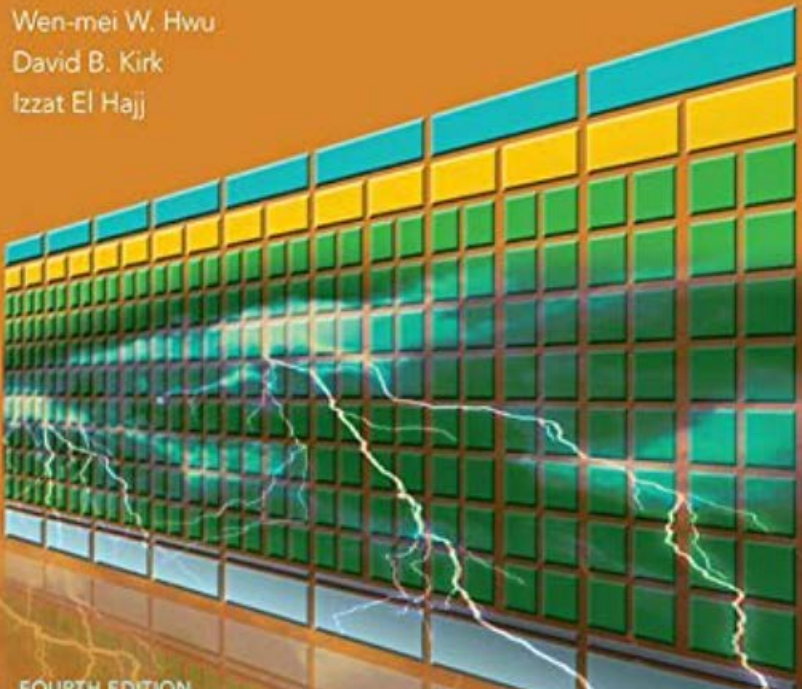
Wen-mei W. Hwu
David B. Kirk
Izzat El Hajj



FOURTH EDITION

Programming Massively Parallel Processors

A Hands-on Approach

MK
MORGAN KAUFMANN

Programming Massively Parallel Processors

A Hands-on Approach

Programming Massively Parallel Processors

A Hands-on Approach

Fourth Edition

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

David B. Kirk
Formerly NVIDIA, United States

Izzat El Hajj
American University of Beirut, Beirut, Lebanon

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

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: 978-0-323-91231-0

Publisher: Katey Birtcher
Acquisitions Editor: Stephen R. Merken
Editorial Project Manager: Naomi Robertson
Production Project Manager: Kiruthika Govindaraju
Cover Designer: Bridget Hoette

Typeset by MPS Limited, Chennai, India

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

Part IV Advanced Practices

Foreword

Written by two exceptional computer scientists and pioneers of GPU computing, Wen-mei and David's *Programming Massively Parallel Processors*, Fourth Edition, by Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj continues to make an invaluable contribution to the creation of a new computing model.

GPU computing has become an essential instrument of modern science. This book will teach you how to use this instrument and give you a superpower tool to solve the most challenging problems. GPU computing will become a time machine that lets you see the future, a spaceship that takes you to new worlds that are now within reach.

Computing performance is needed to solve many of the world's most impactful problems. From the beginning of the history of computers, architects sought parallel computing techniques to boost performance. A hundredfold increase is equivalent to a decade of CPU advancements that relied on sequential processing. Despite the great benefits of parallel computing, creating a new computing model with a virtuous cycle of users, developers, vendors, and distributors has been a daunting chicken-and-egg problem.

After nearly three decades, NVIDIA GPU computing is pervasive, and millions of developers have learned parallel programming, many from earlier editions of this book.

GPU computing is affecting every field of science and industry, even computer science itself. The processing speed of GPUs has enabled deep learning models to learn from data and to perform intelligent tasks, starting a wave of invention from autonomous vehicles and robotics to synthetic biology. The era of AI is underway.

AI is even learning physics and opening the possibility of simulating the Earth's climate a millionfold faster than has ever been possible. NVIDIA is building a GPU supercomputer called Earth-2, a digital twin of the Earth, and partnering with the world's scientific community to predict the impact of today's actions on our climate decades from now.

A life science researcher once said to me, "Because of your GPU, I can do my life's work in my lifetime." So whether you are advancing AI or doing groundbreaking science, I hope that GPU computing will help you do your life's work.

Jensen Huang
NVIDIA, Santa Clara, CA, United States

Preface

We are proud to introduce to you the fourth edition of *Programming Massively Parallel Processors: A Hands-on Approach*.

Mass market computing systems that combine multicore CPUs and many-thread GPUs have brought terascale computing to laptops and exascale computing to clusters. Armed with such computing power, we are at the dawn of the widespread use of computational experiments in the science, engineering, medical, and business disciplines. We are also witnessing the wide adoption of GPU computing in key industry vertical markets, such as finance, e-commerce, oil and gas, and manufacturing. Breakthroughs in these disciplines will be achieved by using computational experiments 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 computational thinking and parallel programming skills will become as pervasive as calculus skills.

The primary target audience of this book consists of graduate and undergraduate students in all science and engineering disciplines in which computational thinking and parallel programming skills are needed to achieve breakthroughs. The book has also been used successfully by industry professional developers who need to refresh their parallel computing skills and keep up to date with 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, bioengineering, physics, chemistry, astronomy, and geography, and they use computation to advance their fields. Thus these developers are both experts in their domains and programmers. The book takes the approach of teaching parallel programming by building up an intuitive understanding of the techniques. We assume that the reader has at least some basic C programming experience. We use CUDA C, a parallel programming environment that is supported on NVIDIA GPUs. There are more than 1 billion of these processors in the hands of consumers and professionals, and more than 400,000 programmers are actively using CUDA. The applications that you will develop as part of your learning experience will be runnable by a very large user community.

Since the third edition came out in 2016, we have received numerous comments from our readers and instructors. Many of them told us about the existing features they value. Others gave us ideas about how we should expand the book's contents to make it even more valuable. Furthermore, the hardware and software for heterogeneous parallel computing have advanced tremendously since 2016. In the hardware arena, three more generations of GPU computing architectures, namely, Volta, Turing, and Ampere, have been introduced since the third edition.

In the software domain, CUDA 9 through CUDA 11 have allowed programmers to access new hardware and system features. New algorithms have also been developed. Accordingly, we added four new chapters and rewrote a substantial number of the existing chapters.

The four newly added chapters include one new foundational chapter, namely, Chapter 4 (Compute Architecture and Scheduling), and three new parallel patterns and applications chapters: Chapter 8 (Stencil), Chapter 10 (Reduction and Minimizing Divergence), and Chapter 13 (Sorting). Our motivation for adding these chapters is as follows:

- Chapter 4 (Compute Architecture and Scheduling): In the previous edition the discussions on architecture and scheduling considerations were scattered across multiple chapters. In this edition, Chapter 4 consolidates these discussions into one focused chapter that serves as a centralized reference for readers who are particularly interested in this topic.
- Chapter 8 (Stencil): In the previous edition the stencil pattern was briefly mentioned in the convolution chapter in light of the similarities between the two patterns. In this edition, Chapter 8 provides a more thorough treatment of the stencil pattern, emphasizing the mathematical background behind the computation and aspects that make it different from convolution, thereby enabling additional optimizations. The chapter also provides an example of handling three-dimensional grids and data.
- Chapter 10 (Reduction and Minimizing Divergence): In the previous edition the reduction pattern was briefly presented in the performance considerations chapter. In this edition, Chapter 10 provides a more complete presentation of the reduction pattern with an incremental approach to applying the optimizations and a more thorough analysis of the associated performance tradeoffs.
- Chapter 13 (Sorting): In the previous edition, merge sort was briefly alluded to in the chapter on the merge pattern. In this edition, Chapter 13 presents radix sort as a noncomparison sort algorithm that is highly amenable to GPU parallelization and follows an incremental approach to optimizing it and analyzing the performance tradeoffs. Merge sort is also discussed in this chapter.

In addition to the newly added chapters, all chapters have been revised, and some chapters have been substantially rewritten. These chapters include the following:

- Chapter 6 (Performance Considerations): Some architecture considerations that were previously in this chapter were moved to the new Chapter 4, and the reduction example was moved to the new Chapter 10. In their place, this chapter was rewritten to provide a more thorough handling of thread granularity considerations and, more notably, to provide a checklist of common performance optimization strategies and the performance bottlenecks

that each strategy tackles. This checklist is referred to throughout the rest of the textbook as we optimize the code for implementing various parallel patterns and applications. The goal is to reinforce a systematic and incremental methodology for optimizing the performance of parallel programs.

