.9, where variables are systematically eliminated from lower positioned equations.

In the first step, all equations are divided by their coefficient for the X variable: 3 for Equation 1, 2 for Equation 2, and 1 for Equation 3. This makes the coefficients for X in all equations the same. In step two, Equation 1 is subtracted from Equation 2 and Equation 3. These subtractions eliminate variable X from Equation 2 and Equation 3, as shown in Fig. A.9.

We can now treat Equation 2 and Equation 3 as a smaller system of equations with one fewer variable than the original equation. Since they do not have variable X, they can be solved independently from Equation 1. We can make more progress by eliminating variable Y from Equation 3. This is done in step 3 by dividing Equation 2 and Equation 3 by the coefficients for their Y variables: $-1/6$ for Equation 2 and 1/3

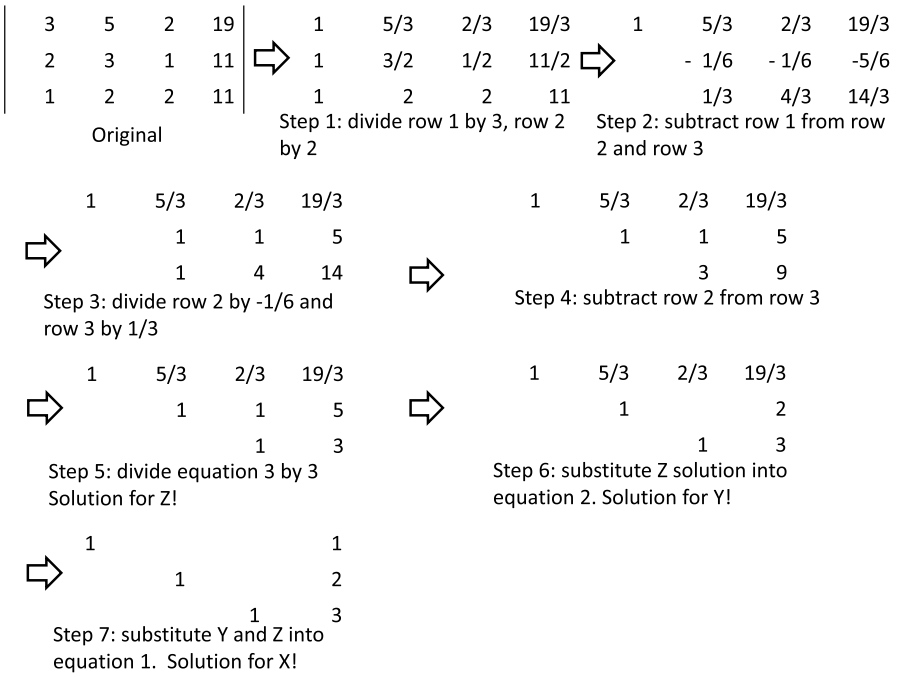


FIGURE A.10

Gaussian elimination and backward substitution in matrix view.

for Equation 3. This makes the coefficients for Y in both Equation 2 and Equation 3 the same. In step four, Equation 2 is subtracted from equation 3, which eliminates variable Y from equation 3.

For systems with larger number of equations, the process would be repeated more. However, since we have only three variables in this example, the third equation has only the Z variable. We simply need to divide Equation 3 by the coefficient for variable Z. This conveniently gives us the solution $Z = 3$.

With the solution for Z variable in hand, we can substitute the Z value into Equation 2 to get the solution $Y = 2$. We can then substitute both $Z = 3$ and $Y = 2$ into Equation 1 to get the solution $X = 1$. We now have the complete solution for the original system. It should be obvious why step 6 and step 7 form the second phase of the method called backward substitution. We go backwards from the last equation to the first equation to get solutions for more and more variables.

In practice, the equations are stored in matrix forms in computers. Since all calculations only involve the coefficients and the right-hand-side values, we can just store these coefficients and right-hand-side values in a matrix. Fig. A.10 shows the matrix view of the Gaussian elimination and back substitution process. Each row of the matrix corresponds to an original equation. Operations on equations become operations on matrix rows.

After Gaussian elimination, the matrix becomes a triangular matrix. This is a very popular type of matrix for various physics and mathematics reasons. We see that the end goal is to make the coefficient part of the matrix into a diagonal form, where each row has only a value 1 on the diagonal line. This is called an identity matrix because the result of multiplying any matrix multiplied by an identity matrix is itself. This is also the reason why performing Gaussian elimination on a matrix is equivalent to multiplying the matrix by its inverse matrix.

It is straightforward to design a parallel algorithm for the Gaussian elimination procedure that we described in Fig. A.10. For example, we can write a CUDA kernel and designate each thread to perform all calculations to be done on a row of the matrix. For systems whose coefficients can fit into shared memory, we can use a thread block to perform Gaussian elimination. All threads iterate through the steps. After each division step, all threads participate in barrier synchronization. They then all perform a subtraction step, after which one thread will stop its participation since its designated row has no more work to do until the back substitution phase. After the subtraction step, all threads need to perform barrier synchronization again to ensure that the next step will be done with the updated information. With systems of equations with many variables, we can expect reasonable amount of speedup from the parallel execution.

Unfortunately, the simple Gaussian elimination algorithm we have been using can suffer from numerical instability. This can be illustrated with the example below.

$$5Y + 2Z = 16 \text{ (Equation 1)}$$

$$2X + 3Y + Z = 11 \text{ (Equation 2)}$$

$$X + 2Y + 2Z = 11 \text{ (Equation 3)}$$

We will encounter a problem when we perform step 1 of the algorithm. The coefficient for the X variable in Equation 1 is zero. We will not be able to divide Equation 1 by the coefficient for variable X and eliminate the X variable from Equation 2 and Equation 3 by subtracting Equation 1 from Equation 2 and Equation 3. The reader should verify that this system of equation is solvable and has the same solution $X = 1$, $Y = 2$, and $Z = 3$. Therefore, the algorithm is numerically unstable. It can fail to generate a solution for certain input values even though the solution exists.

A more subtle issue arises when the coefficient value of the leading variable in one the steps is small and may have been generated with rounding errors from a previous iteration. In this case, using dividing the rest of the coefficients by this small, inaccurate value could introduce even larger error for future steps and eventually the solution. For this reason, the numerical instability of an algorithm is often measured with the growth of errors over its steps.

This is a well-known problem with Gaussian elimination algorithms and can be addressed with a method commonly referred to as pivoting. The idea is to find one of the remaining equations whose coefficient for the lead variable is not zero, and preferably large in absolute value. By swapping the current top equation with the identified equation, the algorithm can successfully and more accurately eliminate the lead variable from the rest of the equations. If we apply pivoting to the three equations, we end up with the following set.

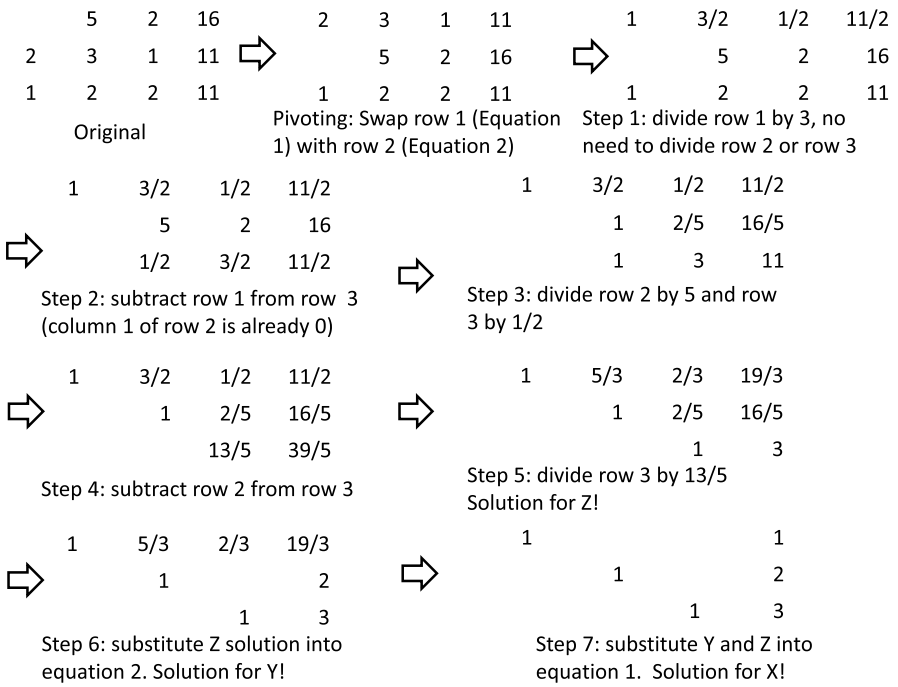


FIGURE A.11

Gaussian elimination with pivoting.

$$2X + 3Y + Z = 11 \text{ (Equation 1', original Equation 2)}$$

$$5Y + 2Z = 16 \text{ (Equation 2', original Equation 1)}$$

$$X + 2Y + 2Z = 11 \text{ (Equation 3', original Equation 3)}$$

Note that the coefficient for X in Equation 1' is no longer zero. We can proceed with Gaussian elimination, as illustrated in Fig. A.11.

The reader should follow the steps in Fig. A.11. The most important additional insight is that some equations may not have the variable that the algorithm is eliminating at the current step (see row 2 of Step 1 in Fig. A.11). The designated thread does not need to do the division on the equation.

In general, the pivoting step should choose the equation with the largest absolute coefficient value among all the lead variables and swap its equation (row) with the current top equation as well as swap the variable (column) with the current variable. While pivoting is conceptually simple, it can incur significant implementation complexity and performance overhead. In the case of our simple CUDA kernel implementation, recall that each thread is assigned a row. Pivoting requires an inspection and perhaps swapping of coefficient data spread across these threads. This is not a

big problem if all coefficients are in the shared memory. We can run a parallel max reduction using threads in the block as long as we control the level of control flow divergence within warps.

However, if the system of linear equations is being solved by multiple thread blocks or even multiple nodes of a compute cluster, the idea of inspecting data spread across multiple thread blocks or multiple compute cluster nodes can be an extremely expensive proposition. This is the main motivation for communication avoiding algorithms that avoids a global inspection of data such as pivoting [6]. In general, there are two approaches to this problem. Partial pivoting restricts the candidates of the swap operation to come from a localized set of equations so that the cost of global inspection is limited. This can, however, slightly reduce the numerical accuracy of the solution. Researchers have also demonstrated that randomization tends to maintain a high level of numerical accuracy for the solution.

A.8 Summary

This appendix introduces the concepts of floating-point format and representable numbers that are foundational to the understanding of precision. Based on these concepts, we also explain the denormalized numbers and why they are important in many numerical applications. In early CUDA devices, denormalized numbers were not supported. However, later hardware generations support denormalized numbers. We have also explained the concept of arithmetic accuracy of floating-point operations. This is important for CUDA programmers to understand the potential lower accuracy of fast arithmetic operations implemented in the special function units. More importantly, the readers should now have a good understanding of why parallel algorithms often can affect the accuracy of calculation results and how one can potentially use sorting and other techniques to improve the accuracy of their computation.

A.9 Exercises

1. Draw the equivalent of Fig. A.5 for a 6-bit format (1-bit sign, 3-bit mantissa, 2-bit exponent). Use your result to explain what each additional mantissa bit does to the set of representable numbers on the number line.
2. Draw the equivalent of Fig. A.5 for another 6-bit format (1-bit sign, 2-bit mantissa, 3-bit exponent). Use your result to explain what each additional exponent bit does to the set of representable numbers on the number line.
3. Assume that in a new processor design, due to technical difficulty, the floating point arithmetic unit that performs addition can only do “round to zero” (rounding by truncating the value towards 0). The hardware maintains sufficient number of bits that the only error introduced is due to rounding. What is the maximal ULP error value for add operations on this machine?

4. A graduate student wrote a CUDA kernel to reduce a large floating-point array to the sum of all its elements. The array will always be sorted with the smallest values to the largest values. To avoid branch divergence, he decided to implement the algorithm of Fig. A.4. Explain why this can reduce the accuracy of his results.
5. Assume that in a arithmetic unit design, the hardware implements an iterative approximation algorithm that generates two additional accurate mantissa bits of the result for the sin() function in each clock cycle. The architect decided to allow the arithmetic function to iterate 9 clock cycles. Assume that the hardware fill in all remaining mantissa bits as 0's. What would be the maximal ULP error of the hardware implementation of the sin() function in this design for the IEEE single-precision numbers? Assume that the omitted "1." mantissa bit must also be generated by the arithmetic unit.

References

[1] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys 23 (1) (1991).
[2] IEEE standard for floating-point arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019.
[3] OCP, Open Compute Project microscaling formats (mx) specification, https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf, 2023.
[4] NVIDIA, Cuda floating-point reference, https://docs.nvidia.com/cuda/floating-point/index.html.
[5] W. Kahan, Pracniques: further remarks on reducing truncation errors, Communications of the ACM 8 (1) (1965) 40.
[6] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, Minimizing communication in numerical linear algebra, SIAM Journal on Matrix Analysis and Applications 32 (3) (2011) 866–901.

APPENDIX

Deep learning basics

B

With special contributions from Carl Pearson and Boris Ginsburg

Machine learning, a term coined by Arthur Samuel of IBM in 1959 [1], is a field of computer science which studies methods for learning application logic from data rather than designing explicit algorithms. Machine learning is most successful in computing tasks where designing explicit algorithms is infeasible, mostly because there is not enough knowledge in the design of such explicit algorithms. That is, one can give examples on what should happen in various situations but not general rules for making such decisions for all possible inputs. For example, machine learning has contributed to the recent improvements in application areas such as automatic speech recognition, computer vision, natural language processing, and recommender systems. In these application areas, one can provide many input examples and what should come out for each input, but there is no algorithm that can correctly process all possible inputs.

The kinds of application logic created with machine learning can be organized according to the types of tasks that they perform. There is a wide range of machine learning tasks. Here we show a few out of a large number:

- Classification – to determine to which of the k categories an input belongs. An example is object recognition: which type of food is shown in this photo?
- Regression – to predict a numerical value given some inputs. An example is to predict the price of a stock at the end of the next trading day.
- Transcription – to convert unstructured data into textual form. An example is optical character recognition.
- Translation – to convert a sequence of symbols in one language to a sequence of symbols in another. An example is translating from English to Chinese.
- Embedding – to convert an input to a vector while preserving relationships between entities. An example is to convert a natural language sentence into a multidimensional vector.

The readers are referred to a large body of literature on the mathematical background and practical solutions to the various tasks of machine learning. The purpose of this appendix is to introduce the computation kernels involved in the neural network approach to the classification task.

A concrete understanding of these kernels will allow the reader to understand and develop kernels for deep learning approaches to other machine learning tasks. As such, in the next section, we will go into details of the classification task to establish the background knowledge needed to understand neural networks.

B.1 Classifiers

In machine learning, a classifier is a function f that performs a classification task by mapping an n-dimensional vector input to one of the k categories or labels:

$$f : R^n \rightarrow 1, 2, ..., k \tag{B.1}$$

For example, the input may be a 2-dimensional vector ($n = 2$) whose first element is the income level of a person and the second element is the age of the same person. A classifier f may be defined to map a person with such a vector to one of the four ($k = 4$) categories: person with post-graduate degree (1), person with bachelor's degree (2), high school graduate (3), or none of the above (4).

The classifier function f is a mathematical expression parameterized by θ that defines how the input vector x is mapped to numerical code y, i.e.:

$$y = f(x, \theta) \tag{B.2}$$

The parameter θ is commonly referred to as the *model*. It encapsulates f's mathematical expression coefficients that are learned from data. This definition of θ is best illustrated with a concrete example.

Let us consider a type of classifiers called *perceptron* [2]:

$$y = sign(W \cdot x + b) \tag{B.3}$$

where W is a vector of coefficients for the input vector x and b is a bias constant. W and x are of the same length. The $sign$ function returns value 1 if its input is positive, 0 if its input is 0, and -1 if its input is negative. That is, the $sign$ function as a classifier activates, i.e., finalizes, the mapping of the input value into three categories: $-1, 0, 1$, thus it is often called the *activation* function. Activation functions introduce non-linearity into an otherwise linear function of a perceptron.

In this case, the model θ is the combination of the vector W and the constant b. The structure of the model is a $sign$ function whose input is a linear expression of input x elements where the coefficients are elements of W (i.e., dot product between the W vector and the x vector), and the constant is b.

Since the elements of W are multiplied with the elements of x before the pairwise products are summed up in the dot product, they serve as weighing coefficients for the x elements in the final sum. Therefore, they are commonly referred to as the $Weights$ of the model. Since f is a linear function of x that feeds into the activation $sign$ function, perceptrons are a form of $linear$ classifiers.

Fig. B.1 shows a perceptron example where each input is a 2D vector (x_1, x_2). The perceptron's model θ consists of a weight vector (w_1, w_2) and a bias constant b. As shown in Fig. B.1, the linear expression $w_1 \cdot x_1 + w_2 \cdot x_2 + b$ defines a line in the $x_1 - x_2$ space that cuts the space into two parts: the part where all points make the expression greater than zero and the part where all points make the expression less than zero. All points on the line make the expression equal to 0.