- Chapter 7 (Convolution): In the previous edition the chapter on the convolution pattern used a one-dimensional convolution as a running example, with a brief handling of two-dimensional convolutions toward the end. In this edition this chapter was rewritten to focus more on two-dimensional convolution from the start. This change allows us to address the complexity and intricacies of higher-dimensional tiling and equip the readers with a better background for learning convolutional neural networks in Chapter 16.
- Chapter 9 (Parallel Histogram): In the previous edition the chapter on the histogram pattern applied the thread coarsening optimization from the start and combined the privatization optimization with the use of shared memory. In this edition this chapter was rewritten to follow a more incremental approach to performance optimization. The initial implementation that is now presented does not apply thread coarsening. Privatization and the use of shared memory for the private bins are distinguished as two separate optimizations, the former aimed at reducing contention of atomics and the latter aimed at reducing access latency. Thread coarsening is applied after privatization, since one major benefit of coarsening is to reduce the number of private copies committed to the public copy. The new organization of the chapter is more consistent with the systematic and incremental approach to performance optimization that is followed throughout the book. We also moved the chapter to precede the chapters on the reduction and scan patterns in order to introduce atomic operations sooner, since they are used in multiblock reduction and single-pass scan kernels.
- Chapter 14 (Sparse Matrix Computation): In this edition this chapter was rewritten to follow a more systematic approach for analyzing the tradeoffs between different sparse matrix storage formats. The beginning of the chapter introduces a list of considerations that go into the design of different sparse matrix storage formats. This list of design considerations is then used throughout the chapter to systematically analyze the tradeoffs between the different formats.
- Chapter 15 (Graph Traversal): In the previous edition the chapter on graph traversal focused on a particular BFS parallelization strategy. In this edition this chapter was significantly expanded to cover a more comprehensive set of alternative parallelization strategies and to analyze the tradeoffs between them. These strategies include vertex-centric push-based, vertex-centric pull-based, edge-centric, and linear algebraic implementations in addition to the original implementation, which was the vertex-centric push-based frontier-based implementation. The classification of these alternatives is not unique to BFS but applies to parallelizing graph algorithms in general.

- Chapter 16 (Deep Learning): In this edition this chapter was rewritten to provide a comprehensive yet intuitive theoretical background for understanding modern neural networks. The background makes it easier for the reader to fully understand the computational components of neural networks, such as fully connected layers, activation, and convolutional layers. It also removes some of the common barriers to understanding the kernel functions for training a convolutional neural network.
- Chapter 19 (Parallel Programming and Computational Thinking): In the previous edition this chapter discussed algorithm selection and problem decomposition while drawing examples from the chapters on iterative MRI reconstruction and electrostatic potential map. In this edition the chapter was revised to draw examples from many more chapters, serving as a concluding chapter for Parts I and II. The discussion of problem decomposition was particularly expanded to introduce the generalizations of output-centric decomposition and input-centric decomposition and to discuss the tradeoffs between them, using many examples.
- Chapter 21 (CUDA Dynamic Parallelism): In the previous edition this chapter went into many programming details relating to the semantics of different programming constructs and API calls in the context of dynamic parallelism. In this edition the focus of the chapter has shifted more toward the application examples, with the other programming details discussed more briefly while referring interested readers to the CUDA programming guide.

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 have striven to keep all our explanations intuitive and practical. Second, we keep the book as concise as possible. Although it is tempting to keep adding new material, we wanted to minimize the number of pages a reader needs to go through to learn all the key concepts. We accomplished this by moving the previous chapter on numerical considerations to the appendix. While numerical considerations are an extremely important aspect of parallel computing, we found that a substantial amount of the content in the chapter was already familiar to many of our readers who come from a computer science or computational science background. For this reason we preferred to dedicate more space to covering additional parallel patterns.

In addition to adding new chapters and substantially rewriting others since the previous edition, we have also organized the book into four major parts. This 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. The second part applies these concepts by covering six common computation patterns and showing how they can be parallelized and optimized. Each parallel pattern also introduces a new programming feature or technique. The third part introduces additional advanced patterns and applications

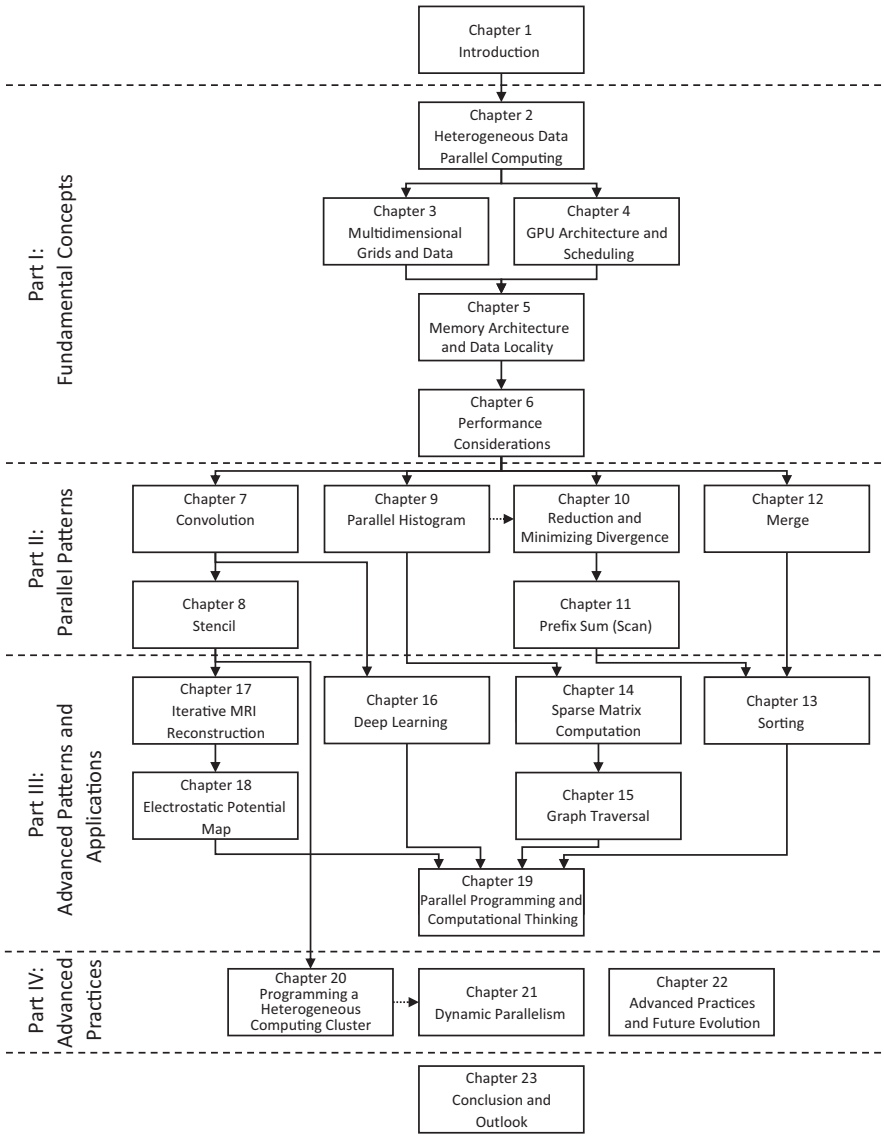


FIGURE P.1

Organization of the book.

and continues to apply the optimizations that are practiced in the second part. However, it puts more emphasis on exploring alternative forms of problem decomposition to parallelize a computation and analyzes the tradeoffs between different decompositions and their associated data structures. Finally, the fourth part exposes the reader to advanced practices and programming features.