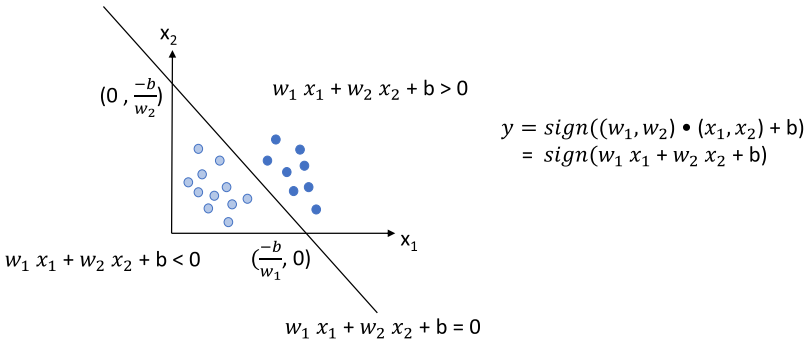


FIGURE B.1

A perceptron linear classifier example where input is a 2D vector.

Visually, given a combination of (w_1, w_2) and b values, we can draw a line in the $x_1 - x_2$ space as shown in Fig. B.1. For example, for a perceptron whose model is $(w_1, w_2) = (2, 3)$ and $b = -6$, we can easily draw a line by connecting the two intersection points with the x_1 axis (i.e., $(\frac{-b}{w_1}, 0) = (3, 0)$) and with the x_2 axis (i.e., $(0, \frac{-b}{w_2}) = (0, 2)$). The line thus drawn corresponds to the equation $2x_1 + 3x_2 - 6 = 0$. With this drawing, we can easily visualize the outcome of input points: any point above the line (shown as darkly shaded dots in Fig. B.1) is classified as 1, any point on the line is classified as 0, and any point below the line (shown as lightly shaded dots in Fig. B.1) is classified as class -1.

The process of computing the mapping for an input is commonly referred to as *inference* for the classifier. In the case of a perceptron, we simply plug the input coordinate values into $y = sign(W \cdot x + b)$. In our example, if the input point is $(5, -1)$, we can perform inference by plugging its coordinates into the perceptron function:

$$y = sign(2 * 5 + 3 * (-1) - 6) = sign(1) = 1 \tag{B.4}$$

So, $(5, -1)$ is classified to class 1, i.e., it is among the darkly shaded dots.

Multi-layer Classifiers

Linear classifiers are useful when there is a way to draw hyperplanes (i.e., lines in a 2D space and planes in a 3D space) that partition the space into regions and thus define each category of data points. Ideally, each category of data points should occupy exactly one such region. For example, in a 2D, 2-category classifier, we need to be able to draw a line that separates points of one category from those of the other. Unfortunately, this is not always feasible.

Consider the classifiers in Fig. B.2. Assume that all input's coordinates fall in the range of [0, 1]. The classifier should classify all points whose x_1 and x_2 values are both greater than 0.5 (points that fall in the upper right quadrant of the domain) as category 1 and the rest as category 0 or category -1. This classifier could be approximately implemented with a line like that shown in Fig. B.2(a). For example,

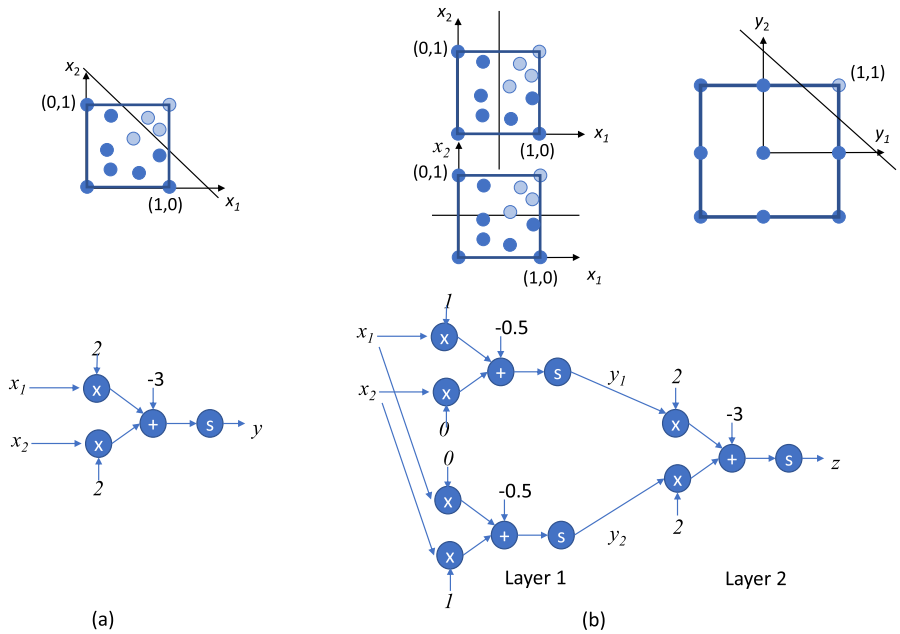


FIGURE B.2

A multi-layer perceptron example.

a line $2x_1 + 2x_2 - 3 = 0$ would classify most of the points properly. However, some of the lightly shaded points whose x_1 and x_2 are both greater than 0.5 but the sum is less than 1.5, e.g., (0.55, 0.65), would be misclassified into category -1 (darkly shaded). This is because any line will necessarily either cut away part of the upper right quadrant or include part of the rest of the domain. There is no single line that can properly classify all possible inputs.

A *multi-layer perceptron* (MLP) allows the use of multiple lines to implement more complex classification patterns. In a multi-layer perceptron, each layer consists of one or more perceptrons. The outputs of perceptrons in one layer are the inputs to those in the next layer. An interesting and useful property is that while the inputs to the first layer have infinite number of possible values, the output of the first layer and thus the input to the second layer can have only a modest number of possible values. For example, if the perceptron from Fig. B.1 was used, as a first layer, its outputs are restricted to -1, 0, 1.

Fig. B.2(b) shows a 2-layer perceptron that can precisely implement the desired classification pattern. The first layer consists of two perceptrons. The first one, $y_1 = sign(x_1 - 0.5)$, classifies all points whose x_1 coordinate is greater than 0.5 as class 1, i.e., the output value is 1. The rest of the points are classified as either category -1 or category 0. The second classifier in the first layer, $y_2 = sign(x_2 - 0.5)$, classifies all points whose x_2 coordinate is greater than 0.5 into category 1. The rest of the points are classified as category 0 or category -1.

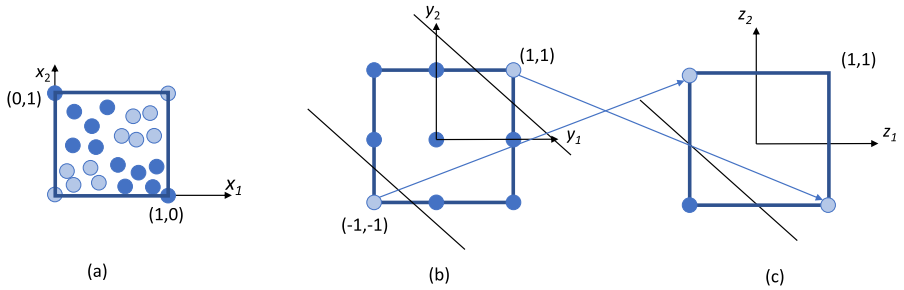


FIGURE B.3

Need for perceptrons with more than two layers.

Therefore, the output of the first layer (y_1, y_2) can only be one of the following nine possibilities: $(-1, -1)$, $(-1, 0)$, $(-1, 1)$, $(0, -1)$, $(0, 0)$, $(0, 1)$, $(1, -1)$, $(1, 0)$, (1,1). That is, there are only 9 possible input pair values to the second layer. Out of these nine possibilities, (1, 1) is special. All the original input points in the lightly shaded category are mapped to (1, 1) by the first layer. Therefore, we can use a simple perceptron in the second layer to draw a line between (1, 1) and the eight other possible points in the $y_1 - y_2$ space, as shown in Fig. B.2(b). This can be done by a line $2y_1 + 2y_2 - 3 = 0$ or many other lines that are small variations of it.

Let us use (0.55, 0.65) input that was misclassified by the single-layer perceptron in Fig. B.2(a). When processed by the two-layer perceptron, the upper perceptron of first layer in Fig. B.2(b) generates $y_1 = 1$ and the lower perceptron generates $y_2 = 1$. Based on these input values, the perceptron in the second layer generates $z = 1$, the correct classification for (0.55, 0.65).

Note that there are still significant limitations of a two-layer perceptron. For example, assume that we need to build a perceptron to classify the input points shown in Fig. B.3(a). The values of the lightly shaded input points can result in input values $(-1, -1)$ or (1, 1) to the second layer. We see that there is no way to draw a single line to properly classify the points in the second layer. We show in Fig. B.2(b) that we can add another line by adding another perceptron in the second layer. The function would be $z_2 = sign(-2y_1 - 2y_2 - 3)$ or small variations of it. The reader should verify that all darkly shaded points in Fig. B.3(b) will be mapped to the $(-1, -1)$ in the $z_1 - z_2$ space. Whereas (1, 1) and $(-1, -1)$ in the $y_1 - y_2$ space are mapped to $(1, -1)$ and $(-1, 1)$ in the $z_1 - z_2$ space. Now we can draw a line $z_1 + z_2 + 1 = 0$, or small variations of it, to properly classify the points, as shown in Fig. B.3(c). Obviously, if we need to partition the input domain into more regions, we might need even more layers to perform proper classification.

B.2 Fully connected and convolutional layers

Layer 1 in Fig. B.2(b) is a small example of a *fully connected layer (FC)*, where every output (i.e., y_1, y_2) is a function of every input (i.e., x_1, x_2). In general, in a

fully connected layer, every one of the m outputs is a function of all the n inputs. All the weights of a fully connected layer form an $m \times n$ weight matrix W where each of the m rows is the weight vector (of size n elements) to be applied to the input vector (of size n elements) to produce one of the m outputs. Therefore, the process of evaluating all the outputs from the inputs of a fully connected layer is a matrix-vector multiplication. As we will see, fully connected layers are core components of many types of neural networks (including large language models), and we will further study their GPU implementations.

Fully connected layers become extremely expensive when m and n become large. The main reason is that a fully connected layer requires an $m \times n$ weight matrix. For example, in an image recognition application, n is the number of pixels in the input image and m is the number of classifications that need to be performed on the input pixels. In this case, n is in the millions for high-resolution images and m can be in the hundreds or more depending on the variety of objects that need to be recognized. Also, the objects can be of different scales and orientations in the images; many classifiers may need to be in place to deal with these variations. Feeding all these classifiers with all inputs is both expensive and likely wasteful.

Convolutional layers reduce the cost of fully connected layers by reducing the number of inputs each classifier takes and sharing the same weights across classifiers. In a convolutional layer, each classifier only takes a subset of the input vector, e.g., a patch of the input image, and performs convolution on the pixels in the patch based on the weights. The output is called an output feature map since each pixel in the output is the activation result of a classifier. Sharing weights across classifiers allows the convolutional layers to have large numbers of classifiers, i.e., large m values, without an excessive number of weights. Computationally, this can be implemented as a 2D convolution. However, this approach effectively applies the same classifier to different part of an image. One can have different sets of weights applied to the same input and generate multiple output feature maps, as we will see later in this appendix.

B.2.1 Training models

So far, we have assumed that the model parameters used by a classifier somehow exist and are available for use. Now we turn to *training*, or the process of using data to determine the values of the model parameters θ. In the case of a 2D vector classifier in Fig. B.1, the model parameters are the weights (w_1, w_2) and the bias b. For simplicity, we will assume *supervised training*, where input data labeled with desired or expected output values are used to determine the weight and bias values. Other training modalities, like *semi-supervised* and *reinforcement learning*, have also been developed to reduce the reliance on labeled data. The reader is referred to the literature to understand how training can be accomplished under such circumstances.

In our example, each 2D classifier is a mathematical expression of the input vector and model parameters. We can thus set up a system of equations using the input vector values and their labels. Each input vector and its label results in an equation where the variables are the model parameters.

For example, if an input vector (1, 5) has a label value 1, we can set up an equation

$$sign(w_1 + 5w_2 + b) = 1 \tag{B.5}$$

Because of the *sign* function, this is not a linear equation. That is, if we have three of such equations, we will not have a unique solution of w_1, w_2, and b values that satisfies these three equations. There can be an infinite number of solutions. Mathematically, this is referred to as an ill-defined problem. In fact, it is entirely possible that one may not be able to find w_1, w_2, and b values that can satisfy all the equations given a dataset. This happens when the points of the dataset in a 2D space simply cannot be classified with a line, as exemplified in Fig. B.2. Thus, a large number of labeled input data are used to iteratively find a solution of the parameter values that result in an acceptable number of mistakes by the classifier when processing the dataset.

In the perceptron example in Fig. B.1, each data point used for training would be labeled with its desired classification result: -1, 0, or 1. The training process starts with an initial guess of the (w_1, w_2) and b values, performs inference on the input data, and generates classification results. These classification results are compared to the labels, and any discrepancy is used to make adjustments to the (w_1, w_2) and b values until the discrepancy between the classification results and the labels settles at an acceptable level.

Such an iterative method requires a measure of the discrepancy between the output values and the provided label values and a procedure to adjust the model parameters so that the output values will better match their expected values after the adjustment. As the reader probably has guessed, there are many different measures and different model parameter adjustment methods. We will use a simple but commonly used form of each to illustrate the training process.

Error Function. An *error function*, sometimes referred to as a *cost function*, is defined to quantify the difference between the classification result and the corresponding label for each data point. Assume that y is the classification output class and t is the label. The following is an example error function:

$$E = \frac{(y - t)^2}{2} \tag{B.6}$$

This error function has the nice property that the error value is always positive as long as there is any difference, positive or negative, between the values of y and t. If we need to sum up the error across many input data points, both positive and negative differences will contribute to the total rather than canceling each other out. One can also define the error as the absolute value of the difference, among many other options. As we will see, the coefficient $\frac{1}{2}$ simplifies the computation involved in solving the model parameters.

Stochastic Gradient Descent. The training process will attempt to find the model parameter values that minimize the sum of the error function values for all the training data points. This can be done with a *stochastic gradient descent* approach, which

repeatedly runs different permutations of the input data set through the classifier, evolves the parameter values, and checks if the parameter values have converged in that their values have stabilized and changed less than a threshold since the last iteration. Once the parameter values converge, the training process ends. If the sum of the cost function values for the training dataset is above an acceptable level, a new training process will be launched with a different initial guess of the model parameters. In some cases, a new model design, such as added levels, may be needed for the training process to succeed.

Epoch. During each iteration of the training process, called an *epoch*, the training input data set is first randomly shuffled, i.e., permuted, before it is fed to the classifier. This randomization of the input data ordering helps to avoid sub-optimal solutions. For each input data element, its classifier output y value is compared with the label data to generate the error function value. In our perceptron example, if a data label is (class) 1 and the classifier output is (class) -1, the error function value using $E = \frac{(y-t)^2}{2}$ would be 2. If the error function value is larger than a threshold, a back-propagation operation is activated to make changes to the parameters so that the inference error can be reduced.

Back-propagation. The idea of *back-propagation* is to start with the error function and look back into the classifier and identify the way each parameter contributes to the error function value [3]. If the error function value increases when a parameter's value increases for a data element, we should decrease the parameter value so that the error function value can decrease for this data point. Otherwise, we should increase the parameter value to reduce the error function value for the data point. The rate and direction in which a function's value changes as one of its input variables changes are mathematically defined as the partial derivative of the function over the variable. For a perceptron, the model parameters and the input data points are considered input variables for the purpose of calculating the partial derivatives of the error function. Therefore, the back-propagation operation will need to derive the partial derivative values of the error function over the model parameters for each input data element that triggers the back-propagation operation.

Let us use the perceptron $y = sign(w_1x_1 + w_2x_2 + b)$ to illustrate the back-propagation operation. Assume error function $E = \frac{(y-t)^2}{2}$ and that the back-propagation is triggered by a discrepancy between the classification result and the label for training input data element (5, 2). The goal is to modify the w_1, w_2, and b values so that the perceptron will more likely classify (5, 2) correctly. That is, we need to find the values of partial derivatives $\frac{\partial E}{\partial w_1}$, $\frac{\partial E}{\partial w_2}$, and $\frac{\partial E}{\partial b}$ in order to make changes to the w_1, w_2, and b values.

Chain Rule. We see that E is a function of y and y is a function of w_1, w_2, and b. Thus, we can use the chain rule to derive these partial derivatives. For w_1,

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial w_1} \tag{B.7}$$

$\frac{\partial E}{\partial y}$ is straightforward:

$$\frac{\partial E}{\partial y} = \frac{\partial \frac{(y-t)^2}{2}}{\partial y} = y - t \tag{B.8}$$

However, we face a challenge with $\frac{\partial y}{\partial w_1}$. Note that the sign function is not a differentiable function as it is not continuous at 0. To solve this problem, the machine learning community commonly use a smoother version of the sign function that is differentiable near zero and close to the sign function value for x values away from 0. A simple example of such a smoother version is the *sigmoid function* $s = \frac{1-e^{-x}}{1+e^{-x}}$. For x values that are negative with large absolute value, the sigmoid expression is dominated by the e^{-x} terms and the sigmoid function value will be approximately -1. For x values that are positive with large absolute values, the e^{-x} terms diminish, and the function value will be approximately 1. For x values that are close to zero, the function value increases rapidly from near -1 to near 1. Thus, the sigmoid function closely approximates the behavior of a sign function and yet is continuous differentiable for all x values. With this change from sign to sigmoid, the perceptron is $y = sigmoid(w_1x_1 + w_2x_2 + b)$. We can express $\frac{\partial y}{\partial w_1}$ as $\frac{\partial sigmoid(k)}{\partial k}\frac{\partial k}{\partial w_1}$ using the chain rule with an intermediate variable $k = w_1x_1 + w_2x_2 + b$. Based on calculus manipulation, $\frac{\partial y}{\partial w_1}$ is simply x_1 and

$$\frac{\partial sigmoid(k)}{\partial k} = \left(\frac{1-e^{-k}}{1+e^{-k}}\right)' = (1-e^{-k})'\left(\frac{1}{1+e^{-k}}\right) + (1-e^{-k})\left(\frac{1}{1+e^{-k}}\right)' =$$
$$e^{-k}\left(\frac{1}{1+e^{-k}}\right) + (1-e^{-k})(-1 \times \left(\frac{1}{1+e^{-k}}\right)^2(-e^{-k})) = \frac{2e^{-k}}{(1+e^{-k})^2} \tag{B.9}$$

Putting it all together, we have

$$\frac{\partial E}{\partial w_1} = (y-t)\frac{2e^{-k}}{(1+e^{-k})^2}x_1 \tag{B.10}$$

Similarly,

$$\frac{\partial E}{\partial w_2} = (y-t)\frac{2e^{-k}}{(1+e^{-k})^2}x_2 \tag{B.11}$$

$$\frac{\partial E}{\partial b} = (y-t)\frac{2e^{-k}}{(1+e^{-k})^2} \tag{B.12}$$

where,

$$k = w_1x_1 + w_2x_2 + b \tag{B.13}$$

It should be clear that all the three partial derivative values can be completely determined by the combination of the input data (x_1, x_2 and t) and the current values of the model parameters (w_1, w_2 and b). In our example, $x_1 = 5$ and $x_2 = 2$. We can

calculate the k value and all the partial derivative values by using the current w_1, w_2, and b values along with these two values.

The final step for the back-propagation is to modify the parameter values. Recall that the partial derivative of a function over a variable gives the direction and rate of change in the function value as the variable changes its value. If the partial derivative of the error function over a parameter has a positive value given the combination of input data and current parameter values, we want to decrease the value of the parameter so that the error function value will decrease. On the other hand, if the partial derivative of the error function of the variable has a negative value, we want to increase the value of the parameter so that the error function value will decrease.

Learning Rate. Numerically, we would like to make bigger changes to the parameters to whose change the error function is more sensitive, i.e., when the absolute value of the partial derivative of the error function over this parameter is a large value. These considerations lead to us to subtract from each parameter a value that is proportional to the partial derivative of the error function over that parameter. This is accomplished by multiplying the partial derivatives with a constant ϵ, called the *learning rate* constant in machine learning, before it is subtracted from the parameter value. The larger ϵ is, the faster the values of the parameters evolve so the solution can potentially be reached with fewer iterations. However, large ϵ also increases the chance of instability and prevent the parameter values from converging to a solution. This is because changes may cause the classification results for some other input data points to flip from agreement to disagreement with their labels. In our perceptron example, the modifications to the parameters are:

$$w_1 \leftarrow w_1 - \epsilon \frac{\partial E}{\partial w_1} \tag{B.14}$$

$$w_2 \leftarrow w_2 - \epsilon \frac{\partial E}{\partial w_2} \tag{B.15}$$

$$b \leftarrow b - \epsilon \frac{\partial E}{\partial b} \tag{B.16}$$

For the simplicity, we will use a generic symbol θ to represent the model parameters in formula and expressions. That is, we will represent the three expressions above with one generic expression:

$$\theta \leftarrow \theta - \epsilon \frac{\partial E}{\partial \theta} \tag{B.17}$$

The reader should understand that for each of these generic expressions one can replace θ with any of the parameters to apply the expression to the parameter.

Batch. In practice, because the training data can be very large, one may not be able to fit all the training data into the memory of the computing device used for training. Thus, during each epoch, the training dataset is often divided into batches so that each batch can fit into the memory. Since the back-propagation process is quite expensive, it is not triggered by individual data points whose inference result

differs from its label. Rather, the training process runs an entire batch through the inference and accumulates their error function values. If the total error in the batch is too large, back-propagation is triggered for the mini-batch. During back-propagation, the inference result of each data point in the batch is checked, and if it is not correct, the data is used to derive partial derivative values that are used to modify the model parameter values as described above.

Mini-Batch. Since the training process can take many iterations through large datasets, multiple computing devices are often deployed to accelerate the training process. In this case, each batch is further divided into mini-batches. Each mini-batch is assigned to one of the computing devices. In this case, each device will perform forward propagation on the mini-batch and trigger their own back-propagation when the total error for the mini-batch is too big. When all the computing devices complete their min-batch, they collectively sum up all their changes to the model parameters and generate an updated version before they move on to the next batch. This way, the training process using multiple computing devices can update the model parameters in a manner that approximates the training process using a single computing device for the entire batch.

Training Multi-Layer Classifiers. For multi-layer classifiers, the back-propagation starts with the last layer and modify the parameter values in that layer as we discussed above. The question is how we should modify the parameters of the previous layers. Keep in mind that we can derive $\frac{\partial E}{\partial \theta}$ based on $\frac{\partial E}{\partial y}$, as we have demonstrated for the final layer. Once we have $\frac{\partial E}{\partial y}$ for the previous layer, we have everything we need to calculate the modifications to the parameters in that layer.

A simple and yet important observation is that the output of the previous layer is also the input to the final layer. Therefore, $\frac{\partial E}{\partial y}$ of the previous layer is really the $\frac{\partial E}{\partial x}$ of the final layer. So, the key is to derive $\frac{\partial E}{\partial x}$ for the final layer after we modify the parameter values of the final layer. As we can see below, $\frac{\partial E}{\partial x}$ is not that different from $\frac{\partial E}{\partial \theta}$, i.e., $\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial x}$.

Note that we have already derived $\frac{\partial E}{\partial y}$ for the final layer when we were deriving $\frac{\partial E}{\partial y}$ and it can be simply reused. $\frac{\partial y}{\partial x}$ for the final layer is also quite straightforward since the inputs play the same role as the parameters as far as y is concerned. We simply need to do a partial derivative of the intermediate function $k = w_1x_1 + w_2x_2 + b$ with respect to the inputs. For our perceptron example, we have:

$$\frac{\partial E}{\partial x_1} = (y - t)\frac{2e^{-k}}{(1 + e^{-k})^2}w_1 \tag{B.18}$$

$$\frac{\partial E}{\partial x_2} = (y - t)\frac{2e^{-k}}{(1 + e^{-k})^2}w_2 \tag{B.19}$$

Where $k = w_1x_1 + w_2x_2 + b$. In the perceptron example in Fig. B.2(b), x_1 of the final layer (Layer 2) is y_1, output of the top perceptron of Layer 1 and x_2 is y_2, output of the bottom perceptron of Layer 1. Now we are ready to proceed with the calculation of $\frac{\partial E}{\partial \theta}$ for the two perceptrons in the previous layer and make adjustments to the

parameter values in these two perceptrons. Obviously, this process can be repeated if there are more layers.

Feed-Forward Networks. By connecting layers of classifiers and feed the output of each layer to the next, we form a *feed-forward network*. Fig. B.2(b) shows an example of a 2-layer feed-forward network. All our discussions on inference with and training of multi-layer perceptrons assume this property. In a feed-forward network, all outputs of an earlier layer go to one or more of the later layers. There are no connections from a later layer output to an earlier layer input. Therefore, the back-propagation can simply iterate from the final stage backwards with no complications caused by feedback loops.

B.3 Convolutional neural networks

Convolutional neural network (CNN) is a category of feed-forward networks that are easier to train and that can be generalized much better than other deep learning models. The CNN was invented in late 1980s [4]. By the early 1990s, CNNs had been experimentally applied to automated speech recognition, optical character recognition (OCR), handwriting recognition, and face recognition [3].

However, until the late 1990s, the mainstream of computer vision and that of automated speech recognition had been based on carefully engineered features. The amount of labeled data was insufficient for training a deep-learning system to compete with recognition/classification functions crafted by human experts. Furthermore, it was a common belief that it was computationally infeasible to automatically learn the parameter values of hierarchical feature extractors with enough layers to outperform human-crafted application-specific feature extractors even if the required labeled datasets existed

Interest in deep, feed-forward networks were revived around 2006 by a group of researchers who introduced unsupervised learning methods that could create multi-layer, hierarchical feature detectors without requiring labeled data [5,6]. The first major application of this approach was in speech recognition. The breakthrough was made possible by GPUs that allowed researchers to train networks ten times faster than traditional CPUs. This advancement, coupled with the massive amount of media data available online, drastically elevated the position of deep-learning approaches. Despite their success in speech, convolutional neural networks were largely ignored in the field of computer vision until 2012.

In 2012, a group of researchers from the University of Toronto trained a convolutional neural network to classify 1000 different classes in the ILSVRC contest [7]. The network was huge by the norms of the time: it had approximately 60 million parameters and 650,000 neurons. It was trained on 1.2 million high-resolution images from the ImageNet database. The network was trained in only one week on two GPUs using a CUDA-based convolutional neural network library written by Alex Krizhevsky [8]. The network achieved break-through results with a winning test error rate of 15.3%. In comparison, the second-place team that used the traditional

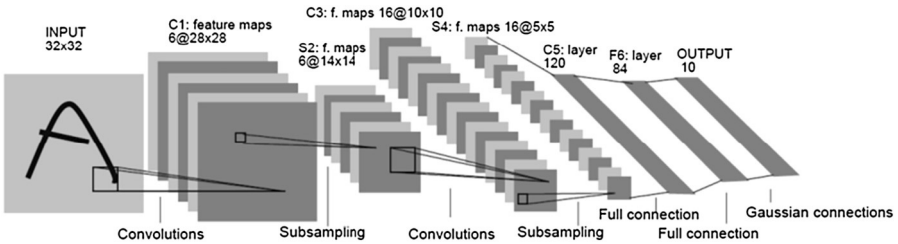


FIGURE B.4

LeNet-5, a convolutional neural network for handwritten digit recognition. The letter A in the input should be classified as none of the 10 classes (digits).

computer vision algorithms had an error rate of 26.2%. This success triggered a revolution in computer vision, and convolutional neural networks became a mainstream tool in computer vision, natural language processing, reinforcement learning, and many other areas.

B.4 CNN inference and training: sequential implementation

This section presents a sequential implementation of CNN inference and training in preparation for their CUDA implementation. We will use LeNet-5, the network designed in the late 1980s for digit recognition [3]. As shown in Fig. B.4, LeNet-5 is composed from three types of layers: convolutional layers, subsampling layers, and fully connected layers. These three types of layers continue to be the key components of today's neural networks. We will consider the logical design and sequential implementation of each type of layer. The input to the network is shown as a gray image with a hand-written digit represented as a 2D 32 $\times$ 32 pixel array. The last layer computes the output, which is the probability for the original image to belong to each one of the 10 classes (digits) that the network is set up to recognize.

We will refer to inputs to and outputs from the CNN layers as *feature maps*, or simply, *features*. For example, in Fig. B.4, the computation of the C1 convolutional layer (Convolutions) at the input end of the network is organized to generate six (28 $\times$ 28) output feature maps from the (32 $\times$ 32) INPUT pixel array. The convolutional layer output to be produced for input feature maps consists of pixels, each of which is produced by performing a convolution between a small local patch of the feature map pixels produced by the previous layer (INPUT in the case of C1) and a set of weights (i.e., convolution filters as defined in Chapter 7) called a *filter bank*. The convolution result is then fed into an activation function like sigmoid (Section B.1) to produce an output pixel in the output feature map. One can think of the convolutional layer for each pixel of an output feature map as a perceptron (Section B.1) whose inputs are the patch of pixels in the input feature maps. That is, the value of each output pixel is the sum of convolution results from the corresponding patches in all input feature maps.

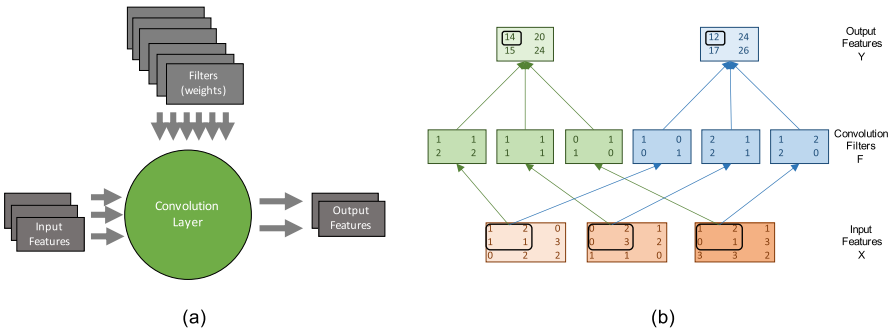


FIGURE B.5

Forward propagation path of a convolutional layer.

Fig. B.5 shows a small convolutional layer example. There are three input feature maps, two output feature maps, and six filter banks. Different input and output feature map pairs in a layer use different filter banks. Since there are three input feature maps and two output feature maps in Fig. B.5, we need $3 \times 2 = 6$ filter banks. For the C3 layer of LeNet in Fig. B.4, there are 6 input feature maps and 16 output feature maps. Thus, a total of $6 \times 16 = 96$ filter banks are used in C3.

Fig. B.5(b) illustrates more details of the calculations done by a convolutional layer. We omitted the activation function for the output pixels for simplicity. We show that each output feature map is the sum of convolutions of all input feature maps. For example, the upper left corner element of output feature map 0 (value 14) is calculated as the convolution between the circled patch of input feature maps and the corresponding filter banks[1]:

$$(1,2,1,1)\cdot(1,1,2,2)+(0,2,0,3)\cdot(1,1,1,1)+(1,2,0,1)\cdot(0,1,1,0)= \\ 1+2+2+2+0+2+0+3+0+2+0+0=14 \quad \text{(B.20)}$$

One can also think of the three 2D input maps as a 3D input feature map and the three 2D filter banks as a 3D filter bank. In Fig. B.5(b), the three 2D filter banks on the left form a 3D filter bank and the three on the right form a second 3D filter bank. In general, if a convolutional layer has n input feature maps and m output feature maps, $n * m$ different 2D filter banks will be used. One can also think about these filter banks as m 3D filter banks. Although not shown in Fig. B.4, all 2D filter banks used in LeNet-5 are 5×5 convolution filters.

Fig. B.6 shows `convLayer_forward`, a sequential C function implementation of the forward propagation path of a convolutional layer. The input feature maps are stored in a 3D array parameter `X[C, H, W]`, where `C` is the second input parameter

[1] Although we use integer values for elements in the feature maps and filter bank for simplicity, they are floating numbers in reality.

```
1  void convLayer_forward(int M, int C, int H, int W, int K,
       float* X, float* W, float* Y) {
2     int H_out = H - K + 1;
3     int W_out = W - K + 1;
4     for(int m = 0;  m < M;  m++) // for each output feature
          map
5        for(int h = 0; h < H_out; h++) // for each output
             element
6           for(int w = 0; w < W_out; w++) {
7              Y[m, h, w] = 0;
8              for(int c = 0;  c < C; c++) // sum over all
                   input feature maps
9                 for(int p = 0; p < K; p++) // KxK filter
10                   for(int q = 0; q < K; q++)
11                      Y[m, h, w] +=  X[c, h + p, w + q] * W
                             [m, c, p, q];
12          }
13 }
```

FIGURE B.6

A C implementation of the forward propagation path of a convolutional layer.

that specifies number of input feature maps, `H` is the third input parameter that specifies the height of each input map image, and `W` is the fourth input parameter that specifies the width of each input map image (line 1). That is, the highest dimension index for `X` selects one of the feature maps (often referred to as channels) and the indices of the lower two dimensions select one of the pixels in an input feature map. For example, the input feature maps for the C1 layer of Fig. B.4 are stored in `X[1, 32, 32]` since there is only one input feature map (INPUT in Fig. B.4) that consists of 32 pixels in each of the *x* and *y* dimensions. This also reflects the fact that one can think of the 2D input feature maps to a layer altogether form a 3D input feature map, which is also referred to as the *input tensor* to the layer.

The output feature maps of a convolutional layer are to be stored in a 3D array parameter `Y[M, H-K + 1, W-K + 1]`, where `M` is the number of output feature maps and `K` is the height (and width) of each 2D filter (line 1). For example, the output feature maps for the C1 layer in Fig. B.4 is stored in `Y[6, 28, 28]` since C1 generates six output feature maps using 5 × 5 filters.

The filter banks are stored in a 4D array parameter `W[M, C, K, K]`.[2] There are `M` × `C` filter banks, one for each pair of input feature map and output feature map. Filter bank `W[m, c, _, _]` is used when using input feature map `X[c, _, _]` to calculate output feature map `Y[m, _, _]`. Recall that each output feature map is the sum of convolutions of all input feature maps.

[2] Note that `W` is used for both the width of images and the name of the filter bank (weight) matrix. In each case, the usage should be clear from the context.

Each iteration of the outermost (m) for-loop (lines 4-12) generates an output feature map. Each iteration of the next two levels (h and w) of for-loops (lines 5-12) generates one pixel of the current output feature map. The next three loop levels (lines 8-11) perform the convolution between the input feature maps and the associated filter banks before accumulating the convolution result for each input feature map into the output feature map (line 11).