How to use the book

We would like to offer some of our experience in teaching courses with this book. Since 2006 we have taught multiple types of courses: in one-semester format and in one-week intensive format. The original ECE498AL course has become a permanent course known as ECE408 or CS483 at the University of Illinois at Urbana-Champaign. We started to write up some of the early chapters of this book when we offered ECE498AL the second time. The first four chapters were also tested in an MIT class taught by Nicolas Pinto in the spring of 2009. 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 covered in approximately a single 75-minute lecture each. The chapters that may need two 75-minute lectures to be fully covered are Chapter 11 (Prefix Sum (Scan)), Chapter 14 (Sparse Matrix Computation), and Chapter 15 (Graph Traversal). 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 I and II of this book, students learn about fundamentals and basic patterns, and they practice the skills that they learn via guided programming assignments. This phase consists of 12 chapters and typically takes around seven weeks. Each week, students work on a programming assignment corresponding to that week's lectures. 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 2 weeks include a series of four lectures based on Chapters 3 through 6 that give students the 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 in which they see how the performance of their implementations increases dramatically throughout this period. In the remaining four weeks, the lectures cover common data-parallel programming patterns that are needed to develop a high-performance parallel application based on Chapters 7 through 12. Throughout these weeks, students complete assignments on convolution, histogram, reduction, and prefix sum. 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 consists of Parts III and IV, students learn about advanced patterns and applications while they work on a final project that involves accelerating an advanced pattern or application. They also learn about advanced practices that they may find useful when finalizing their projects. Although we do not usually assign weekly programming assignments during this phase, the project typically has a weekly milestone to help the students pace themselves. Depending on the duration and format of the course, instructors may not be able to cover all the chapters in this phase and may need to skip some. Instructors might 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 dependences between chapters to assist instructors in selecting what chapters they can skip or reorder to customize the course for their particular context.

Tying it all together: the final project

While the lectures, labs, and chapters of this book help to 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' worth of 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 Illinois-NVIDIA GPU Teaching Kit, we would like to offer the reasoning 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 types of math, physics, and chemistry courses) that are required to understand and work on the application, and a list of web and traditional resources on which students can draw 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 Illinois-NVIDIA GPU Teaching Kit.

The design document

Once the students have decided on a project and formed a team, they are required to 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 later career success. The design document should discuss the background and motivation for the project, the application-level objectives and potential impact, the 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 that are directed to the student individually, so 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 could be accommodated by the final project process. As a result, we moved from the final project to a class competition. At the midpoint 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 auto-graded 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 Illinois-NVIDIA GPU Teaching Kit is a publicly available resource that contains lecture slides and recordings, lab assignments, final project guidelines, and

sample project specifications for instructors who use this book for their classes. In addition, we are in the process of making the courses of the Illinois undergraduate-level and graduate-level offerings based on this book publicly available. While this book provides the intellectual contents for these classes, the additional material will be crucial in achieving the overall education goals.

Finally, we encourage you to submit your feedback. We would like to hear from you if you have any ideas for improving this book. We would like to know how we can improve the supplementary online material. Of course, we also like to know what you liked about the book. We look forward to hearing from you.

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

Acknowledgments

There are so many people who have made special contributions to this fourth edition. We would like first to thank the contributing chapter coauthors. 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 acknowledge Ian Buck, the father of CUDA, and John Nickolls, the lead architect of Tesla GPU Computing Architecture. Their teams built excellent infrastructure for this course. Many engineers and researchers at NVIDIA have also contributed to the rapid advancement of CUDA, which 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 have spent numerous hours of their precious time to give us insightful feedback since the third edition: Sonia Lopez Alarcon (Rochester Institute of Technology), Bedrich Benes (Purdue University), Bryan Chin (UCSD), 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), Dejan Milojici (Hewlett-Packard Labs), Pinar Muyan-Ozcelik (California State University, Sacramento), Greg Peterson (University of Tennessee−Knoxville), José L. Sánchez (University of Castilla−La Mancha), Janche Sang (Cleveland State University), and Jan Verschelde (University of Illinois at Chicago). Their comments helped us to significantly improve the content and readability of the book.

Steve Merken, Kiruthika Govindaraju, Naomi Robertson, 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, who challenged us to embark on the project. Beth Katsinas arranged a meeting between Dick Blahut and NVIDIA Vice President Dan Vivoli. Through that gathering, Blahut was introduced to David and 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, Mike Flynn, Michael Garland, John Hennessy, Pat Hanrahan, Nick Holonyak, Dick Karp, Kurt Keutzer, Chris Lamb, Dave Liu, David Luebke, Dave Kuck, Nacho

Navarro, Sanjay Patel, Yale Patt, David Patterson, Bob Rao, Burton Smith, Jim Smith, 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.

CHAPTER

Introduction 1

Chapter Outline

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 rely on the advancement of processor speed, memory speed, and memory capacity to enhance application-level capabilities such as the timeliness of weather forecasts, the accuracy of engineering structural analyses, the realism of computer-generated graphics, the number of airline reservations processed per second, and 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 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 $\times 86$ 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 the two decades of growth, these single-CPU microprocessors brought GFLOPS, or giga (10^9) floating-point operations per second, to the desktop and TFLOPS, or tera (10^{12}) floating-point operations per second, to data centers. 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. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00006-9

However, this drive has slowed down since 2003, owing to energy consumption and heat dissipation issues. These issues limit the increase of the clock frequency and the productive activities that can 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 have switched to a model in which multiple physical CPUs, referred to as processor cores, are used in each chip to increase the processing power. A traditional CPU can be viewed as a single-core CPU in this model. 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 has exerted a tremendous impact on the software developer community.

Traditionally, the vast majority of software applications are written as sequential programs that are executed by processors whose design was envisioned by von Neumann in his seminal report in 1945 (von Neumann et al., 1972). The execution of these programs can be understood by a human as sequentially stepping through the 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 *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 runs faster as each new processor generation is introduced. Computer users also grew to expect that these programs run faster with each new generation of microprocessors. This expectation has not been valid for over a decade. A sequential program will run on only one of the processor cores, which will not become significantly faster from generation to generation. Without performance improvement, application developers will no longer be able to introduce new features and capabilities into their software as new microprocessors are introduced; this reduces the growth opportunities of the entire computer industry.

Rather, the application software that will continue to enjoy significant performance improvement with each new generation of microprocessors will be *parallel programs*, in which multiple threads of execution cooperate to complete the work faster. This new, dramatically escalated advantage of parallel programs over sequential programs has been referred to as the concurrency revolution (Sutter and Larus, 2005). The practice of parallel programming is by no means new. The high-performance computing (HPC) community has been developing parallel

programs for decades. These parallel programs typically ran on expensive large-scale computers. Only a few elite applications could justify the use of these computers, thus limiting the practice of parallel programming to a small number of application developers. Now that all new microprocessors are parallel computers, the number of applications that need to be developed as parallel programs has increased dramatically. There is now a great 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 (Hwu et al., 2008). The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began with two-core processors, and the number of cores has increased with each semiconductor process generation. A recent example is a recent Intel multicore server microprocessor with up to 24 processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full $\times 86$ instruction set, supporting hyperthreading with two hardware threads, designed to maximize the execution speed of sequential programs. Another example is a recent ARM Ampere multicore server processor with 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 increases with each generation. A recent exemplar is the NVIDIA Tesla A100 graphics processing unit (GPU) with tens 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 2021, the peak floating-point throughput of the A100 GPU is 9.7 TFLOPS for 64-bit double-precision, 156 TFLOPS for 32-bit single-precision, and 312 TFLOPS for 16-bit half-precision. In comparison, the peak floating-point throughput of the recent Intel 24-core processor is 0.33 TLOPS for double-precision and 0.66 TFLOPS for single-precision. The ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs has been increasing for the past several years. These are not necessarily application speeds; they are merely the raw speeds that the execution resources can potentially support in these chips.

Such a large gap in peak performance between multicores and many-threads has amounted to a significant "electrical potential" buildup, 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 important, 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 is more work to do, there is more opportunity to divide the work among cooperating parallel workers, that is, threads.

One might ask why there is such a large peak performance gap between many-threaded GPUs and multicore 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. 1.1A, 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 control logic consume chip area and power that could otherwise be 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, which 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.

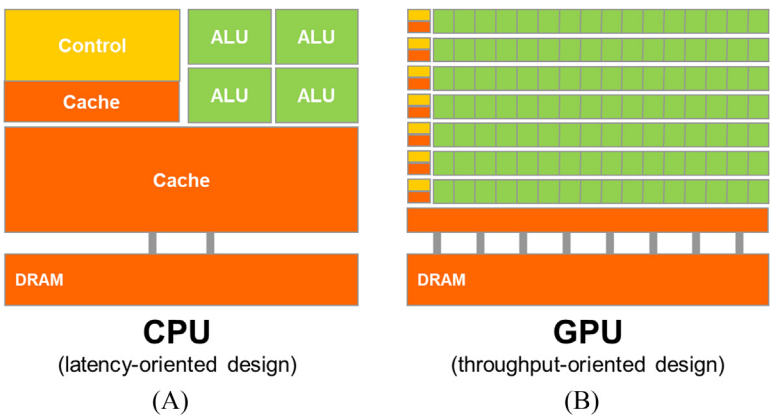


FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies: (A) CPU design is latency oriented; (B) GPU design is throughput-oriented.

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 important 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 into 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 in which various system software, applications, and I/O devices expect their memory accesses to work) that is 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 10 times 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. 1.1A, in contrast to the larger number of smaller arithmetic units and a larger number of memory channels in Fig. 1.1B.

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. 1.1B 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 the sequential parts on the CPU and the numerically intensive parts on the GPUs. This is why the Compute Unified Device Architecture (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 marketplace, 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 that are funded by the government and large corporations have been successfully developed on these traditional parallel computing systems. This has changed with many-thread GPUs. Because of 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 Magnetic Resonance Imaging (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 that require racks of computer server boxes in clinical settings, while this is common in academic departmental settings. In fact, the National Institutes of Health (NIH) refused to fund parallel programming projects for some time; they believed 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 the 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 GPU. 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.

Everything changed in 2007 with the release of CUDA (NVIDIA, 2007). CUDA did not represent software changes alone; 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. 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 that are used as accelerators in heterogeneous computing systems. For example, field-programmable gate arrays 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 these 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 increases in future hardware generations. As we will discuss in the chapters on parallel patterns, advanced patterns, and applications (Parts II and III, Chapters 7 through 19), when an application is suitable for parallel execution, a good implementation on a GPU can achieve a speed up of more than 100 times over sequential execution on a single CPU core. If the application includes what we call “data parallelism,” it is 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 quite fast enough. Despite the myriad of computing applications in today’s world, many exciting mass

market applications of the future are what we previously considered supercomputing applications, or superapplications. 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. However, 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 increasing computing speeds 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 for science and medicine.

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition (HD) TV in comparison to older NTSC TV. Once we experience the level of details in the picture on 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 three-dimensional (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 will begin to see 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 that are offered by more computing speed are much better user interfaces. Smartphone users now 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 3D 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 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, even if you lost a wheel. With increased computing speed, the games can be based on dynamic simulation rather than prearranged 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*, in which 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 that have been 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 have been ineffective in practical applications because it takes too much labeled data and too much computation to train these networks. The rise of the 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 has 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 a 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 system B over the time used to execute the same application in system 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 200/10=20, which is referred to as a 20× (20 times) speedup.

The speedup that is 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 100× 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 infinite amount of speedup in the parallel portion can only slash 30% off the execution time, achieving no more than 1.43× speedup. The fact that the level of speedup that one can achieve through parallel execution can be severely limited by the parallelizable portion of the application is referred to as Amdahl's Law (Amdahl, 2013). On the other hand, if 99% of the execution time is in the parallel portion, a 100× speedup of the parallel portion will reduce the application execution to 1.99% of the original time. This gives the entire application a 50× 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 speed up its execution.

Researchers have achieved speedups of more than 100× for some applications. However, this is typically achieved only after extensive optimization and tuning after 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. However, one must 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 in using them.

Keep in mind that the level of speedup that is 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 the 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 multicore CPUs and many-core GPUs have brought terascale computing to laptops 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—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 although these portions can take up a large portion of the code, they tend to account for only a small portion of the execution time of superapplications.

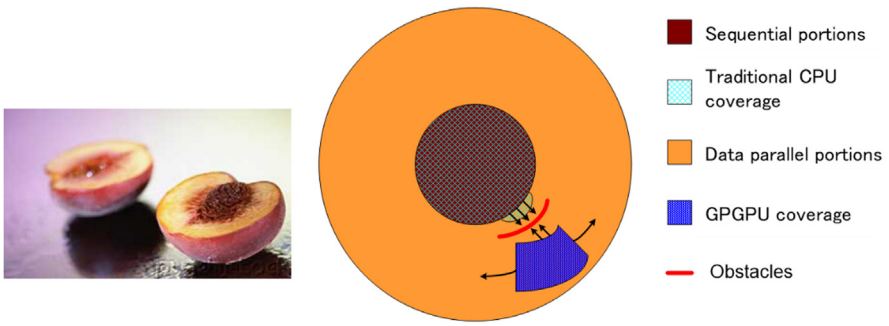


FIGURE 1.2

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.

Then come what we call the "peach flesh" 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 GPGPU programming interfaces cover only a small portion of the peach flesh 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 flesh of exciting applications. Parallel programming models and their underlying hardware are still evolving at a fast pace 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 that of sequential algorithms. Many parallel algorithms perform the same amount of work as their sequential counterparts. However, some parallel algorithms do more work than their sequential counterparts. In fact, sometimes they may do so much more work that they ended 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 requires nonintuitive ways of thinking about the problem and may require redundant work during execution. There are important algorithm primitives, such as prefix sum, that can facilitate the conversion of sequential, recursive formulation of the problems into more parallel forms. We will more formally introduce the concept of *work efficiency* and will illustrate the methods and tradeoffs that are involved in designing parallel algorithms that achieve the same level of computational complexity as their sequential counterparts, using important parallel patterns such as prefix sum in Chapter 11, Prefix Sum (Scan).

Second, the execution speed of many applications is limited by memory access latency and/or throughput. We refer to these applications as memory bound; by contrast, compute bound applications are limited by the number of instructions performed per byte of data. 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, Memory Architecture and Data Locality and Chapter 6, Performance Considerations, and will 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 is the case for their sequential counterparts. 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 in sizes and distributions can cause uneven amount of work to be assigned to the parallel threads and can 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 and/or dynamically refining the number of 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*. 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 will often find themselves waiting for other threads instead of performing useful work. We will discuss various strategies for reducing this synchronization overhead throughout this book.

Fortunately, most of these challenges have been addressed by researchers. There are also common patterns across application domains that allow us to apply solutions that were derived in one domain to challenges in other domains. 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 proposed in the past several decades (Mattson et al., 2004). The ones that are the most widely used are OpenMP (Open, 2005) for shared memory multiprocessor systems and Message Passing Interface (MPI) (MPI, 2009) 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 to 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 programmer to understand all the detailed parallel programming concepts that are 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 in which OpenMP compilers fall short.

On the other hand, MPI is a programming interface in which computing nodes in a cluster do not share memory (MPI, 2009). All data sharing and interaction must be done through explicit message passing. MPI has been widely used in 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 that is needed to port an application into MPI can be quite high, owing 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. On the basis of the domain decomposition, the programmer also needs to call message sending and receiving functions to manage the data exchange between nodes. CUDA, by contrast, provides shared memory for parallel execution in the GPU to address this difficulty. While CUDA is an effective interface with each node, most application developers need to use MPI to program at the cluster level. Furthermore, there has been increasing support for multi-GPU programming in CUDA via APIs such as the NVIDIA Collective Communications Library (NCCL). It is therefore important that a

parallel programmer in HPC understands how to do joint MPI/CUDA programming in modern computing clusters employing multi-GPU nodes, a topic that is presented in Chapter 20, Programming a Heterogeneous Computing Cluster.

In 2009, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, jointly developed a standardized programming model called Open Compute Language (OpenCL) (The Khronos Group, 2009). 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 that are 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 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 important, virtually all techniques that are learned in 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. Therefore 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 (Wing, 2006) 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 that will enable programmers to 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, Compute Architecture and Scheduling, to introduce the fundamentals

of the GPU architecture. We also discuss more specialized architecture concepts as part of our discussions of 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. Programmers 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, memory consistency, and atomicity 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. We will show that by focusing on data parallelism, one can achieve both high performance and high reliability in one's applications.

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 (Mattson et al., 2004) 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. 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 four parts. Part I covers fundamental concepts in parallel programming, data parallelism, GPUs, and performance optimization. These foundational chapters equip the reader with the basic knowledge and skills that are necessary for becoming a GPU programmer. Part II covers primitive parallel patterns, and Part III covers more advanced parallel patterns and applications. These two parts apply the knowledge and skills that were learned in the first part and introduce other GPU architecture features and optimization techniques as the need for them arises. The final part, Part IV, introduces advanced practices to complete the knowledge of readers who would like to become expert GPU programmers.

Part I on fundamental concepts consists of Chapters 2–6. Chapter 2, Heterogeneous Data Parallel Computing, introduces data parallelism and CUDA C programming. The chapter relies on the fact that the reader has had previous experience with C programming. It first introduces CUDA C as a simple, small extension to C that supports heterogeneous CPU/GPU computing and the widely used single-program, multiple-data parallel programming model. It then covers the thought processes that are 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 is to teach enough concepts of the CUDA C programming model so that the reader can write a simple parallel CUDA C program, it covers several basic skills that are needed to develop a parallel application based on any parallel programming interface.