The output feature maps of a convolutional layer typically go through a *subsampling* (also known as *pooling*) *layer*. The S2 and S4 layers in Fig. B.4 are examples of sampling layers. A subsampling layer reduces the size of image maps by combining pixels. They are used to allow subsequent convolution layers to classify larger-scale patterns without using larger convolution masks. For example, in Fig. B.4, subsampling layer S2 takes the 6 input feature maps of size 28×28 and generates 6 feature maps of size 14×14. Each pixel in a subsampling output feature map is generated from a 2×2 neighborhood in the corresponding input feature map. The values of these four pixels are averaged to form one pixel in the output feature map. One can also use different ways to combine the pixel values in the neighborhood into a subsampled pixel. The output of a subsampling layer has the same number of output feature maps as the previous layer, but each map has half the number of rows and columns. For example, the number of output feature maps (6) of the subsampling layer S2 is the same as the number of its input feature maps, or the output feature maps of the convolutional layer C1.

Fig. B.7 shows a sequential C implementation of the forward propagation path of a subsampling layer. Each iteration of the outermost (m) for-loop (lines 2-11) generates an output feature map. The next two levels (h, w) of for-loops (lines 3-11) generate individual pixels of the current output map. The two innermost for-loops (lines 6-9) sum up the pixels in the neighborhood. K is equal to 2 in our LeNet-5 subsampling example in Fig. 19.1. A bias value b[m] that is specific to each output feature map is then added to each output feature map and the sum goes through a sigmoid activation function (Section B.2.1). The reader should recognize that each output pixel is generated by the equivalent of a perceptron (Section B.1) that takes four of the input pixels in each feature map as its input and generates a pixel in the corresponding output feature map. Interested readers should also look into the literature for other types of activation functions.

We will focus on the CUDA implementation of the convolution layers in this book since they typically account for the majority of the execution time of CNN training and inference. Interested readers should refer to the literature for the sequential implementations of the Full Connection (i.e., Fully Connected Layer) and the Gaussian Connections Layer in Fig. B.4.

B.4.1 CNN training

Training of CNNs is based on the stochastic gradient descent method and the back-propagation procedure that were discussed in Section B.2.1 [9]. The training data set is labeled with the "correct answer." In the handwriting recognition example, the labels give the correct letter in the image. The label information can be used to generate

```
1  void subsamplingLayer_forward(int M, int H, int W, int K,
       float* Y, float* S){
2    for(int m = 0;  m < M;  m++) // for each output feature
         map
3      for(int h = 0; h < H/K; h++) // for each output
           element,
4        for(int w = 0; w < W/K; w++) { // this code
             assumes that H and W
5          S[m, x, y] = 0.; // are multiples of K
6          for(int p = 0; p < K; p++) { // loop over KxK
               input samples
7            for(int q = 0; q < K; q++)
8              S[m, h, w] += Y[m, K*h + p, K*w+ q] /(K*
                   K);
9          }
10         S[m, h, w] = sigmoid(S[m, h, w] + b[m]); // add
               bias and apply non-linear activation
11       }
12 }
```

FIGURE B.7

A sequential C implementation of the forward propagation path of a subsampling layer. The layer also includes an activation function, which would be included in a convolutional layer if there is no subsampling layer after the convolutional layer.

the "correct" output of the last stage: the correct probability values of the 10-element vector where the probability of the correct digit is 1.0 and those for all other digits are 0.0.

For each training image, the final stage of the network calculates the loss (error) function as the difference between the generated output probability vector element values and the "correct" output vector element values. Given a sequence of training images, we can numerically calculate the gradient of the loss function with respect to the elements of the output vector. Intuitively, it gives the rate at which the loss function value changes when the values of the output vector elements change.

The back-propagation process starts by calculating the gradient of loss function $\frac{\partial E}{\partial y}$ for the last layer. It then propagates the gradient from the last layer towards the first layer through all layers of network. Each layer receives as its input $\frac{\partial E}{\partial y}$ gradient with respect to its output feature maps (which is just the $\frac{\partial E}{\partial x}$ of the later layer) and computes its own $\frac{\partial E}{\partial x}$ gradient with respect to its input feature maps, as shown in Fig. B.8(b). This process repeats until it finishes adjusting the input layer of the network.

If a layer has learned parameters ('weights') w, then the layer also computes its $\frac{\partial E}{\partial w}$ gradient of loss with respect to its weights, as shown in Fig. B.8(a). For example, the fully connected layer is given as: $y = w \cdot x$. The back-propagation of the gradient $\frac{\partial E}{\partial y}$ is given by the equation:

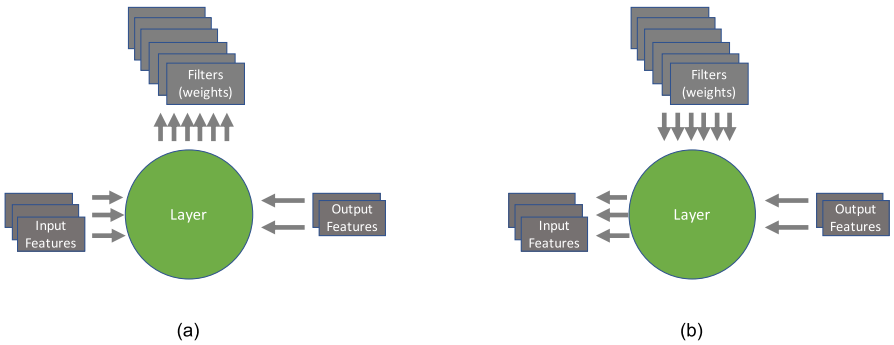


FIGURE B.8

Back-propagation of (a) $\frac{\partial E}{\partial w}$ and (b) $\frac{\partial E}{\partial x}$ for a layer in CNN.

$$\frac{\partial E}{\partial x} = w^T \frac{\partial E}{\partial y} \text{ and } \frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} x^T \tag{B.21}$$

This equation can be derived on an element-by-element basis like what we did for the 2-layer perceptron example. Recall that each fully connected layer output pixel is calculated by a perceptron that takes the pixels in the input feature map as input. As we showed for training multi-layer perceptrons in Section B.2.1, $\frac{\partial E}{\partial x}$ for one of the inputs x is the sum of products between $\frac{\partial E}{\partial y}$ for each output y element that the input element contributes to and the w value via which the x value contributes to the y value. Because each row of the w matrix relates all the x elements (columns) to an y element (one of the rows) for the fully connected layer, each column of w (i.e., row of w^T) relates all y (i.e., $\frac{\partial E}{\partial y}$) elements back to an x (i.e., $\frac{\partial E}{\partial x}$) element since transposition switches the roles of rows and columns. Thus, the matrix-vector multiplication $w^T \frac{\partial E}{\partial y}$ results in a vector that has the $\frac{\partial E}{\partial x}$ values for all input x elements.

Similarly, since each w element is multiplied with one x element to generate a y element, the $\frac{\partial E}{\partial w}$ of each w element can be calculated as the product of an element of $\frac{\partial E}{\partial y}$ with an x element. Thus, the matrix multiplication between $\frac{\partial E}{\partial y}$ (a single column matrix) and x^T (a single row matrix) results in a matrix of $\frac{\partial E}{\partial w}$ values for all w elements of the fully connected layer. This can also be seen as an outer product between the $\frac{\partial E}{\partial y}$ and x vectors.

Let's turn our attention to the back-propagation for a convolutional layer. We will start from the calculation of $\frac{\partial E}{\partial x}$ from $\frac{\partial E}{\partial y}$, which will ultimately be used to calculate the gradients for the previous layer. The gradient $\frac{\partial E}{\partial x}$ with respect to the channel c of input x is given as the sum of the "backward convolution" with the corresponding $W(m, c)$ over all the m layer outputs:

$$\frac{\partial E}{\partial x}(c, h, w) = \sum_{m=0}^{M-1} \sum_{p=0}^{K-1} \sum_{q=0}^{K-1} \left(\frac{\partial E}{\partial y}(m, h - p, w - q) * w(m, c, k - p, k - q)\right) \tag{B.22}$$

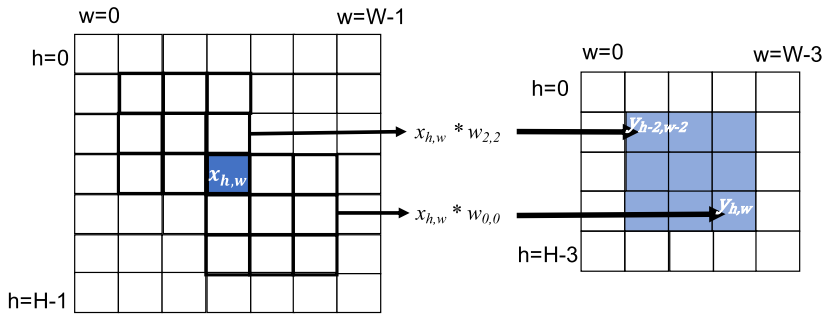


FIGURE B.9

Convolutional layer: Back-propagation of (a) $\frac{\partial E}{\partial w}$ and (b) $\frac{\partial E}{\partial x}$.

The "backward convolution" through the $h - p$ and $w - q$ indexing allows the gradients of all output y elements that received contributions from an x element in the forward convolution to contribute to the gradient of that x element through the same weights. This is because in the forward inference of the convolutional layer, any change in the value of the x element is multiplied by these w elements and contributes to the change in the loss function value through these y elements. Fig. B.9 shows the indexing pattern using a small example with 3×3 filter banks. The 9 shaded y elements in the output feature map are the y elements that receive contributions from $x_{h,w}$ in the forward inference. For example, input element $x_{h,w}$ contributes to $y_{h-2,w-2}$ through multiplication with $w_{2,2}$ and to $y_{h,w}$ through multiplication with $w_{0,0}$. Therefore, during back-propagation, $\frac{\partial E}{\partial x_{h,w}}$ should receive contribution from the $\frac{\partial E}{\partial y}$ values of these 9 elements and the computation is equivalent to a convolution with a transposed filter bank w^T.

Fig. B.10 shows the C code for calculating each element of $\frac{\partial E}{\partial x}$ for each input feature map. Note that the code assumes that $\frac{\partial E}{\partial y}$ has been calculated for all the output feature maps of the layer and passed in with a pointer argument `dE_dY`. This is a reasonable assumption since $\frac{\partial E}{\partial y}$ for the current layer is the $\frac{\partial E}{\partial x}$ for its immediate next layer whose gradients should have been calculated in the back-propagation before the reaching the current layer. It also assumes that the space of $\frac{\partial E}{\partial x}$ has been allocated in the device memory whose handle is passed in as a pointer argument `dE_dX`. The function generates all the elements of $\frac{\partial E}{\partial x}$.

The sequential code for calculating $\frac{\partial E}{\partial w}$ for a convolutional layer computation is similar to that of $\frac{\partial E}{\partial x}$ and is shown in Fig. B.11. Since each $W(m, c)$ affects all elements of output $Y(m)$, we should accumulate gradients for each $W(m, c)$ over all pixels in the corresponding output feature map:

$$\frac{\partial E}{\partial W}(m, c, p, q) = \sum_{h=0}^{H_{out}-1} \sum_{w=0}^{W_{out}-1} (X(c, h + p, w + q) * \frac{\partial E}{\partial Y}(m, h, w)) \quad \text{(B.23)}$$

```
1  void convLayer_backward_x_grad(int M, int C, int H_in, int
       W_in, int K, float* dE_dY, float* W, float* dE_dX) {
2     int H_out = H_in - K + 1;
3     int W_out = W_in - K + 1;
4     for(int c = 0;  c < C; c++)
5        for(int h = 0; h < H_in; h++)
6           for(int w = 0; w < W_in; w++)
7                dE_dX[c, h, w] = 0;
8     for(int m = 0;  m < M;  m++)
9        for(int h = 0; h < H-1; h++)
10          for(int w = 0; w < W-1; w++)
11             for(int c = 0;  c < C; c++)
12                for(int p = 0; p < K; p++)
13                   for(int q = 0; q < K; q++)
14                      if(h-p >= 0 && w-p >= 0 && h-p <
                             H_out and w-p < W_OUT)
15                         dE_dX[c, h, w] += dE_dY[m, h-p, w-
                               p] * W[m, c, k-p, k-q];
16 }
```

FIGURE B.10

$\frac{\partial E}{\partial x}$ calculation of the backward path of a convolutional layer.

```
1  void convLayer_backward_w_grad(int M, int C, int H, int W,
       int K, float* dE_dY, float* X, float* dE_dW) {
2     int H_out = H - K + 1;
3     int W_out = W - K + 1;
4     for(int m = 0; m < M; m++)
5        for(int c = 0; c < C; c++)
6           for(int p = 0; p < K; p++)
7              for(int q = 0; q < K; q++)
8                 dE_dW[m, c, p, q] = 0.;
9     for(int m = 0;  m < M;  m++)
10       for(int h = 0; h < H_out; h++)
11          for(int w = 0; w < W_out; w++)
12             for(int c = 0;  c < C; c++)
13                for(int p = 0; p < K; p++)
14                   for(int q = 0; q < K; q++)
15                      dE_dW[m, c, p, q] += X[c, h+p, w+q] *
                              dE_dY[m, c, h, w];
16 }
```

FIGURE B.11

$\frac{\partial E}{\partial w}$ calculation of the backward path of a convolutional layer.

Note that while the calculation of $\frac{\partial E}{\partial x}$ is important for propagating the gradient to the previous layer, the calculation of the $\frac{\partial E}{\partial w}$ is key to the adjustments to the weight values of the current layer.

After the $\frac{\partial E}{\partial w}$ values at all filter bank element positions are computed, weights are updated to minimize the expected error using the formula presented in Section B.2.1: $w \leftarrow w - \epsilon * \frac{\partial E}{\partial w}$, where ϵ is the learning rate constant. The initial value of ϵ is set empirically and reduced through the epochs according to the rule defined by user. The value of ϵ is reduced through the epochs to ensure that the weights converge to a minimal error. Recall that the negative sign of the adjustment term makes the change to be opposite to the direction of the gradient so that the change will likely reduce the error. Recall also that the weight values of the layers determine how the input is transformed through the network. The adjustment of these weight values of all the layers adapts the behavior of the network. That is, the network "learns" from a sequence of labeled training data and adapts its behavior by adjusting all weight values at all its layers for inputs whose inference result were incorrect and triggered back-propagation.

B.5 Summary

This appendix serves as an introduction to machine learning, in order to provide the necessary background for Chapters 19 and 20. It starts with some examples of machine learning tasks. Then, focuses on classifiers and introduces perceptrons, a type of linear classifier that is foundational for understanding modern neural networks, such as convolutional neural networks (CNNs). We discuss how the forward inference and backward propagation training passes are implemented for both single-layer and multi-layer perceptrons. In particular, we discussed the need for differentiable activation functions and how the model parameters can be updated through chain rules in a multi-layer perceptron network during the training process.

Based on the conceptual and mathematical understanding of perceptrons, we introduce convolutional neural networks and the sequential implementation of their inference phase and their training phase. In the inference phase, we put the emphasis on the convolutional layers and the subsampling layers, which are common types of layers in CNNs. In the training phase, we focus on the procedure to compute gradients that are necessary to update the weights of the neural network layers in the back-propagation process.

References

[1] A.L. Samuel, Some studies in machine learning using the game of checkers, IBM Journal of Research and Development 3 (3) (1959) 210–229.

[2] F. Rosenblatt, The Perceptron, a Perceiving and Recognizing Automaton, Tech. Rep. 85-460-1, Cornell Aeronautical Laboratory, 1957.

[3] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, L. Jackel, Handwritten digit recognition with a back-propagation network, Advances in Neural Information Processing Systems 2 (1989).
[4] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.
[5] G.E. Hinton, S. Osindero, Y.-W. Teh, A fast learning algorithm for deep belief nets, Neural Computation 18 (7) (2006) 1527–1554.
[6] R. Raina, A. Madhavan, A.Y. Ng, Large-scale deep unsupervised learning using graphics processors, in: Proceedings of the 26th Annual International Conference on Machine Learning, 2009, pp. 873–880.
[7] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, Advances in Neural Information Processing Systems 25 (2012).
[8] A. Krizhevsky, cuda-convnet, https://code.google.com/p/cuda-convnet/, 2014.
[9] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors, Nature 323 (6088) (1986) 533–536.

APPENDIX

CUDA memories, address spaces, and pointers

C

With special contributions from Isaac Gelado and Mark Harris

Throughout this book, we have mostly assumed a fairly simple model of interaction between the host and the device in a heterogeneous computing system. In this simple model, as presented in Chapter 2, each device has a device memory (CUDA global memory) that is separate from the host memory, or the system memory. The data to be processed by a kernel running on a device need to be transferred from the host memory to the device memory by calling the `cudaMemcpy()` function. The data produced by the device also need to be transferred from the device memory to the host by calling the `cudaMemcpy()` function before it can be utilized by the host. Early GPU generations only supported this simple host/device interaction model. Although the model is simple and easy to understand, it has resulted in several problems at the application level.

First, I/O devices such as disk controllers and network interface cards are designed to operate efficiently on the host memory. Since the device memory is separate from the host memory, input data need to be transferred from the host memory to the device memory and output data need to be transferred from the device memory to the host memory for use by I/O devices. Such additional transfers increase the I/O latency and reduce the achievable I/O throughput. For many applications, the ability of I/O devices to operate directly on the device memory would improve overall application performance and simplify the application code.

Second, traditional programming systems such as Python place their application data structures in the host memory. Some of the data structures are large. The device memory in CUDA-enabled GPUs tends to offer higher bandwidth but smaller capacity compared to the host memory, which forces application developers to partition their large data structures into chunks that each fit into the device memory. For example, in Chapter 21, the 3D electrostatic energy grid array was partitioned into 2D slices that are transferred between the host memory and the device memory. For many applications, it would be much better if entire data structures can reside in the device memory. For some applications, there may not even be a good way to partition the data structure into smaller chunks. For these applications, it would be best if the GPU can directly access the data in the host memory or have the CUDA runtime system software to migrate the data that are used during kernel execution.