Chapter 3, Multidimensional Grids and Data, presents more details of the parallel execution model of CUDA, particularly as it relates to handling multidimensional data using multidimensional 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 computation using CUDA C.

Chapter 4, Compute Architecture and Scheduling, introduces the GPU 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, multithreading and latency tolerance, and occupancy, all of which are defined and discussed in the chapter.

Chapter 5, Memory Architecture and Data Locality, extends Chapter 4, Compute Architecture and Scheduling, 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, Performance Considerations, presents several important performance considerations in current CUDA hardware. In particular, it gives more details about desirable patterns of thread execution and memory accesses. These details form the conceptual basis for programmers to reason about the consequences 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 II on primitive parallel patterns consists of Chapters 7−12. Chapter 7, Convolution, 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, Stencil, presents stencil, a pattern that is similar to convolution but 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 3D organizations of threads and data and to showcase an optimization introduced in Chapter 6, Performance Considerations, that targets thread granularity.

Chapter 9, Parallel Histogram, covers histogram, a pattern that is widely used in statistical data analysis as well as pattern recognition in large datasets. 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, Reduction and Minimizing Divergence, 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 on performance and show techniques for how this impact can be mitigated. Chapter 11, Prefix Sum (Scan), 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. Finally, Chapter 12, Merge, 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.

Part III on advanced parallel patterns and applications is similar in spirit to Part II, but the patterns that are covered are more elaborate and often include more application context. Thus 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 that are needed to achieve high performance. These chapters help the readers to put all the materials from the previous chapters together and support them as they take on their own application development projects.

Part III consists of Chapters 13−19. Chapter 13, Sorting, presents two forms of parallel sorting: radix sort and merge sort. This advanced pattern leverages more primitive patterns that were covered in previous chapters, particularly prefix sum and parallel merge. Chapter 14, Sparse Matrix Computation, presents sparse matrix computation, which is widely used for processing very large datasets.

The chapter introduces the reader to the concepts of rearranging data for more efficient parallel access: data compression, padding, sorting, transposition, and regularization. Chapter 15, Graph Traversal, 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 discussed. These strategies build on the more primitive patterns, such as histogram and merge.

Chapter 16, Deep Learning, covers deep learning, which is becoming an extremely important area for GPU computing. We introduce the efficient implementation of convolutional neural networks and leave more in-depth discussion to other sources. The efficient implementation of the convolution neural networks leverages techniques such as tiling and patterns such as convolution. Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction, covers non-Cartesian MRI reconstruction and how to leverage techniques such as loop fusion and scatter-to-gather transformations to enhance parallelism and reduce synchronization overhead. Chapter 18, Electrostatic Potential Map, covers molecular visualization and analysis, which benefit from techniques to handle irregular data by applying lessons learned from sparse matrix computation.

Chapter 19, Parallel Programming and Computational Thinking, introduces computational thinking, the art of formulating and solving computational problems in ways that are more amenable to HPC. 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 high-quality application software, serial or parallel. The chapter 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 that they gain in this book. We also present a high-level case study to show the opportunities that can be seen through creative computational thinking.

Part IV on advanced practices consists of Chapters 20−22. Chapter 20, Programming a Heterogeneous Computing Cluster, covers CUDA programming on heterogeneous clusters, in which each compute node consists of both CPUs and GPUs. We discuss the use of MPI alongside CUDA to integrate both internode computing and intranode computing and the resulting communication issues and practices. Chapter 21, CUDA Dynamic Parallelism, covers dynamic parallelism, which is the ability of the GPU to dynamically create work for itself based on the data or program structure rather than always waiting for the CPU to do so. Chapter 22, Advanced Practices and Future Evolution, goes through a list of miscellaneous advanced features and practices that are important for CUDA programmers to be aware of. These include topics such as zero-copy memory, unified virtual memory, simultaneous execution of multiple kernels, function calls,

exception handling, debugging, profiling, double-precision support, configurable cache/scratchpad sizes, and others. For example, early versions of CUDA provided limited shared memory capability between the CPU and the GPU. The programmers needed to explicitly manage the data transfer between CPU and GPU. However, current versions of CUDA support features such as unified virtual memory and zero-copy memory that enable seamless sharing of data between CPUs and GPUs. With such support, a CUDA programmer can declare variables and data structures as shared between CPU and GPU. The runtime hardware and software maintain coherence and automatically perform optimized data transfer operations on behalf of the programmer on a need basis. Such support significantly reduces the programming complexity that is involved in overlapping data transfer with computation and I/O activities. In the introductory part of the textbook, we use the APIs for explicit data transfer so that reader gets a better understanding of what happens under the hood. We later introduce unified virtual memory and zero-copy memory in Chapter 22, Advanced Practices and Future Evolution.

Although the chapters throughout this book are based on CUDA, they help the readers to 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. In-depth experience with CUDA also enables us to gain maturity, which will help us to learn concepts that may not even be pertinent to the CUDA model.

Chapter 23, Conclusion and Outlook, 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

Amdahl, G.M., 2013. Computer architecture and amdahl's law. Computer 46 (12), 38–46.

Hwu, W.W., Keutzer, K., Mattson, T., 2008. The concurrency challenge. IEEE Design and Test of Computers 312–320.

Mattson, T.G., Sanders, B.A., Massingill, B.L., 2004. Patterns of Parallel Programming, Addison-Wesley Professional.

Message Passing Interface Forum, 2009. MPI – A Message Passing Interface Standard Version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, September 4.

NVIDIA Corporation, 2007. CUDA Programming Guide, February.

OpenMP Architecture Review Board, 2005. OpenMP application program interface.

Sutter, H., Larus, J., 2005. Software and the concurrency revolution, in. ACM Queue 3 (7), 54–62.

The Khronos Group, 2009. The OpenCL Specification version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

von Neumann, J., 1972. First draft of a report on the EDVAC. In: Goldstine, H.H. (Ed.), The Computer: From Pascal to von Neumann. Princeton University Press, Princeton, NJ, ISBN 0−691-02367-0.

Wing, J., 2006. Computational thinking. Communications of the ACM 49 (3).

CHAPTER

Heterogeneous data parallel computing

2

With special contribution from David Luebke

Chapter Outline

Data parallelism refers to the phenomenon in which the computation work to be performed on different parts of the 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. 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

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00014-8

images or videos with millions to trillions of pixels. Scientific applications model fluid dynamics using billions of grid points. Molecular dynamics applications must simulate interactions between thousands to billions of atoms. Airline scheduling deals 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 we 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 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:

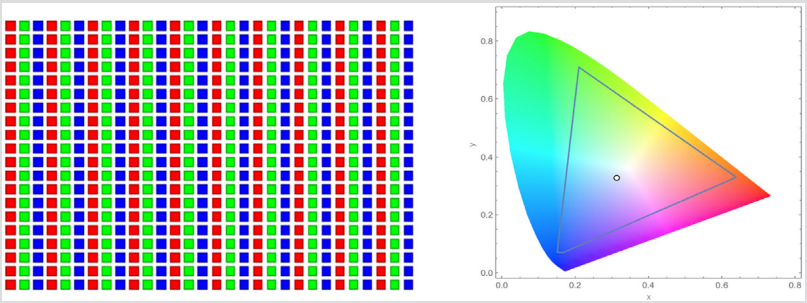
$$L = r*0.21 + g*0.72 + b*0.07$$

FIGURE 2.1

Conversion of a color image to a grayscale image.

RGB Color Image Representation

In an RGB representation, each pixel in an image is stored 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.



The actual allowable mixtures of these three colors vary across industry-specified color spaces. Here, the valid combinations of the three colors in the AdbobeRGB™ color space are shown as the interior of the triangle. The vertical coordinate (y value) and horizontal coordinate (x value) of each mixture show the fraction of the pixel intensity that should be G and R. The remaining fraction (1-y−x) of the pixel intensity should be assigned to B. To render an image, the r, g, b values of each pixel are used to calculate both the total intensity (luminance) of the pixel as well as the mixture coefficients (x, y, 1-y-x).

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 computation structure shown in Fig. 2.2. For example, *O*[0] is generated by calculating the weighted sum of the RGB values in *I*[0] according to the formula above; *O*[1] is generated by calculating the weighted sum of the RGB values in *I*[1]; *O*[2] is generated by calculating the weighted sum of the RGB values in *I*[2]; and so on. None of these per-pixel computations depend on each other. All of them can be performed independently. Clearly, color-to-grayscale conversion exhibits a rich amount of data parallelism. 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.

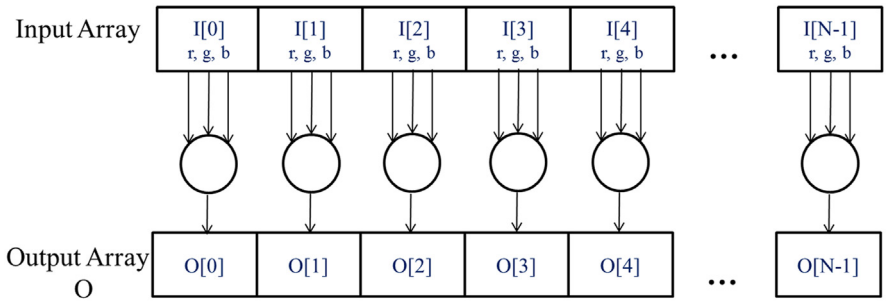


FIGURE 2.2

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

Task Parallelism vs. Data 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 are usually a larger number of independent tasks and therefore larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks includes vibrational forces, rotational forces, neighbor identification for non-bonding forces, non-bonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large datasets, 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. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce streams.

2.2 CUDA C program structure

We are now ready to learn how to write a CUDA C program to exploit data parallelism for faster execution. CUDA C[1] extends the popular ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel 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 coexistence of a *host* (CPU) and one or more *devices* (GPUs) in the computer. Each CUDA C source file can have a mixture of host code and device code. By default, any traditional C program is a CUDA 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 program is illustrated in Fig. 2.3. The execution starts with host code (CPU serial code). When a kernel function 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. We will discuss how these grids are organized soon. When all threads of a grid have completed their execution, the grid terminates, and the execution continues on the host until another grid is launched.

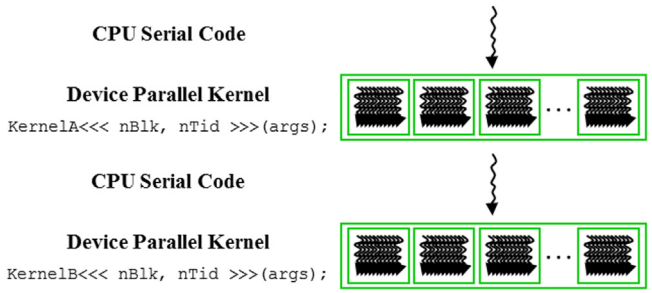


FIGURE 2.3

Execution of a CUDA program.

[1] There has been a steady movement for CUDA C to adopt C++ features. We will be using some of these C++ features in our programming examples.

Note that Fig. 2.3 shows a simplified model in which the CPU execution and the GPU execution do not overlap. Many heterogeneous computing applications manage overlapped CPU and GPU execution to take advantage of both CPUs and GPUs.

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 O. 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 will be generated. CUDA programmers can assume that these threads take very few clock cycles to generate and schedule, owing to efficient hardware support. This assumption contrasts with traditional CPU threads, which typically take thousands of clock cycles to generate and schedule. In the next chapter we will show how to implement color-to-grayscale conversion and image blur kernels. In the rest of this chapter we will use vector addition as a running example for simplicity.

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 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, he/she creates and manages multiple threads using thread libraries or special languages. In CUDA, the execution of each 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 threads that process different parts of the data in parallel.

2.3 A vector addition kernel

We use vector addition to demonstrate the CUDA C program structure. 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 traditional

```
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.

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 have only host code in Fig. 2.4, we see only variables suffixed with "_h".

Pointers in the C Language

The function arguments A, B, and C in Fig. 2.4 are pointers. In the C language, 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 program can be accessed through a pointer that points to its 0^{th} element. For example, the statement P = &(A[0]) makes P point to the 0^{th} 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 0^{th} element.

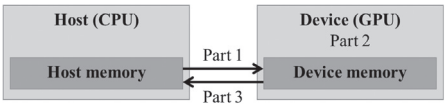
In Fig. 2.4, passing an array name A as the first argument to function call to vecAdd makes the function's first parameter A_h point to the 0^{th} element of A. As a result, A_h[i] in the function body can be used to access A[i] for the array A in the main function.

See Patt & Patel (Patt & Patel, 2020) for an easy-to-follow explanation of the detailed usage of pointers in C.

Assume that the vectors to be added are stored in arrays A and B that are allocated and initialized in the main program. The output vector is in array C, which is also allocated in the main program. 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. This naming convention will be helpful when we introduce device code in the next few steps.

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




```
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 function that moves the work to a device.

calls the actual 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 deallocates 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 a "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 on them 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 called device *global* memory, or simply global memory. For example, the NVIDIA Volta V100 comes with 16GB or 32GB 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 Chapter 5, Memory Architecture and Data Locality.

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. This corresponds to Part 1 of Fig. 2.5. Similarly, after device execution the programmer needs to transfer result 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. This corresponds to Part 3 of Fig. 2.5. The CUDA runtime system (typically running on the host) provides applications programming interface (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

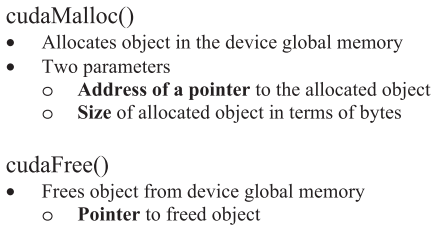


FIGURE 2.6

CUDA API functions for managing device global memory.

of device global memory for an object. The reader should notice the striking similarity between `cudaMalloc` and the standard `C` runtime library malloc function. This is intentional; CUDA C is C with minimal extensions. CUDA C uses the standard C runtime library malloc function to manage the host memory[2] and adds `cudaMalloc` as an extension to the C runtime library. By keeping the interface as close to the original C runtime libraries as possible, CUDA C minimizes the time that a C programmer spends relearning 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.[3] This parameter allows the `cudaMalloc` function to write the address of the allocated memory into the provided pointer variable regardless of its type.[4] 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.

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

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

[2] CUDA C also has more advanced library functions for allocating space in the host memory. We will discuss them in Chapter 20, Programming a Heterogeneous Computing Cluster.

[3] The fact that `cudaMalloc` returns a generic object makes the use of dynamically allocated multidimensional arrays more complex. We will address this issue in Section 3.2.

[4] Note that `cudaMalloc` has a 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.

This 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`) casted 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, in 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` 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 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 runtime errors.

The reader should complete Part 1 of the `vecAdd` example in Fig. 2.5 with similar declarations of `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. This 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 types of 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 in the device global memory.

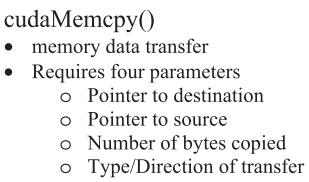


FIGURE 2.7

CUDA API function for data transfer between host and device.

The vecAdd function calls the cudaMemcpy function to copy the A_h and B_h vectors from the host memory to A_d and B_d in the device memory before adding them and to copy the C_d vector from the device memory to C_h in the host memory after the addition has been done. Assuming 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. The two symbolic constants, cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost, are recognized, predefined constants of the CUDA programming environment. Note that the same 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.