As more applications adopted GPU computing, their needs motivated CUDA system software developers and GPU hardware designers to provide a better solution. Researchers have been aware of these needs and have proposed solutions since the early days of CUDA [1]. The rest of this section will go over a brief history of advancements that address these limitations.

C.1 Unified device memory address space

In early CUDA-enabled GPUs, shared, memory, local memory, and global memory formed their own separate address spaces. The developer could use pointers into the global memory but not others. Starting with the Fermi Architecture introduced in 2009, these memories became parts of a unified address space. This unified address space enabled a single set of load/store instructions and pointer addresses to access any of the GPU memory spaces (global, local, or shared memory), rather than different instructions and pointers for each. This made it easier to abstract away which memory contains a particular operand, allowing the programmer to deal with this only during allocation, and making it simpler to pass CUDA data objects to other procedures and functions, irrespective of which memory area they come from.

The unified device memory address space support made CUDA code modules much more "composable". That is, a CUDA device function can accept a pointer that may point to any of these memories. For example, without a unified GPU address space, a device function needs to have one implementation for each type of memory that one of its arguments can reside in. Unified GPU address space allows variables in all main types of GPU memories to be accessed the same way, and thus allowing one device function to accept arguments that can reside in different types of GPU memory. The code would run faster if a function argument pointer points to a shared memory location and slower if it points to a global memory location. The programmer can still perform manual data placement and transfers as a performance optimization. This capability significantly reduced the cost of building production quality CUDA libraries. It also enabled full C and C++ pointer support for CUDA kernel and device functions.

C.2 Zero-copy access to host memory

In 2009, CUDA 2.2 introduced zero-copy access to the host memory. This enables the host code to supply to a kernel a special device data pointer to host memory. The CUDA kernel code running on the GPU can use this pointer to directly access the host memory through the system interconnect such as the PCIe bus without calling `cudaMemcpy()`. Zero-copy memory is pinned host memory (see Chapter 23) and is allocated by calling `cudaHostAlloc()` with `cudaHostAllocMapped` as the value of the flag argument. We mentioned that the other values of the flag argument are for more advanced usage. The data pointer returned by `cudaHostAlloc()` cannot be directly passed to the kernel. The host code has to obtain a valid device data pointer using `cudaHostGetDevicePointer()` first, and pass the device data pointer returned by this function to the kernel. This means that different data pointers for host and device codes are used to access the same data in the host memory.

As we explained in Chapter 23, the host memory pages allocated for zero-copy access by the GPU must be pinned to prevent the operating system from accidentally paging out the data while the GPU is accessing them. Obviously, the access will suffer from the long latency and limited bandwidth of the system interconnect. The

bandwidth of the system interconnect is typically less than 10% of the global memory bandwidth. As we have learned in Chapter 5, a kernel's performance is typically limited by the global memory bandwidth unless we use tiling techniques to drastically reduce the number of global memory accesses performed per floating-point operation. If most of the memory accesses of a kernel are to zero-copy memory, the kernel's execution speed can be even more severely limited by the bandwidth of the system interconnect. Therefore, one should only use zero-copy memory for application data structures that are occasionally and sparsely accessed by a kernel running on a GPU.

C.3 Unified virtual address space

In 2011, CUDA 4 introduced the Unified Virtual Address Space (UVAS) which originated from the GMAC library [1]. Until this CUDA release, the host and the device had their own virtual address spaces, each of them mapping host or device data pointers to physical host or device memory locations. These disjoint virtual address spaces imply that the same physical memory location could be accessed by different virtual addresses in the host and the device, which happens when using zero-copy memory. UVAS specifies a single virtual address space shared by the host and the device. The UVAS guarantees that each physical memory address is only mapped to one virtual memory location. This enables the CUDA runtime to determine whether a data pointer is referencing host or device memory by just inspecting its virtual memory address. This feature removes the need to specify the data copy direction on `cudaMemcpy()` calls. Instead, the programmer can simply specify `cudaMemcpyDefault` and the direction of transfer is inferred from the values of the pointers.

It is important to note that the UVAS in CUDA 4 does not guarantee the accessibility of the data referenced by a pointer. For example, the host code cannot use a device pointer returned by `cudaMalloc()` to directly access the device memory and vice versa. Zero-copy access to the host memory is the exception: the host code can pass a pointer to zero-copy memory (allocated through `cudaHostAlloc()`) to the device as a kernel parameter. When the kernel code dereferences this zero-copy pointer, the pointer value is translated to a physical system memory location, and used to access the system memory directly through the PCIe bus. Note that this approach does not necessarily allow the kernel code to dereference a pointer value read from a memory location, such as following a chain of pointers while traversing a linked data structure, unless all pointers reference memory that has been allocated using `cudaHostAlloc()`.

The limitations in both the types of data structures that can be supported and the bandwidth of data accesses of zero-copy memory motivate further improvements in the memory model of GPU architectures beyond UVAS.

C.4 Large virtual and physical GPU address spaces

One fundamental limitation of early CUDA-enabled GPUs is the size of their virtual and physical addresses. These early devices support 32-bit virtual addresses and up to

32-bit physical addresses. For these devices, the size of the device memory is limited to 4 GB, the maximal amount of memory that can be addressed with 32 physical address bits. Furthermore, CUDA kernels could only operate on data sets whose sizes were less than 4 GB, the maximal number of virtual memory locations that can be accessed through 32-bit pointers, regardless of whether the data set resides in the host memory or the device memory. Furthermore, modern CPUs are based on 64-bit virtual addresses with 48 bits actually utilized. These host virtual addresses could not be accommodated by the 32-bit virtual addresses used by GPUs, which contributed to the limitation of the types of data structures supported by and the memory allocation mechanism (`cudaHostAlloc()`) required for zero-copy memory access.

To remove this limitation, GPU generations starting with the Kepler GPU architecture introduced in 2013 have adopted modern virtual memory architecture with 64-bit virtual addresses and physical addresses of at least 40 bits. The obvious benefit is that these GPUs can incorporate more than 4 GB of DRAM and that CUDA kernels can now operate on large data sets. While the enlarged virtual and physical address spaces obviously enable the use of large device memories, they also open the door for much better host/device interaction models. For example, the host and the device can now use exactly the same pointer value to access a piece of data, regardless of whether it is in the host memory or the device memory.

C.5 Unified physical address space

The large GPU physical address space allows the CUDA system software to place device memory of different GPUs in the system into a *unified physical address space*. The benefit is that one GPU can directly access the memory of any other GPU attached to the same PCIe bus and/or NVLink domain by simply dereferencing a data pointer mapped to the physical address of the other GPU. Prior to the Kepler GPU architecture, communication among different GPUs (e.g., halo exchange in the stencil example in Chapter 23) was only possible through device-to-device memory copies triggered by the host code. This resulted in extra memory consumed to store the data being copied from other GPUs and extra performance overheads due to host-device synchronization and the setup operations needed for the memory copies. Direct access to the memory of other devices in the system enables just passing a device pointer as a kernel call parameter so that the kernel code can use it to load and/or store the data that reside in the other GPU's device memory.

C.6 Unified memory

In 2013, CUDA 6 introduced Unified Memory, which creates a pool of managed virtual memory pages that are shared between the CPU and GPU, bridging the CPU-GPU divide. The managed memory is accessible by both the CPU and GPU using a single pointer. Variables in the managed memory can reside in the CPU physical

CPU code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);
  fread(data, 1, N, fp);
  qsort_char(data, N, 1);

  use_data(data);
  free(data);
}
```

CUDA 6 code with unified memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);
  fread(data, 1, N, fp);
  qsort_char<<<...>>>(data, N, 1);
  cudaDeviceSynchronize()
  use_data(data);
  cudaFree(data);
}
```

FIGURE C.1

Unified Memory simplifies porting of CPU code (left) to CUDA code (right).

memory, the GPU physical memory, or even both. The CUDA runtime software and hardware implement data migration and coherence support. The net effect is that the managed memory looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU. Of course, the application must perform appropriate synchronization operations such as barriers or atomic operations to coordinate any concurrent accesses to the managed memory locations. A shared global virtual address space allows all variables in an application to be accessed with the same virtual address from the CPU code or the GPU code. Such a memory architecture, when exposed by programming tools and run-time systems to applications, can result in several major benefits.

One benefit is the reduced amount of effort required for porting CPU code to CUDA. In Fig. C.1, we show a simple CPU code example on the left side. With Unified Memory, the code can be ported to CUDA with two simple changes. The first change is to use `cudaMallocManaged()` and `cudaFree()` in place of `malloc()` and `free()`. The second change is to launch a kernel and perform device synchronization rather than calling the `qsort()` function. Obviously, one still needs to write or have access to a parallel `qsort` kernel. What we are showing is that the change to the host code is straightforward and easy to maintain.

The performance of the CUDA 6 Unified Memory was limited by the hardware capabilities of the Kepler and Maxwell GPU architectures. The contents of all managed memory locations modified by the CPU had to be flushed out to the GPU device memory before any grid launch. The CPU and GPU could not simultaneously access a managed memory allocation and the Unified Memory address space was limited to the size of the GPU physical memory. These limitations are due to the fact that these GPU architectures lacked the ability to support coherence between the host and device memories and that the data migration was mostly performed by software.

C.7 Access to full host virtual address space

In 2016, the Pascal GPU architecture extended GPU address translation capabilities to accommodate 49-bit virtual addressing. This extension is large enough to cover the

48-bit virtual address spaces of modern CPUs, as well as the GPU's own memory. It allows Unified Memory programs to access the full address spaces of all CPUs and GPUs in the system as a single virtual address space, rather than be limited by the number of virtual address bits that can be handled by the GPU's address translation mechanism. As a result, the CPUs and GPUs can truly share the pointer values, enabling the GPUs to traverse pointer-based data structures in the host memory.

C.8 Page fault handling

Memory page fault handling support starting with the Pascal GPU architecture is a crucial feature that provides more seamless Unified Memory functionality. Combined with the system-wide virtual address space, the ability to handle page faults eliminates the need for the CUDA system software to synchronize (flush) all managed memory contents to the GPU before each grid launch. The CUDA runtime can implement a coherence mechanism by allowing the host and the device to invalidate each other's copies of data when they modify that data in the managed memory. The invalidation can be done using the page mapping and protection mechanisms. When launching a grid, the CUDA system software no longer has to bring all GPU copies of the managed memory data up to date. If the grid accesses a piece of data whose copy in the device memory has been invalidated by the host, the GPU will handle a page fault to bring the data up to date and resume execution.

If a thread grid running on the GPU accesses a page that is not resident in its device memory, it also experiences a page fault, allowing the page to be automatically migrated to the GPU memory on-demand. Alternatively, the page may be mapped into the GPU address space for access over the system interconnects if the data is expected to be accessed only occasionally (mapping on access can sometimes be faster than migration). This support generalizes zero-copy access to a limited part of the host memory access to zero-copy access to the entire host memory as well as other GPU's memory in the system. Note that Unified Memory is system-wide: GPUs (and CPUs) can fault and migrate memory pages either from CPU memory or from the memory of other GPUs in the system. If a CPU function dereferences a pointer and accesses a variable mapped to the GPU physical memory, the data access would still be serviced, but perhaps at a longer latency. Such capability allows the CUDA programs to more easily call legacy libraries that have not been ported to GPUs. In the prior CUDA memory architecture, the developer must manually transfer data from the device memory to the host memory in order to use legacy library functions to process them on the CPU.

The Unified Memory with the page fault handling capability enables a much more general CPU/GPU interaction mechanism than the original zero-copy access. It allows the GPU to traverse large data structures in the host memory. Starting with the Pascal architecture, a GPU device can traverse a linked data structure even if the data structure does not reside in zero-copy memory. This is because the same pointer value is used in the host code and device code to refer to the same variable. Thus,

the embedded pointer values of a linked data structure built by the host can be traversed by the device, and vice versa. In some application areas such as CAD, the host physical memory system may have hundreds of gigabytes of capacity. These physical memory systems are needed because the applications require the entire data set to be "in core". With the ability to directly access very large CPU physical memories, it becomes feasible for GPUs to accelerate these applications.

C.9 Virtual address space control

CUDA 11 introduced a set of low level APIs to provide programmers more flexibility regarding memory allocation. The new API allows reserving a range of the virtual address space using `cuMemAddressReserve()`. Later on, the programmer can allocate physical memory on any device using `cuMemCreate()`, and map it to any position in the reserved range using `cuMemMap()`. These APIs enable building custom layouts of data structures across multiple devices. For instance, it would be possible to allocate a 3D volume across multiple devices while using a single pointer to reference it.

References

[1] I. Gelado, J.E. Stone, J. Cabezas, S. Patel, N. Navarro, W.-m.W. Hwu, An asymmetric distributed shared memory model for heterogeneous parallel systems, in: Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2010, pp. 347–358.

This page intentionally left blank

Index

Symbols

__device__, 35
__global__, 35
__host__, 36
__syncthreads(), 70
__syncwarp(), 80

A

Accesses
  coalesced, 136, 150, 235, 272
  data, 126, 132
  DRAM, 132, 176, 179, 209
  memory, 4, 114, 128, 176, 208, 230, 258, 314, 364, 399, 418, 442, 521
  registers, 533
  sequential, 216
  uncoalesced, 269, 314, 420
Accuracy, 478
Achievable throughput, 210
Acquire semantics, 278, 279
Activation function, 602, 613, 614
Adjacency matrix elements, 428
Adjacent memory locations, 314, 413
Advanced
  optimization, 16, 64, 111, 149, 151–153, 349, 368, 399, 449, 482
  parallel patterns, 16
  patterns, 7, 578, 579
Aggressive loop unrolling, 370
Algorithm
  considerations, 593
  selection, 530
Alignment, 126
Allocated memory, 29, 30, 41, 253
Amdahl's law, 9, 537, 538
Application
  code, 12, 74
  level considerations, 537
  performance, 17
Application Programming Interface (API) functions, 6, 15, 28, 30, 40
Arbitrary length inputs, 244
Arithmetic accuracy, 591
Arithmetic and logic unit (ALU), 100
Arithmetic intensity (AI), 94, 115, 167, 171, 189, 272, 351, 352, 491, 492, 497, 504, 506, 539
Array
  device, 42, 552
  filter, 160
  multidimensional, 45, 50, 64
  structures, 151
Associative operator, 224
Asynchronous
  CUDA kernel calls, 556
  memory copies, 370
  memory operations, 562
Atomic operation, 12, 16, 201, 203, 204, 244, 245, 278, 279, 289, 291, 333, 405–407, 434, 441, 460, 518, 519, 533, 534
  coalesced, 291–293, 295, 442
  coalescing, 290, 291, 294, 442
  contention, 442
  hardware implementation, 213
  throughput, 209–211, 213
Attention, 478
  heads, 480
  mechanism, 478, 508
  sub-layers, 482
Auto-regressive models, 478
Automatic variable, 37, 86, 102, 108, 520
Autoregression phase, 485

B

Back-propagation, 608
Backward pass, 493
Backward-propagation path, 455
Bank conflict, 123, 135, 139–141, 151, 152, 272, 361, 362, 395
Barrier
  program location, 70
  scope, 70
Barrier synchronization, 14, 16, 69, 71, 111, 118, 146, 174, 191, 213, 240, 241, 244, 247, 257, 296, 297, 316, 353, 364, 366, 493, 499
  calls, 146
  CUDA, 73
  execution, 71
  operation, 366
  overhead, 238, 241, 258, 261
  points, 72
  program location, 71
Basic
  histogram kernel, 203
  kernel, 164, 187
  parallel merge kernel, 313
Basic Linear Algebra Subprogram (BLAS), 59

Batch, 610
Batched
  convolution, 472
  convolution layer, 459
  convolutional layer, 470
Batching, 492, 504, 539
Binary
  prefix sum, 293, 298
  search operation, 316, 327
Binning, 524
BLAS API, 368
Block, 24
  boundaries, 49
  cache, 126
  configuration, 91
  coordinate, 34
  dimensions, 46, 89, 106, 165, 177, 193, 268, 525
  index, 41, 45, 165, 215, 244, 245, 269, 277, 278, 353, 381, 460, 461
  level, 498
  scan function, 265
  segment, 215, 216, 268–270, 273, 275, 277
  size, 37, 46, 62, 74, 81, 85, 177, 179, 192, 193, 195, 257
  slots, 85
  sums, 273–276, 280
  thread, 24, 33, 63, 99, 124, 165, 168, 188, 191, 207, 208, 230, 237, 256, 257, 296, 315, 333, 350, 351, 379, 381, 442, 443, 459, 460, 486, 487, 519
Block-level
  input tile, 356
  output tile, 352
  tile, 359
  tiling, 384
Blockage
  constraint, 431
  vertex, 431
`blockDim`, 33
`blockIdx`, 34
Bottom warps, 82
Bottom-up, 374, 434
Boundary
  checks, 112
  elements, 548
  value, 382
  value problems, 410
Boundary condition control divergence, 82
Breadth-first search (BFS), 429
  edge-centric parallelization, 436
  kernel, 432, 445, 446
  tree, 432, 434
  vertex-centric parallelization, 431
Brent-Kung
  adder design, 282, 283
  algorithm, 282, 284, 285
Bubble sort, 331
Built-in variable, 33, 40

C

Cache
  block, 126
  lines, 126, 216, 340
  memory, 132, 519
Caching, 168
Call stack, 98
Caller thread, 312
Calling
  kernel functions, 37
  process, 547
  program, 40
Causality policy, 482
Central processing unit (CPU), 1
Chain rule, 608, 609
Circular-buffer merge kernel, 321
Classifiers, 602
Co-rank function, 306, 308, 309
  code, 325
  execution, 309
  implementation, 308
Coalesced
  access, 136, 150, 235, 272
  access pattern, 415
  atomic operations, 291–293, 295, 442
  group, 293, 294
  memory access patterns, 415, 416
  memory accesses, 347, 389, 415
  memory write patterns, 527
  pattern, 314
  storing, 360
Coalescing
  atomic operations, 290, 291, 294, 442
  hardware, 126
  memory, 123, 124, 129, 193, 216, 234, 298, 299, 314, 339, 340, 369, 410, 413, 521, 527
  memory accesses, 130, 193, 513
Coarsened thread
  block, 247
  granularity, 521
Coarsening
  factor, 141–143, 198, 215, 248, 267–269, 528
  kernel, 197
  loop, 141, 142, 219, 247
  thread, 73, 141, 142, 152, 190, 193, 194, 214, 246, 252, 267, 268, 298, 299, 327, 344, 345, 352, 488, 519, 521, 530, 535, 543

Code
complexity, 320
DCS, 515
device, 24, 39
examples, 9, 40, 142, 146, 164, 369, 438, 542
initialization, 548
kernel, 26, 35, 50, 52, 67, 69, 114, 117, 118, 169, 213, 230, 260, 266, 338, 352, 354, 376, 394, 432, 434, 436, 487, 496, 519, 521
parallel, 12, 21, 123, 204, 539, 577
parallelized, 15
performance, 15
sequence, 369
sequential, 4, 37, 207, 249, 266
snippet, 374, 376, 379
SpMV, 405
transformation, 17, 366
Collective communication, 552
Column-major layout, 51
Column-wise accessibility, 418
Communication
latency, 552
operations, 564, 575
Commutative operator, 225
Compare-and-swap, 441
Compiled kernel code, 364
Compiler
CUDA, 46, 172
optimizations, 224
Compressed Sparse Column (CSC) format, 418, 419
Compressed Sparse Row (CSR)
format, 407
kernel, 410
storage format, 407
Computation
complexity, 349, 536
convolution, 159
histogram, 203, 219, 534
kernel, 104
matrix multiplication, 474
overhead, 273
parallel, 16, 26, 201, 251, 285, 401
pattern, 16, 153, 187, 221, 460
reduction, 245
resources, 403, 459
sequential, 285, 524
stable filter, 535
stages, 366
stencil, 183, 189, 198, 352, 396, 542, 543, 548
steps, 282
tasks, 17, 554
throughput, 118, 219
time, 524
Computational
complexity, 11, 203, 253, 258, 309, 329, 523, 524, 526, 538
components, 510
cores, 15
intensity, 94, 154
load, 473
microscopes, 513
model, 7
problems, 532, 540
results, 513
solutions, 529
step, 483
tasks, 17
thinking, 14, 17
throughput, 94
Compute
capability, 88
utilization, 149
Compute-bound, 94
Compute-node-level tile, 359
Compute-to-global-memory-access ratio, 94, 96
Computer aided design (CAD), 430, 435
Consecutive
elements, 41, 127, 129, 130
global memory locations, 126
iterations, 376
keys, 297
memory available, 289
memory locations, 125, 345
output elements, 127
threads, 127–129, 239, 297, 299, 345, 395, 412
access consecutive data elements, 413
access consecutive elements, 129, 406
access data elements, 410, 420
load consecutive elements, 317
Consolidating block segments, 273
Constant
cache, 171
indices, 353, 354
memory, 16, 98, 103, 118, 153, 168, 169, 171, 189, 517–520
Context, 478
Context length, 482
Contiguous
block segment, 215
segments, 214, 216, 495
subsegment, 263, 269
thread segment, 215
Control divergence, 16, 77, 79, 80, 149, 230, 416
overhead, 270
performance impact, 82
thread, 81

Control flow, 79
Control hardware, 77
Conventional vector addition, 26
Convolution, 159
  blurring, 56
  computation, 159
  filter, 159, 160, 165, 167, 457
  filter bank, 463
  input tiles, 173
  kernel, 159, 160, 165, 167, 168, 191
  layer kernel, 463
  mask, 456
  memory bandwidth bottleneck, 172
  operation, 167, 469
  parallel, 164, 179
  pattern, 56, 151, 153, 463, 474
  result, 456
  tiled, 172, 178, 181
Convolutional
  filters, 473
  layer, 454–460, 463, 464, 467, 470, 473, 474, 605, 606
  layer output, 456
  network, 472
  routines, 472
Convolutional neural network (CNN), 17, 59, 179, 454, 455, 460, 612, 621
  implementers, 459
  inference, 613
  layers, 455, 613
  training, 616
Cooperating parallel workers, 4
Cooperative group, 70, 276, 277, 293, 398, 445, 446, 448
  API, 294, 295, 446
  API function, 448
Coordinate (COO)
  elements, 422
  format, 404
Corner turning, 129, 150, 272
Cost function, 607
CPU
  cache lines, 203
  hardware, 4
  serial code, 24
  system memory, 517
  thread, 112
Cubic blocks, 193
cuBLAS, 368
CUDA
  API functions, 28–30, 40, 558
  applications, 74, 87, 88, 90
  atomic reference class, 208
barrier synchronization, 73
C++ kernel function, 32, 35
C++ library, 207, 208
C++ program, 24, 39, 40, 45
C++ programmers, 67
C++ programming, 33, 40
code examples, 24
compiler, 46, 172
convolutional layer kernel, 460
core, 68
device, 36, 69, 74, 98, 99, 101, 124, 171, 266, 514, 533, 565
events, 559
examples, 18
extensions, 40
features, 40
global memory, 552, 623
GPU device, 266
grids, 48, 64
hardware, 16
host function, 36
implementation, 454, 474, 484, 486, 493
kernel, 34, 37, 39, 65, 86, 90, 93, 119, 120, 154, 160, 207, 379, 386, 392, 459, 474, 486, 545, 556
kernel function, 37, 93
kernel performance, 93, 124
keywords, 36, 39
language features, 15
libraries, 223, 249
linear algebra library, 464
memory, 28, 98, 230
memory types, 98, 118
model, 18
namespace, 207
occupancy API, 87
occupancy calculation functions, 445
operations, 555, 564
platform, 24
programmer, 13, 91, 98, 103, 118, 147
programming, 13, 17, 67
  environment, 31
  features, 529
  guide, 207, 230
  interface, 11
  model, 6, 14, 15, 552, 555, 575
programs, 7, 36, 39, 40, 55, 89
registers, 115
runtime, 88, 169, 172, 558, 563, 628
runtime system, 28, 33, 86
shared memory, 170
stencil computation, 541
streams, 555, 563, 564, 566, 575

style interfaces, 13
syntax, 102
systems, 35
thread, 77, 80, 528, 545, 547
tiled matrix multiplication kernel, 470
variables, 15, 98
CUDA-aware MPI, 552
`cudaFree`, 29
`cudaGetErrorString`, 32
`cudaMalloc`, 28
`cudaMallocHost`, 562
`cudaMemcpy`, 30
cuDNN, 368
CUDNN
convolution, 474
library, 472
Custom
CUDA matrix multiplication kernels, 486
kernel, 486
cuTile, 368
CUTLAS, 368
Cutoff, 524
Cutoff binning, 523

D

Data
access, 126, 132
access throughput, 15
element, 162, 210, 540
grid, 517
parallelism, 7, 15, 21, 23, 24, 248
reuse analysis, 350
size scalability, 523
throughput, 369
tiled, 463
transfer, 28
Deadlock, 72
Decoder layers, 478
Decoder-only models, 478
Decoding phase, 485
Decoupled lookback, 280
Default stream, 556
Delta value, 311
Denormalization format, 589
Designated subarrays, 320
Destination pointers, 31
Device
arrays, 42, 552
code, 24, 39
constant memory, 169, 517
CUDA, 36, 69, 74, 98, 99, 101, 124, 171, 266, 514, 533, 565
DMA, 563
driver, 88
function, 36, 40, 42, 103, 239, 263, 264, 269, 278, 353, 354, 361, 388, 389, 440, 441, 497, 498
global memory, 28, 30, 50, 517
GPU, 36, 537
implementation, 87
initialization, 511
kernel, 39, 119
memory, 27–29, 104, 552, 563
memory arrays, 30, 31
parallel kernel, 24
properties, 119
scope, 278, 279
thread, 36
version, 36
Digital twins, 8
Dimensions
block, 46, 89, 106, 165, 177, 193, 268, 525
grid, 460
thread block, 37
tile, 177, 350, 396
Direct Coulomb Summation (DCS), 514, 530
code, 515
function, 516
kernel, 517, 519, 521
method, 514
Direct Memory Access (DMA), 563
device, 563
hardware, 563
operation, 563
Direction-optimized, 436
Direction-optimized implementation, 436
Discriminative tasks, 478
Distance value, 381
Distributed memory model, 545
Distributed shared memory, 70, 102
Divergence pattern, 231
Divergent warp execution, 80
Domain decomposition, 13
Double data rate (DDR), 133
Double-buffering, 146, 147, 152, 258, 366
DPX instructions, 398
Draft model, 507
Dynamic programming, 373
Dynamic Random-Access Memory (DRAM), 5, 28, 68
access, 132, 176, 179, 209
architecture, 123
bandwidth, 10, 166, 170, 171, 179
banks, 132, 133, 139
burst, 125, 126, 132, 135, 137, 150, 209
bursting, 132, 139

cell, 124, 133
cell arrays, 134
chips, 124, 125
interface, 209
location, 125, 132
modules, 68
requests, 132
system, 132, 135, 137, 220
technology, 99, 139
traffic, 178

E

Efficiency
grid, 523
memory access, 303, 314, 404–406, 409, 412, 413, 415, 417, 418, 420
memory bandwidth, 421
parallel execution, 530
parallel scan, 269
tiling, 162, 164
Efficient parallelization, 11
Electrostatic potential map computation, 17
Elements
consecutive, 41, 127, 129, 130
filter, 167, 168, 179
halo, 192, 193
input, 255
neighboring, 161
segment, 257, 500
tile, 109, 110, 130, 319–321, 498, 500, 503
tiled, 322
values, 62, 174, 205, 283
vector, 27, 37, 38, 46, 458
warp access, 140
ELL
arrays, 413
format, 410
kernel, 411, 413, 415, 422
ELLPACK rormat, 410
Embarrassingly parallel applications, 12
Embedding layer, 480
Embedding vectors, 478, 481
Embeddings, 480
Encoder layers, 478
Energy grid, 519, 527
Epoch, 608
Error correcting code (ECC), 130
Error function, 607
Etching patterns, 580
Excess
elements, 414
encoding, 584
threads, 142
Exclusive scan, 253, 300, 337–341, 343–345
operation, 253, 298, 339, 342, 343, 345, 347
output, 253
Execution
resources, 234
thread, 2
Extracting parallelism, 333, 449

F

False dependence, 110, 146, 147, 258, 366
Favorable patterns, 126
Feature maps, 456, 613
Features, 456, 613
Feed-forward
networks, 612
sub-layer, 480, 481
Field Programmable Gate Array (FPGA), 7
Filter, 456
arrays, 160
bank, 613
elements, 167, 168, 179
kernel, 292, 295
operation, 289–291, 296, 300
pattern, 301
weight values, 172
Filtering pattern, 150, 152, 153
Fine-grained multithreading, 83
Flash attention, 491–494, 539
CUDA implementation, 494
formulation, 497, 510
kernel, 497, 504, 511
kernel code, 495
tiling, 495
Floating point
additions, 96, 272
ALU throughput, 370
ALUs, 366, 367
arithmetic instructions, 366
computations, 366
data representation, 583
elements, 272
operations, 119, 167, 351, 366, 506, 507, 519, 520
values, 38, 138, 139, 407
FLOPS, 93
Floyd-Warshall algorithm, 377, 378
Formulating convolutional layers, 463, 474
Forward pass, 493, 495
Forward path, 455
Foundational computation, 203
Fractional value, 22
Full host virtual address space, 627
Fully connected layer (FC), 605

Function
declaration, 36
declarations, 40
device, 36, 40, 42, 103, 239, 263, 264, 269, 278, 353, 354, 361, 388, 389, 440, 441, 497, 498
kernel, 15, 28, 30, 32, 35–37, 45–47, 50, 98, 102–104, 169, 228, 256, 319
values, 186
Fused kernel, 486, 492, 498

G

Gap penalty, 382
Garbage values, 395
Gather
access pattern, 534
approach, 518
kernel, 527
pattern, 536
General Matrix Multiply (GEMM), 350
Generation phase, 485
Generative tasks, 478
Generic class, 206
Generic objects, 206
Genome sequence alignment, 382
Get, 569
Ghost cells, 161
Ghost values, 179
Global index, 34
Global memory, 28, 68, 98, 139
access, 83, 97, 105, 125, 150, 196, 198, 236, 273, 274, 316, 320, 343, 492
access coalescing, 124
location, 126
space, 495
Global scan, 273
GNU Compiler Collection (gcc), 70
GPU
compute hardware, 554
constant memory, 517
device, 36, 537
device memory, 552
hardware, 93
memory, 509, 548
memory architecture, 67
neighboring, 545
NVIDIA, 368, 369
parallelization, 378, 384
programmers, 15, 16
threads, 33, 139, 144, 384, 409, 574
GPU Processing Cluster (GPC), 68
GPU-level tile, 359
Graphics Double Data Rate (GDDR), 68
Graphics Processing Unit (GPU), 3
Grid, 24, 33
array, 185, 191
computation efficiency, 531
configuration, 87, 493
data, 517
dimensions, 460, 556
efficiency, 523
indices, 461
launch, 40, 343
launch latency, 628
launch overhead, 448
points, 21, 184–187, 189–191, 514–521, 531, 533, 536, 542–545, 548, 549, 551, 554
relaunch, 378
space, 514, 524, 525
spacing, 185, 525, 543
structure, 519
system, 526, 527
thread, 27, 33, 35, 37, 69, 379, 517, 519, 556
values, 188
Grid-wide barrier synchronization, 70, 276, 445
Group calls, 567
Grouped-query attention (GQA), 508

H

Halo
cells, 172, 178, 179
elements, 192, 193
exchange, 544
input elements, 172
overhead, 192
points, 543, 544
Hardware
accelerators, 6
architecture, 14, 123
capability, 119
cards, 28
coalescing, 126
constraints, 192
CPU, 4
CUDA, 16
design, 135
efficiency reasons, 34
execution throughput, 89
expertise, 14
GPU, 93
implementations, 74, 101, 103
limitations, 118
limits, 92, 94, 121, 154
memory access, 5, 364, 366, 367
parallel, 530
peak, 96
platforms, 472

registers, 36
resources, 1, 69, 74, 88, 93, 94, 123, 143, 251
scheduler, 208, 214
support, 206, 368, 369
systems, 87
threads, 3
underutilization, 246, 248
vendors, 368
works, 14
Head dimension, 481, 484
Helper device functions, 239
Heterogeneous parallel computing, 3
Hidden dimensions, 482
Hiding memory latency, 132, 149, 154
Hierarchical technique, 360
Hierarchical tiling, 352, 359, 360
High-Bandwidth Memory (HBM), 68, 133
High-definition (HD), 8
High-performance computing (HPC), 13
High-performance parallel programs, 14
Histogram
computation, 203, 219, 534
kernel, 203, 208, 214, 216, 217, 220
pattern, 152, 153
Homology score matrix, 382
Hopper architecture, 68, 69
Hopper H100 GPU, 68, 93
Host code, 24, 26–28
Host memory, 624
Hybrid ELL-COO format, 414
Hybrid format, 414
Hyperplane
partitioning, 390
transformation, 389
Hyperscalers, 579
Hypertile, 390
approach, 392
computation, 396
implementation, 393–395
transformations, 399
width, 395
Hypertiling, 391, 396

I

Idempotence, 332, 433
Identity value, 221
IEEE format, 590
Image blur, 55
Image blurring function, 56
Implementation approaches, 374
Improved memory coalescing, 235
Improving memory coalescing, 298
In-flight batching, 506
In-place stable filter, 300
Inactive
threads, 75, 80, 230, 266, 286, 298
warp, 235
warp lanes, 285
Inclusive scan, 252, 254
Inclusive scan operation, 252, 253
Incomplete tiles, 391, 392, 394
Increment operation, 208, 209
Independent thread scheduling, 80
Indices
block, 41, 45, 165, 215, 244, 245, 269, 277, 278, 353, 381, 460, 461
constant, 353, 354
grid, 461
thread, 45, 60, 63, 75, 165, 171, 216, 229, 239, 297, 353, 470, 524
tile, 461, 463
Inference, 603
InfiniBand, 564
Inherent parallelism, 532
Initialization
code, 548
device, 511
operation, 559
Inner dimension, 356
Innermost loop, 516, 517
Input
elements, 214
grid, 570
tensor, 458, 615
tile, 172, 357
tile element, 176
Installed base, 6
Instruction
execution overhead, 139, 141, 142, 151, 369
scheduling, 144, 145
throughput, 136
Instruction Register (IR), 78, 79
Integer values, 38, 241, 304
Intensity values, 202
Inter-block scan, 273, 275
Interleaving memory access, 367
Intrinsic functions, 70, 294, 295, 500
Intrinsics, 70
Irregular access, 150, 314
Irregular access patterns, 150, 151
Iteration
loop, 62, 81, 142, 144, 146, 147, 167, 171, 216, 219, 257–259, 261, 265, 366
number, 338, 411
radix sort, 336, 339, 342, 345

J

Jacobi
- iteration, 573
- iterative method, 542, 545, 546, 549, 557, 563, 568, 571–574
- kernel, 548, 549, 555–557, 559, 561, 562
- method computation, 572
- solver, 561
- stencils, 397

Jagged diagonal format, 416
Jagged Diagonal Storage (JDS)
- arrays, 417
- format, 150, 151, 416–418, 421
- kernel, 417, 418, 535

K

Kernel
- arguments, 495
- boundaries, 492
- call, 24, 40, 41, 45–47, 117, 273, 274, 386, 388, 500, 501, 517, 555–561, 563, 564
- call overheads, 493
- calling overhead, 389, 396
- coarsening, 197
- code, 26, 35, 50, 52, 67, 69, 114, 117, 118, 169, 213, 230, 260, 266, 338, 352, 354, 376, 394, 432, 434, 436, 487, 496, 519, 521
- computation, 104
- computing, 558, 564
- convolution, 159, 160, 165, 167, 168, 191
- CUDA, 34, 37, 39, 65, 86, 90, 93, 119, 120, 154, 160, 207, 379, 386, 392, 459, 474, 486, 545, 556
- DCS, 517, 519, 521
- design, 515
- device, 39, 119
- execution, 40, 64, 103, 137, 168, 169, 171, 213, 220, 234, 395
- filter, 292, 295
- flash attention, 497, 504, 511
- function, 15, 28, 30, 32, 35–37, 45–47, 50, 98, 102–104, 169, 228, 256, 319
- fusion, 498, 574
- gather, 527
- histogram, 203, 208, 214, 216, 217, 220
- implementation, 176
- invocation, 104, 119, 519, 527
- invocation overhead, 497
- launch, 384, 574
- launch extensions, 40
- launch overhead, 472
- launch process, 172
- matrix multiplication, 62, 64, 92, 97, 114, 115, 141, 363, 364, 370, 474, 475
- merge, 314, 316, 322, 328
- parallel, 267
- parallel merge, 313
- parallel scan, 258, 268, 271, 284
- parameters, 172
- radix sort, 336
- reduction, 227, 237, 257, 270, 275
- scan, 270, 272, 273, 285
- signature, 165
- stable filter, 298
- stencil, 189, 542
- thread coarsening, 195, 196, 198
- tiled, 176, 179, 192, 320

Keys
- consecutive, 297
- mapping, 337, 338, 340
- performance considerations, 577

Kogge-Stone, 254
- adder design, 255
- algorithm, 254, 255, 260, 266, 282, 284
- kernel, 256, 260, 266
- parallel algorithm, 266
- parallel scan, 259
- parallel scan algorithm, 266

KV
- cache, 490
- cache arithmetic intensity, 504
- caching, 479, 488, 491, 492, 504

L

L2 norm, 542, 543, 545
Labeling blocks, 48
Lane, 238, 263
Lane index, 238, 239, 263, 264
Large Language Model (LLM), 477–479, 510
Latency, 209, 539
- bound kernel, 244
- hiding, 83
- memory, 139, 219
- memory access, 11, 118, 209, 210, 364
- optimization, 539
- requirement, 539
- synchronization, 149, 152
- time, 83
- tolerance, 15, 67, 82–85, 115
- waiting, 280

Launched
- thread, 573
- thread blocks, 387
- warps, 231

LDGSTS instruction, 369

Leader thread, 245, 278, 279, 292–294
Learning rate, 610, 621
Least-significant bit (LSB), 346
Least-significant digit (LSD), 346
Level value, 441
Linear algebraic formulation, 438
Linear algebraic formulation of graph problems, 438
Linear solvers, 594
Lines cache, 126, 216, 340
List elements, 305, 331, 332
Loads
  memory, 113, 174, 528
  uncoalesced, 130
  vector, 138, 151, 155, 272, 356, 369, 370
Local
  arrays, 146
  memory, 98, 102, 103
  variable, 37
Locality, 107
Logic operations, 553
Logical output tile, 361
Logit, 484
Logit element, 484
Lookback, 277
Loop
  body, 145, 147, 253, 260
  coarsening, 141, 142, 219, 247
  header, 260
  implements, 488
  interchange, 359, 516
  iteration, 62, 81, 142, 144, 146, 147, 167, 171, 216, 219, 257–259, 261, 265, 366
  nest, 111, 167, 171, 175, 176, 359
  nesting, 180
  parallelism, 37
  thread coarsening, 146
  unrolling, 123, 144–146, 149, 270, 271, 363, 364
  variable, 110
Lower-precision formats, 592
Luminance values, 22

M

`malloc`, 28
Managing parallelism, 14
Many-thread trajectory, 3
Mapping threads, 48
Marker values, 311
Massive
  data parallelism, 124
  parallelism, 5
Massively parallel
  computation, 425
  computing, 18, 24, 251, 577
  computing systems, 578
  processors, 13, 14
  program, 209
  programming, 18
Matrix
  elements, 64, 65, 96, 107, 468, 494, 539
  multiplication, 45, 59, 60, 96, 127, 349, 350
    computation, 474
    computational engine, 474
    example, 63, 104, 105, 117, 128
    formulation, 465, 467
    functions, 97
    implementation, 64, 97
    kernel, 62, 64, 92, 97, 114, 115, 141, 363, 364, 370, 474, 475
    kernel function, 155
    operations, 96, 97, 350, 368, 463
    results, 503
    routines, 368
    throughput, 112, 368, 369
Maximum
  element value, 484, 487
  occupancy, 363
  threads, 86, 116
Memoization, 374, 375
Memory, 68
  accelerator, 370
  access, 4, 114, 128, 176, 208, 230, 258, 314, 364, 399, 418, 442, 521
    bandwidth, 135
    behavior, 531, 533
    channels, 4, 84
    coalescing, 126, 193, 513
    divergence, 234
    efficiency, 303, 314, 404–406, 409, 412, 413, 415, 417, 418, 420
    hardware, 5, 364, 366, 367
    instructions, 364, 366, 368–370, 519
    latency, 11, 118, 209, 210, 364
    overhead, 146
    pattern, 112, 113, 127, 128, 150, 216, 360, 404, 407, 410, 475
    requests, 136
    speed, 11
    throughput, 4, 5, 124, 137, 209
  addresses, 126, 562
  alignment, 126, 301
  allocation, 28, 29, 253, 570
  architecture, 67
  area, 549
  array, 300, 526
  available, 563

bandwidth, 5, 93, 94, 111, 130, 133, 136, 138, 167, 176, 185, 272, 273, 314, 320, 327, 364, 370, 401, 403, 410, 475, 492, 523, 578
  bottleneck, 351
  considerations, 167, 189, 272
  consumption, 463
  efficiency, 421
  limitations, 10
  resources, 96
  usage, 119
  utilization, 138, 189
  utilization efficiency, 316
banks, 139, 363
bottleneck, 170
bound, 167
box, 99
buffer, 147, 259, 366, 563
cache, 132, 519
capacity, 401, 403, 506
channel, 68
coalescing, 123, 124, 129, 150, 193, 216, 234, 298, 299, 314, 339, 340, 369, 410, 413, 521, 527
coalescing behavior, 344
constant, 16, 98, 103, 118, 153, 168, 169, 171, 189, 517–520
controller, 68, 132
copy, 30, 169, 555, 560, 561, 563
copy operation, 560, 563
CUDA, 28, 98, 230
device, 27–29, 552, 563
fence operation, 230
GPU, 509
hierarchy, 360, 422
instructions, 279, 364
latency, 139, 219
latency hiding, 123
load, 113, 174, 528
load operation, 100, 101
location, 111, 113, 131, 147, 150, 204, 206, 209, 210, 235, 293, 298, 300, 333, 340, 360, 366, 395, 411, 434, 543, 563
management, 290, 562
model, 333
parallelism, 154
read accesses, 133
requests, 150, 340
requirements, 504–506, 508, 509
resources, 68, 90, 147, 364
size, 505
space, 101, 126, 543, 545, 570
stores, 209, 297
structures, 94, 360
  subsystem, 149
  transactions, 126
  update, 524
  usage, 115, 170, 290, 509
  utilization, 149–151
  variable, 168
Memory-bound, 94, 272
Merge
  kernel, 314, 316, 322, 328
  operation, 289, 303–306, 308, 333, 334, 347, 499, 503, 535
  parallel, 16, 308, 313, 328, 535
  pattern, 16, 150, 314, 329
  sort, 16, 304, 329, 330, 333, 334, 346, 374, 530, 535
Message Passing Interface (MPI), 12, 541, 545
  exchange operations, 562
  non-blocking communication, 551
  processes, 546
  rank, 547
Mini-batch, 611
Mixture of Experts (MoE) technique, 510
Model depth, 481
Model service optimization, 507
Most-significant digit (MSD) radix sort, 346
Movement patterns, 301
Multi-core trajectory, 3
Multi-GPU stencil, 545, 563, 568
Multi-head attention, 480, 482
Multi-layer classifiers, 603
Multi-layer perceptron (MLP), 604
Multi-modal models, 477
Multi-query attention (MQA), 508
Multidimensional
  arrays, 45, 50, 64
  data, 48, 64
  grid, 64
  grid organization, 45
  indices, 64
Multiple lookback, 279
Multiple-Instruction-Multiple-Data (MIMD), 79
Multiple-Instruction-Single-Data (MISD), 79
Multithreaded manner, 368

N

Naïve parallel algorithm, 265
Naïve parallel scan, 280
National Institute of Health (NIH), 6
Natural language processing (NLP), 477
NCCL
  communication
    operations, 565
    primitives, 564

system, 564
communicator, 565, 566
example, 573
group calls, 567
library, 575
primitives, 566, 567
ranks, 565
user guide, 566
Necessitating optimizations, 251
Neighboring
blocks, 178
elements, 161
GPUs, 545
grid points, 187, 188, 521, 542
threads, 129
tiles, 178, 390
Nested loops, 58, 59, 354, 378, 499, 501, 511, 518
Normalized representation, 584
Number of ranks, 548
Numerical
grids, 187
stability, 594
values, 304
NVIDIA Collective Communications Library (NCCL), 17, 541, 545, 562–564
API, 564
communication system, 564, 566
NVIDIA CUDA Compiler (NVCC), 39, 369
NVIDIA GPUs, 368, 369
NVIDIA OpenSHMEM Library (NVSHMEM), 541, 568, 569
API functions, 570, 572
memory access, 574
NVLINK, 564

O

Object code, 36
Occupancy, 85, 90, 135, 144, 149, 363
analysis, 86
calculator, 87, 90
considerations, 363
level, 92
maximum, 363
thread, 115, 138, 144, 149, 363
tuning, 115
Odd-even sort, 331
Odd-even transposition sort, 331
Off-chip memory, 68
Offset arrays, 427
Offset value, 100
On-chip memory, 68
One-sided communication, 569, 570, 575
Open Compute Language (OpenCL), 13
OpenSHMEM, 569
Operand value, 100
Operating system, 562
Operation
convolution, 167, 469
DMA, 563
filter, 289–291, 296, 300
initialization, 559
merge, 289, 303–306, 308, 333, 334, 347, 499, 503, 535
parallel merge, 334, 347
parallel scan, 251, 256
reduction, 221, 223, 224, 240, 280, 543, 545, 552, 553
scan, 252, 261, 262, 265, 267, 270, 272, 273, 276, 277, 298, 299, 301, 535
SpMV, 414, 438
stable filter, 289, 297, 301, 336
stencil, 574
store, 130, 167, 171
wait, 561, 562
waiting, 560, 561
Optical character recognition (OCR), 454, 612
Optimal substructure, 374
Optimization
latency, 539
opportunities, 349
performance, 15, 96, 153
privatization, 16, 153, 291
skills, 538
steps, 540
strategies, 16, 153, 276, 540
techniques, 11, 15, 176, 201, 219, 223, 249, 254, 350, 492, 513, 539
thread coarsening, 352, 519
tiling, 172
Optimized kernel, 273
Optimized sequential code, 518
Optimizing code, 129, 577
Optimizing parallel patterns, 286, 347
Ordered merge function, 303
Ordering operation, 304
Output
arrays, 167
element, 175
grid, 570
tile, 172, 360
Overarching goals, 14
Overhead
computation, 273
halo, 192
memory access, 146
parallelization, 141, 152, 153, 193, 246, 267

privatization, 214
synchronization, 12, 141, 146, 152, 237, 241, 258, 261, 273, 280, 300, 562
Overlapping computation, 541, 553, 554, 559, 563, 567, 575
Owner computes, 228

P

Packing, 150
Padded elements, 411–414
Padding, 140, 395
Padding elements, 362, 410–413, 415
Page fault handling, 628
Page-locked memory, 562
Parallel
algorithms, 11, 16, 17, 102, 153, 159, 179, 226, 251, 253, 265, 266, 401, 529, 530
application, 3, 9, 15, 118, 422
application development, 538
approach, 523
array, 480
BFS, 439, 450
code, 12, 21, 123, 204, 539, 577
computation, 16, 26, 201, 251, 285, 401
computation patterns, 12, 154, 159, 179, 308, 577
compute units, 376
computers, 3
computing, 7, 14, 40, 159, 164, 211, 401, 532, 538, 539, 577, 579
capabilities, 10
concepts, 40
device, 15
devices, 580
pattern, 16
primitives, 410
revolution, 579
system, 9
systems, 6, 33, 118, 221, 519
techniques, 10
convolution, 164, 179
CUDA program, 15
execution, 7, 9, 12, 13, 24, 55, 124, 137, 216, 249, 266, 329, 374, 518, 523, 524, 530–532
efficiency, 530
model, 15
performance, 3
resources, 143, 231
forms, 11
hardware, 530
hardware architectures, 14
hardware execution resources, 248
histogram approach, 207
histogram computation, 218
implementations, 9, 266, 270, 282, 289, 302, 390, 429, 431, 432, 435
iterations, 460
kernel, 267
libraries, 438
loops, 460
memory organization, 136
merge, 16, 308, 313, 328, 535
implementation, 347
kernel, 313
operation, 334, 347
sort, 304, 333, 334, 346, 347
nature, 376
odd-even sort, 331
organization, 132
patterns, 7, 12, 14, 15, 64, 123, 151, 221, 286, 303, 396, 401, 421, 422, 529, 577, 579
portion, 9, 10
program, 2, 3, 11, 12, 21, 111, 123, 254, 539, 540
programmer, 13, 21, 110, 401, 523, 529, 530
programming, 2, 3, 7, 11, 14, 118, 529, 531, 578
details, 12
interfaces, 12, 15
languages, 12, 21
models, 11, 15, 123
optimization techniques, 539
projects, 6
styles, 17, 33
techniques, 40
radix sort, 336
reduction, 221, 223, 224, 246, 249
algorithms, 223, 224
implementations, 225
pattern, 225, 249
tree, 226, 230, 254
tree algorithm, 226
scan, 251, 254, 256, 259, 261, 267, 270, 280, 282
algorithms, 253, 286
efficiency, 269
kernel, 258, 268, 271, 284
operation, 251, 256
run, 251
software, 2, 6
sort methods, 345
sorting, 347
sorting algorithms, 16, 329, 331, 347
sorting networks, 345
SpMV, 410, 411, 420–422
stable filter, 297
stencil, 187

sum reduction tree, 227
systems, 14
thinking, 540
thinking skills, 529
thread, 5, 12, 15, 221, 305
thread execution, 136
units, 193
unstable filter, 290
vector addition kernel, 226
work multiple units, 152
Parallelism
data, 7, 15, 21, 23, 24, 248
level, 244, 454, 460, 528
loop, 37
massive, 5
memory, 154
wavefront, 17, 373, 376, 377
Parallelizable
portion, 9
sorting algorithm, 304
Parallelization
approach, 306, 337, 405, 408, 419
arrangement, 165
GPUs, 378, 384
opportunities, 384
overhead, 141, 152, 153, 193, 246, 267
strategy, 204
wavefront, 384
Parallelized
code, 15
merge operation, 308
Parallelizing
applications, 518
computations, 17
efficient, 329
graph algorithms, 17
graph traversal, 449
merge overhead, 327
merge sort, 333
radix sort, 336
radix sort overhead, 344, 345
scan, 282
scan overhead, 267
SpMV, 407
useful applications, 14
vector addition overhead, 142
work, 141
Parallelogram, 390
Parallelogram hypertiles, 390
Parameter values, 32, 209, 453, 454
Partial differential equation (PDE), 179
Partial wave, 73, 143
Partition pattern, 336
Patch pixels, 58
Pattern
coalesced, 314
computation, 16, 153, 187, 221, 460
convolution, 56, 151, 153, 463, 474
filter, 301
gather, 536
histogram, 152, 153
memory access, 112, 113, 127, 128, 150, 216, 360, 404, 407, 410, 475
merge, 16, 150, 314, 329
parallel computing, 16
parallel reduction, 225, 249
recognition, 16, 453
scan, 16, 290
stable filter, 329, 336, 534, 535
wavefront, 376, 535
PCIe, 564
Peak
computational throughput, 94, 118
hardware, 96
memory bandwidth, 94, 96
performance, 3, 4, 111, 272, 274, 276
throughput, 97, 111
Perceptron, 602
Performance
benefit, 141, 143
bottleneck, 14, 153, 154, 291
code, 15
considerations, 16, 123, 154, 166, 540
enhancements, 124
impact, 82
improvement, 1, 2, 112
limiter, 93
objective, 152
optimization, 15, 96, 153
optimization techniques, 140, 331, 529
parallel execution, 3
peak, 3, 4, 111, 272, 274, 276
penalty, 130, 143
portability, 13
requirements, 74
stable filter, 300
tuning, 17, 124
Periodic boundary condition technique, 551
Persistent grid, 493
Physical
GPU address spaces, 625
memory, 562
page, 562
Pinned memory, 562
allocation, 562
buffer, 563

Pipeline latency, 136, 149
Pivoting, 598
Point-to-point communication, 549
Pointer, 26
Pointer variable, 26, 29, 30, 50
Pooling layer, 616
Positional Encoding layer, 480
Prefill phase, 491
Prefix
  subarrays, 308
  sum, 252, 337
  sum operation, 293, 298
Private copy, 102, 152, 211–213, 442
Privatization, 152, 211, 295, 442
  optimization, 16, 153, 291
  overhead, 214
Privatized histogram kernel, 213
Problem decomposition, 532
Problem formulation, 538
Problematic accesses, 113
Processing blocks, 76, 83
Processing element (PE), 570, 574
Processor
  hardware, 170
  installed base, 6
Program code, 2
Program Counter (PC), 78
Programmer
  CUDA, 13, 91, 103, 118, 147
  parallel, 13, 21, 110, 401, 523, 529, 530
Prospective device, 89
Provisioned access throughput, 132
Public copy, 152, 153, 211, 213, 214, 442
Pull, 434
Pull implementation, 434
Push, 432
Push implementation, 432
Put, 569

Q

Qualifier keywords, 35
Queried device, 89
Querying device properties, 87

R

Race condition, 205, 257, 331, 332, 433
Radix
  sort, 251, 329, 330, 334–336, 340, 342, 344–346, 530, 535
    iteration, 336, 339, 342, 345
    iteration kernel, 340
    kernel, 336
  sorting algorithms, 346
  value, 335, 342, 344
Ranks number, 548
Read-after-write dependence, 110, 146, 147
Read-modify-write, 204
Read-modify-write race condition, 205
Receiving thread, 240
Rectangular tiles, 390, 399
Reduce operation, 276
Reduce-scan-scan, 275
Reduce-scan-scan decomposition, 275, 276, 280
Reduced memory traffic, 104
Reduction
  computation, 245
  kernel, 227, 228, 237, 257, 270, 275
  operation, 221, 223, 224, 240, 280, 543, 545, 552, 553
  parallel, 221, 223, 224, 246, 249
  tree, 223, 225–228, 238, 244, 246, 255, 258, 276, 282–284
Redundant
  accesses, 358
  computation, 510
  global memory accesses, 273, 285, 539
Register, 98
  accesses, 533
  CUDA, 115
  hardware, 36
  thread, 500
  tiling, 112, 123, 149, 151, 190, 196, 252, 270, 271, 357–359, 497, 539
  tiling kernel, 197
Reinforcement learning, 606
Related patterns, 301
Relative performance, 401
Release semantics, 279
Representable numbers, 586
Reset operation, 561
Resident warps, 90
Residual, 542
Residual connection, 481
Resource partitioning, 85
Resource utilization efficiency, 231
Retrieval-augmented generation (RAG), 481
Return value, 29, 209, 240, 293, 308, 442
Returned value, 42, 209, 293, 320
Revised kernel, 117
RGB values, 22, 23
Roofline model, 94
Row-major layout, 51
Runtime API, 40

S

Sample, 459
Scalability, 23
Scalable parallel execution, 21
Scan, 251
  kernel, 257, 270, 272, 273, 285
  operation, 252, 261, 262, 265, 267, 270, 272, 273, 276, 277, 298, 299, 301, 535
  parallel, 251, 254, 256, 259, 261, 267, 270, 280, 282
  parallelizing, 282
  pattern, 16, 290
Scan-scan-add, 261, 275
Scan-scan-add decomposition, 261, 276, 280
Scanned
  block segment, 263, 268, 269, 281
  block sums, 273, 274
  segment, 256, 261, 263, 273
  thread sums, 269, 270, 281
  warp, 265
Scatter
  access pattern, 534
  approach, 517
  kernel, 527
Scattered accesses, 150
Scope, 70
Scoring matrix, 382
Scratchpad memory, 101
Segment
  block, 215, 216, 268–270, 273, 275, 277
  elements, 257, 500
  scanned, 256, 261, 263, 273
  size, 215, 244, 269
  starting location, 245
  thread, 215, 272, 281
  variable, 244
Semi-supervised learning, 606
Sequence length, 482
Sequential
  accesses, 216
  code, 4, 37, 207, 249, 266
  computation, 285, 524
  implementation, 613
  merge algorithm, 305
  operations, 251
  reduction operation, 226
  scan operation, 269
  sum reduction code, 222
Serialized operations, 209
Shaded keys, 290
Shared memory, 69, 99, 139, 151, 152, 236
  bank conflicts, 139, 499
  model, 545
  tiling, 104, 123, 151, 153, 168, 171–173, 189, 198, 349, 358–360, 463, 497
  tiling kernel, 192, 195
  variables, 103
Shear factor, 391
Shear transformation, 390
Shuffle data, 238
Sigmoid function, 609
Signal values, 159
Similarity score, 382, 388
Single Instruction, Multiple Thread (SIMT), 76, 77
Single lookback, 277
  method, 277–279
  synchronization, 400
Single Program Multiple Data (SPMD), 15
Single-block reduction kernel code, 245
Single-Instruction-Multiple-Data (SIMD), 79
  hardware, 74, 77, 79, 80, 234
  model, 76, 90
Single-Instruction-Single-Data (SISD), 79
Single-Program Multiple-Data (SPMD), 32, 33
Skeleton code, 38
Small dynamic programming problems, 398
Smith-Waterman algorithm, 382, 384
Softmax function, 478, 484, 490
Software pipelining, 364, 366
Sorted
  input arrays, 304
  segments, 333
  tiles, 346
Sorting pattern, 150, 153, 530
Source code, 36, 38, 52, 60, 552
Space efficiency, 403, 410, 414, 416
Span, 225, 254
Sparse Matrix Vector (SpMV)
  calculation, 414
  computation code, 405
  kernels, 404, 421
  multiplication, 403
  operation, 414, 438
  parallel, 410, 411, 420–422
  parallelizing, 407
Special bit patterns, 590
Specialized hardware, 368, 370
Specialized software, 368
Speculation depth, 507
Speculative decoding, 492, 504, 507
Speed-of-light analysis, 96
Speedup, 9
Spilled registers, 98
Split-K, 498
Stability, 289, 304, 334
Stable filter, 16, 299, 301

computation, 535
implementation, 298, 300
kernel, 298
operation, 289, 297, 301, 336
parallel, 297
pattern, 329, 336, 534, 535
performance, 300
process, 299
Stable partition, 336
State-space models, 478
Static Random Access Memory (SRAM), 69
Statically allocated arrays, 50, 98
Stencil, 542
computation, 183, 189, 198, 352, 396, 542, 543, 548
grid point pattern, 191
kernel, 189, 542
operation, 574
patterns, 55, 151, 153, 183, 193, 196, 198
sweep, 189
Step, 225
Stochastic gradient descent, 607
Storage overhead, 404
Store
instructions, 138, 142, 360, 361
operation, 130, 167, 171
Storing coalesced, 360
Stream, 555
Streaming Multiprocessor (SM), 68, 69, 89, 93
Streaming processor, 68
Stride value, 232, 233, 257, 263
Strip-mining, 111
Structure of arrays, 151
Stub, 31
Subarrays, 308, 314, 315
Subsampling layer, 616
Subsegment
contiguous, 263, 269
location, 269
sequential scan, 268
Summarization phase, 485
Supervised training, 606
Swap operation, 442
Symmetric heap, 570
Symmetric memory allocation, 570
Synchronization, 149, 397, 559
latency, 149, 152
operations, 12, 559, 568
overhead, 12, 141, 146, 152, 237, 241, 258, 261, 273, 280, 300, 562
Synchronous Dynamic Random Access Memory (SDRAM), 68

T

Tabulation, 374
Tail effect, 74, 143
Target model, 507
Task parallelism, 23
Template, 206
Temporary values, 363
Tensor core, 97, 368
Tensor Memory Accelerator (TMA), 369, 504
Terra floating-point operations per second (TFLOPS), 94
Thread, 24
assignment, 388
block, 24, 33, 63, 99, 124, 165, 168, 188, 191, 207, 208, 230, 237, 256, 257, 296, 315, 333, 350, 351, 379, 381, 442, 443, 459, 460, 486, 487, 519
array, 33
assignment, 388
cluster, 69, 102, 213
configuration, 137
dimensions, 37
local buckets, 340
private output array, 295
scheduling, 67, 69
scope, 73
size, 38, 41, 81, 91, 176, 195, 246, 328, 363, 385
coarsening, 73, 141, 142, 152, 190, 193, 194, 214, 246, 252, 267, 268, 298, 299, 327, 344, 345, 352, 488, 519, 521, 530, 535, 543
kernel, 195, 196, 198
loop, 146
optimization, 352, 519
control divergence, 81
CPU, 112
CUDA, 77, 80, 528, 545, 547
device, 36
divergence, 81
execution, 258
execution patterns, 16
granularity, 16
granularity coarsening, 123, 154, 246
grid, 27, 33, 35, 37, 69, 379, 517, 519, 556
hierarchy, 352
indices, 45, 60, 63, 75, 165, 171, 216, 229, 239, 297, 353, 470, 524
assignment, 249
calculation, 207
values, 81, 342, 444
labels, 48, 433, 435, 436
loops, 408, 419, 435
memory accesses, 126

occupancy, 115, 138, 144, 149, 363
organization, 34, 48, 173, 174, 177, 460, 461
parallel execution, 135
registers, 500
scheduling, 67, 74, 81
segment, 215, 272, 281
slots, 85, 86, 116
subsegments, 268, 271, 272, 281
sums, 269, 270, 281
warp, 76
Thread-level
input tile, 356
output tile, 356
privatization, 217
tile, 359
`threadIdx`, 34
Threading, 32
Throughput, 539
atomic operations, 209–211, 213
computation, 118, 219
computational, 94
data, 369
matrix multiplication, 112, 368, 369
memory access, 4, 5, 124, 137, 209
peak, 97, 111
Tile
anti-diagonals, 397
boundaries, 474
bounds, 388
cells, 388, 394, 395
column, 386, 387, 391, 392
column index, 386, 392
configuration, 468
dimensions, 177, 350, 396
elements, 109, 110, 130, 319–321, 498, 500, 503
indices, 461, 463
loading code, 468
loading process, 320
size, 107, 130, 180, 192, 193, 195, 320, 352, 472
width, 112, 470
wrap, 323
Tiled
algorithm, 111, 118
convolution, 172, 178, 181
convolution kernel, 173, 176, 177, 191
data, 463
elements, 322
kernel, 176, 179, 192, 320
kernel execution, 113
matrix multiplication, 112, 116, 155, 173, 351, 352, 357, 358, 467, 468
matrix multiplication kernel, 108, 112, 114, 142, 470
merge kernel, 314–316, 328
parallel convolution implementation, 179, 180
stencil kernel, 192
stencil sweep kernel, 191
Tiling, 104, 112
approach, 495
concept, 104
data, 151
design, 316
effectiveness, 112
efficiency, 162, 164
flash attention, 495
input feature maps, 463
optimization, 172
registers, 112, 123, 149, 151, 190, 196, 252, 270, 271, 357–359, 497, 539
techniques, 112, 119, 124, 183, 185, 303, 327, 422, 463
Token, 477
Token sequence, 479
Tokenizer, 479
Top-down, 374, 432
Traceback process, 384
Trailing thread, 206, 209
Training models, 606
Training multi-layer classifiers, 611
Transformer
architecture, 478
latency, 539
layers, 480
pass, 491, 492
Transparent scalability, 70, 74
Tree operations, 251
Tree pattern, 16, 305
True dependence, 110, 146, 147, 258, 366
Two-sided communication, 549, 568

U

Un-embedding layer, 481
Uncoalesceable accesses, 340
Uncoalesced
accesses, 269, 314, 420
loads, 130
memory accesses, 130, 340, 376
stores, 130
Uncoarsened kernel, 143
Unidirectional synchronization, 277
Unified
device memory address space, 624
memory, 626
physical address space, 626
Unified Virtual Address Space (UVAS), 625
Unified Virtual Address (UVA), 35

Unoptimized sequential code, 518
Unrolled loop, 144
Unrolling factor, 144
Unrolling loops, 149
Unshaded keys, 290
Unstable filter, 16, 290
  kernel, 290–292, 294, 296
  operation, 291
  pattern, 16
Unstructured grids, 184
User code, 323, 324

V

Value radix, 335, 342, 344
Vector
  addition, 26, 28, 31, 41, 96, 151
  addition kernel, 37, 96, 138, 142, 155
  addition kernel function, 35
  elements, 27, 37, 38, 46, 458
  loads, 138, 151, 155, 272, 356, 369, 370
  operations, 438
  stores, 138, 272, 369
Virtual
  address space, 562
  address space control, 629
  memory location, 563
  page, 562
von Neumann model, 99

W

Wait operation, 561, 562
Waiting
  latency, 280
  operation, 560, 561
  time, 72, 73, 398
Warp, 74
  configurations, 77
  divergence, 390
  executions, 234, 286
  index, 241
  instructions, 84
  level, 237, 284, 498
  match functions, 241
  partitioning, 77
  reconverge, 81
  reduce functions, 241
  scanned, 265
  scheduling, 82–84
  shuffle functions, 237, 241
  size, 75, 130, 239, 241, 286
  specialization, 367, 368, 370, 504
  subsegments, 263
  sums, 263, 265
  threads, 76, 293
  voting, 241, 289, 295
  voting functions, 241
Warp Group Matrix Multiply Accumulate (WGMMA), 369
Warp-level primitives, 77, 152, 237, 261
WarpReduce namespace, 511
Wasted operations, 427
Wave, 73, 141, 143, 376
Waveform parallelism, 373
Wavefront, 374, 376
  parallelism, 17, 373, 376, 377
  parallelization, 384
  parallelization performance, 389
  pattern, 376, 535
Wiring blocks, 431
Work, 225, 254
Work assignment, 228
Work efficiency, 11, 226, 265
  considerations, 265
  improvement, 438
Write-after-read dependence, 110, 146, 147

Z

Zero-copy access, 624
Zero-copy memory, 624
Zero-overhead thread scheduling, 84

Programming Massively Parallel Processors

A Hands-on Approach FIFTH EDITION

Wen-mei W. Hwu, NVIDIA and University of Illinois at Urbana-Champaign
David B. Kirk, Former NVIDIA Fellow
Izzat El Hajj, American University of Beirut

Programming Massively Parallel Processors: A Hands-on Approach, Fifth Edition shows both students and professionals alike the basic concepts of parallel programming and GPU architecture. Concise, intuitive, and practical, it is based on years of road-testing in the authors' own parallel computing courses. Various techniques for constructing and optimizing parallel programs are explored in detail, while case studies demonstrate the development process, which begins with computational thinking and ends with effective and efficient parallel programs. This new edition has been updated with an expanded repertoire of optimizations, new patterns and applications, and more coverage of important CUDA features.

NEW FOR THE FIFTH EDITION

- Expanded optimization checklist with a more comprehensive demonstration of essential optimizations across patterns
- New pattern and application chapters including: filtering, wavefront parallelism, advanced optimizations for matrix multiplication, and large language models (LLMs)
- More coverage of important CUDA features including warp-level programming, cooperative groups, CUDA C++ atomics, and multi-GPU programming with NCCL and NVSHMEM

ABOUT THE AUTHORS



Wen-mei W. Hwu is a Senior Director of Research of NVIDIA and the Sanders-AMD Endowed Chair Professor Emeritus of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. His work focuses on parallel computing—covering architecture, implementation, compilers, and algorithms. Dr. Hwu has received numerous honors, including the ACM/IEEE Eckert-Mauchly Award, the ACM Grace Murray Hopper Award, the IEEE B.R. Rau Award and the IEEE Charles Babbage Award. He is an IEEE and ACM Fellow. He earned his Ph.D. in Computer Science from UC Berkeley.



David B. Kirk is known for major contributions to graphics, hardware, and algorithms. Before pursuing his Ph.D. at Caltech, he earned B.S. and M.S. degrees in mechanical engineering from MIT and worked at Raster Technologies and Hewlett-Packard's Apollo Systems Division. After completing his doctorate, he served as chief scientist and head of technology at Crystal Dynamics. In 1997, he became Chief Scientist at NVIDIA. Dr. Kirk has received numerous honors including the IEEE Seymour Cray Computer Engineering Award and ACM SIGGRAPH Computer Graphics Achievement Award. He is a member of the U.S. National Academy of Engineering.



Izzat El Hajj is an Assistant Professor of Computer Science at the American University of Beirut. His research focuses on leveraging accelerator architectures to tackle challenging computations, with a focus on GPU computing, processing-in-memory, and performance modeling. He earned his Ph.D. in Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He has received the Dan Vivoli Endowed Fellowship (UIUC) and the Distinguished Graduate Award from the American University of Beirut.

ELSEVIER
MORGAN KAUFMANN PUBLISHERS
An imprint of Elsevier
shop.elsevier.com

ISBN 978-0-443-43900-1
9 780443 439001