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

To summarize, the main program 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.

```
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 vecAdd().

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 the device to the 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 (Fig. 2.9).

Error Checking and Handling in CUDA

In general, it is important for a program to check and handle errors. 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. 2.9 shows a call to cudaMalloc:

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

In practice, we should surround the call with code that test for error condition and print out error messages 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. This can save many hours of debugging time.

One could define a C macro to make the checking code more concise in the source.

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 C, a kernel function specifies

the code to be executed by all 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) (Atallah, 1998) parallel programming style, a popular programming style for parallel computing systems.[5]

When a program's host code calls a kernel, the CUDA runtime system launches a grid of threads that are organized into a two-level hierarchy. Each grid is organized as an array of *thread blocks*, which we will refer to as blocks for brevity. All blocks of a grid are of the same size; each block can contain up to 1024 threads on current systems.[6] Fig. 2.9 shows an example in which 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.

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.

The total number of threads in each thread block is specified by the host code when a kernel is called. The same kernel can be called with different numbers of threads at 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 is a struct with three unsigned integer fields (x, y, and z) that help the programmer to 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 because 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. In Fig. 2.9, each thread block is organized as a one-dimensional 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 number of threads in each dimension of a thread block be a multiple of 32 for hardware efficiency reasons. We will revisit this later.

[5] Note that SPMD is not the same as SIMD (single instruction multiple data) [Flynn 1972]. In an SPMD system the parallel processing units execute the same program on multiple parts of the data. However, these processing units do not need to be executing the same instruction at the same time. In an SIMD system, all processing units are executing the same instruction at any instant.

[6] Each thread block can have up to 1024 threads in CUDA 3.0 and beyond. Some earlier CUDA versions allow only up to 512 threads in a block.

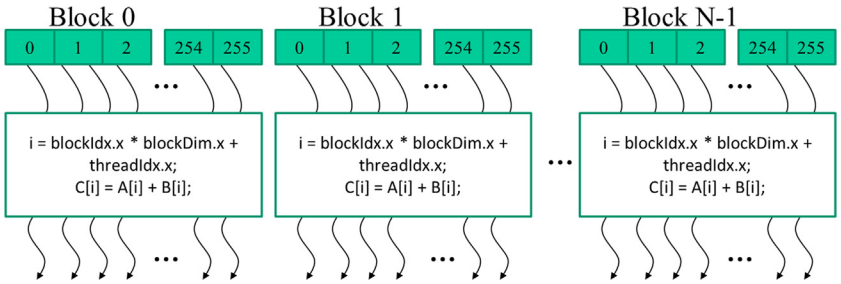


FIGURE 2.9

All threads in a grid execute the same kernel code.

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, and so on.

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 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.

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. The two together gives each telephone line in the whole country a unique phone number. 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 as i=blockIdx.x * blockDim.x + threadIdx.x. Recall that blockDim 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*.

Fig. 2.10 shows a kernel function for vector addition. Note that we do not use the "_h" and "_d" convention in kernels, since there is no potential confusion. We will not have any access to the host memory in our examples. The syntax of a kernel is ANSI C with some notable extensions. First, there is a CUDA-C-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*, it can also be called from the device, as we will see in Chapter 21,

```
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 vector addition kernel function.

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.

CUDA Dynamic Parallelism. The important feature is that calling such a kernel function results in a new grid of threads being launched on the device.

The "`__device__`" keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can be called only 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.[7]

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 the host and can be called only 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 of changing all original function declarations.

Note that one can use both "`__host__`" and "`__device__`" 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 be called only from a host function. The other is executed on the device and can be called only from a device or kernel function. This 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 extension to C, 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 toward a particular part of the data. These built-in variables are the means for threads to access hardware registers that provide the

[7] We will explain the rules for using indirect function calls and recursions in different generations of CUDA later. In general, one should avoid the use of recursion and indirect function calls in their device functions and kernel functions to allow maximal portability.

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.

There is an automatic (local) variable *i* in Fig. 2.10. 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, Memory Architecture and Data Locality.

A quick comparison between Fig. 2.4 and Fig. 2.10 reveals an important insight into CUDA kernels. The kernel function in Fig. 2.10 does not have a loop that corresponds to the one in Fig. 2.4. 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*, in which iterations of the original sequential code are executed by threads in parallel.

Note that there is an if ($i < n$) statement in `addVecKernel` 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 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. This is illustrated in Fig. 2.12. When the host code calls a kernel, it sets the grid and thread block dimensions via *execution*

```
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 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 complete version of the host code in the vecAdd function.

configuration parameters. The configuration parameters are given between the "<<<" and ">>>" 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 the 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 $4 \times 256 = 1024$ 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 in Fig. 2.5. Figs. 2.12 and 2.13 jointly illustrate a simple CUDA program that consists of both host code and a device kernel. The code is hardwired to use thread blocks of 256 threads each.[8] However, the number of thread blocks used depends on the length of the vectors (n). If n is 750, three thread blocks will be used. If n is 4000, 16 thread blocks will be used. If n is 2,000,000, 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. The programmer 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 64 or 128 blocks in parallel. This gives CUDA kernels scalability in execution speed with hardware. That is, the same code

[8] While we use an arbitrary block size 256 in this example, the block size should be determined by a number of factors that will be introduced later.

runs at lower speed on small GPUs and at higher speed on larger GPUs. We will revisit this point in Chapter 4, Compute Architecture and Scheduling.

It is important to point out again that the vector addition example is used for its simplicity. In practice, the overhead of allocating device memory, input data transfer from host to device, output data transfer from device to host, and deallocating device memory will likely make the resulting code slower than the original sequential code in Fig. 2.4. This is because the amount of calculation that is done by the kernel is small relative to the amount of data processed or transferred. Only one addition is performed for two floating-point input operands and one floating-point output operand. Real applications typically have kernels in which much more work is needed relative to the amount of data processed, which makes the additional overhead worthwhile. Real applications also tend to keep the data in the device memory across multiple kernel invocations so that the overhead can be amortized. We will present several examples of such applications.

2.7 Compilation

We have seen that implementing CUDA C kernels requires using various extensions that are not part of C. Once these extensions have been 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 C compiler). As is shown at the top of Fig. 2.14, the NVCC

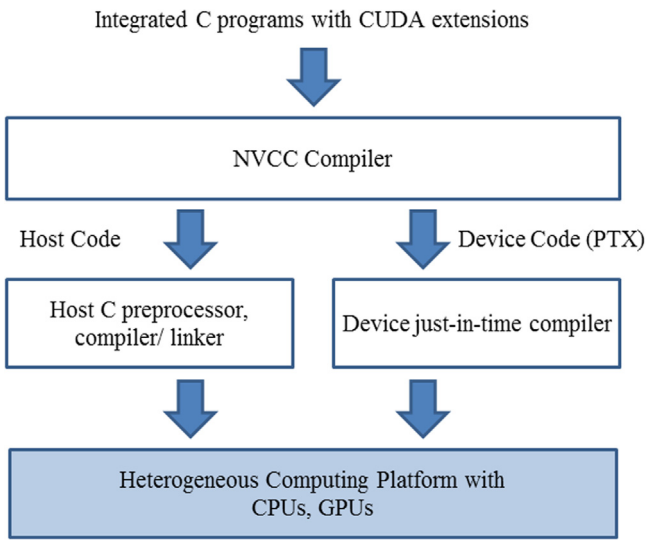


FIGURE 2.14

Overview of the compilation process of a CUDA C program.

compiler processes a CUDA C program, using the CUDA keywords to separate the host code and device code. The host code is straight ANSI C code, which is compiled with the host's standard C/C++ compilers and is run as a traditional CPU process. The device code, which is marked with CUDA keywords that designate CUDA kernels and their associated helper functions and data structures, is compiled by NVCC into virtual binary files called PTX files. These PTX files are further compiled by a runtime 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 your convenience we summarize the extensions that we have discussed in this chapter as follows:

2.8.1 Function declarations

CUDA C extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Fig. 2.12. Using one of "`__global__`," "`__device__`," or "`__host__`," a CUDA C 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 "`__host__`" and "`_device__`" are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host. If a function declaration does not have any CUDA C extension keyword, the function defaults into a host function.

2.8.2 Kernel call and grid launch

CUDA C extends the C function call syntax with kernel execution configuration parameters surrounded by <<< and >>>. 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 (NVIDIA, 2021) for more details of the kernel launch extensions as well as other types of execution configuration parameters.

2.8.3 Built-in (predefined) variables

CUDA kernels can access a set of built-in, predefined 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, Multidimensional Grids and Data, we will discuss more details of using these variables.

2.8.4 Runtime application programming interface

CUDA supports a set of API functions to provide services to CUDA C programs. The services that we discussed in this chapter are `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 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 C program. The chapter is by no means a comprehensive account of all CUDA features. Some of these features will be covered in the remainder of the book. However, our emphasis will be on the key parallel computing concepts that are supported by these features. We will introduce only the CUDA C features that are needed in our code examples for parallel programming techniques. In general, we would like to encourage the reader to always consult the CUDA C Programming Guide for more details of the CUDA C features.

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)?
   (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 processes `2*blockDim.x` consecutive elements that form two sections. All threads in each block will 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 minimum 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. 5. If we want to allocate an array of v integer elements in the CUDA device `global` memory, what would be an appropriate expression for the second argument of the `cudaMalloc` call?

(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, cudaMemcpyDeviceTHost);`

(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 following CUDA kernel and the corresponding host function that calls it:

```
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        }
```

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 summer intern was 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. What is your response?

References

Atallah, M.J. (Ed.), 1998. Algorithms and Theory of Computation Handbook. CRC Press.

Flynn, M., 1972. Some computer organizations and their effectiveness. IEEE Trans. Comput. **C-** 21, 948.

NVIDIA Corporation, March 2021. NVIDIA CUDA C Programming Guide.

Patt, Y.N., Patel, S.J., 2020. ISBN-10: 1260565912, 2000, 2004 Introduction to Computing Systems: From Bits and Gates to C and Beyond. McGraw Hill Publisher.

CHAPTER

Multidimensional grids and data

3

Chapter Outline

In Chapter 2, Heterogeneous Data Parallel Computing, we learned to write a simple CUDA C++ program that launches a one-dimensional grid of threads by calling a kernel function to operate 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 will look more generally at how threads are organized and learn how threads and blocks can be used to process multidimensional arrays. Multiple examples will be 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 coordinates, that is, 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, Heterogeneous Data Parallel Computing, these threads are organized into a two-level hierarchy: A grid consists of one or more blocks, and each block consists of one or more threads. All threads in a 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

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00004-5

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.

In general, a grid is a three-dimensional (3D) array of blocks, and each block is a 3D 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 by 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. The total number of threads in the grid is 128*32 = 4096:

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

Note that `dimBlock` and `dimGrid` are host code variables that are 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 result 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 follows:

```
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 chose to fix the block size at 256. The value of variable n at kernel call time will determine 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 has been launched, the grid and block dimensions will remain the same until the entire grid has finished execution.

For convenience, CUDA provides a special shortcut for calling a kernel with one-dimensional (1D) 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 dimensions 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 "shorthand" 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 in which 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 variables `gridDim` and `blockDim` are preinitialized 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 `vectAddkernel` kernel will 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` are built-in variables in a kernel 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 $2^{31} - 1$,[1] and those 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 3D array of threads. Two-dimensional (2D) blocks can be created by setting `blockDim.z` to 1. One-dimension blocks can be created by setting both `blockDim.y` and `blockDim.z` to 1, as in the `vectorAddkernel` example. As we 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`

[1]Devices with a capability of less than 3.0 allow blockIdx.x to range from 1 to $2^{16} - 1$.

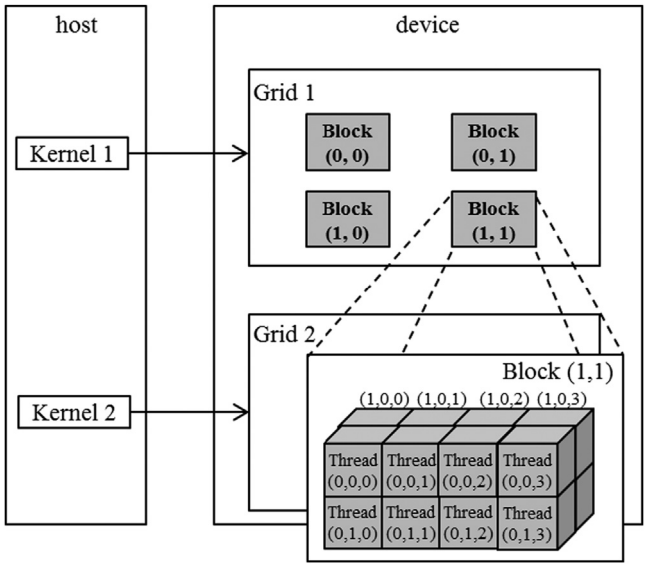


FIGURE 3.1

A multidimensional example of CUDA grid organization.

values of (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowed, but (32, 32, 2) is not allowed because the total number of threads would exceed 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>>>(...);
```

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 an ordering that is the reverse of that used in the C statements for setting configuration parameters, in which 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 $4 \times 2 \times 2$ arrays of threads. Since all blocks within a grid have the

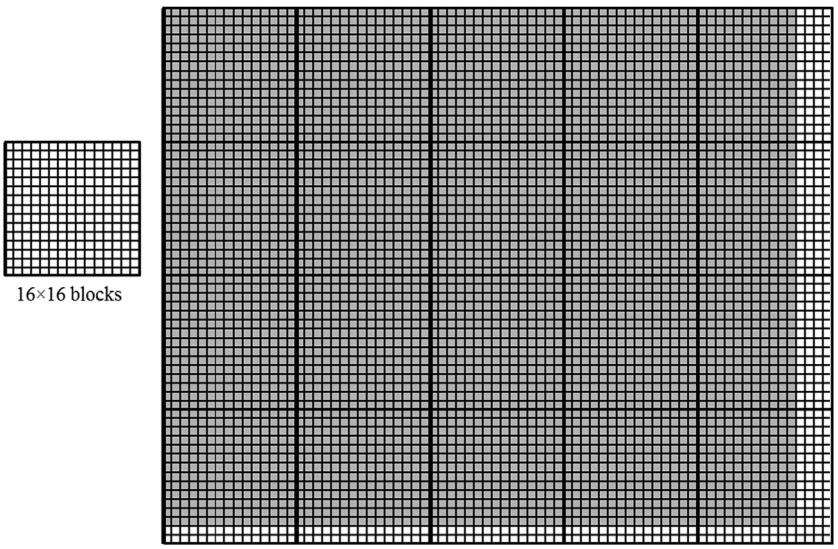


FIGURE 3.2

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

same dimensions, we show only 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 a 2D array 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 such an arrangement for processing a $62 \times 761F1F$[2] picture P (62 pixels in the vertical or y direction and 76 pixels in the horizontal or x

[2]We will refer to the dimensions of multidimensional data in descending order: the z dimension followed by the y dimension, and so on. For example, for a picture of n pixels in the vertical or y dimension and m pixels in the horizontal or x dimension, we will refer to it as a n × 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 on the basis of a multidimensional array that is to be processed by its threads.

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 will need four blocks in the y direction and five 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:

$$\text{Vertical (row) row coordinate} = \text{blockIdx.y}^{*}\text{blockDim.y} + \text{threadIdx.y}$$

$$\text{Horizontal (Column) coordinate} = \text{blockIdx.x}^{*}\text{blockDim.x} + \text{threadIdx.x}$$

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

$$\text{Pin}_{\text{blockIdx.y}^{*}\text{blockDim.y}+\text{threadIdx.y},\text{blockIdx.x}^{*}\text{blockDim.x}+\text{threadIdx.x}} = \text{Pin}_{1^{*}16+0,0^{*}16+0} = \text{Pin}_{16,0}$$

Note that in Fig. 3.2 we have two extra threads in the y direction and four extra threads in the x direction. That is, we will generate 64×80 threads to process 62×76 pixels. This is similar to the situation in which a 1000-element vector is processed by the 1D kernel `vecAddKernel` in Fig. 2.9 using four 256-thread blocks. 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, as follows:

```
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` will result in 125, 94, 16, and 16, respectively.

Before we show the kernel code, we first need to understand how C statements access elements of dynamically allocated multidimensional arrays. Ideally, we

would like to access `Pin_d` as a 2D array in which an element at row j and column i can be accessed as `Pin_d[j][i]`. However, the ANSI C standard on the basis of which CUDA C was developed 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 runtime. Thus the information on the number of columns in a dynamically allocated 2D array is not known at compile time by design. As a result, programmers need to explicitly linearize, or "flatten," a dynamically allocated 2D array into an equivalent 1D array in the current CUDA C.

In reality, all multidimensional arrays in C are linearized. This is due to the use of a "flat" memory space in modern computers (see the "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 1D array and translates the multidimensional indexing syntax into a 1D offset. In the case of dynamically allocated arrays, the current CUDA C compiler leaves the work of such translation to the programmers, owing to lack of dimensional information at compile time.

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 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 in which a 2D array can be linearized. One 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 slightly more readable. Fig. 3.3 shows an example in which a 4×4 matrix M is linearized into a 16-element 1D array, with all elements of row 0 first, followed by the four elements of row 1, and so on. Therefore the 1D 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 1D index for $M_{2,1}$ is 2*4 + 1 = 9. This is illustrated in Fig. 3.3, in which M_9 is the 1D equivalent to $M_{2,1}$. This is the way in which C compilers linearize 2D arrays.

Another way to linearize a 2D array is to place all elements of the same column in consecutive locations. The columns are then placed one after another into the memory space. This arrangement, called the *column-major layout*, is used by FORTRAN compilers. Note that the column-major layout of a 2D array is equivalent to the row-major layout of its transposed form. We will not spend more time on this except to mention 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

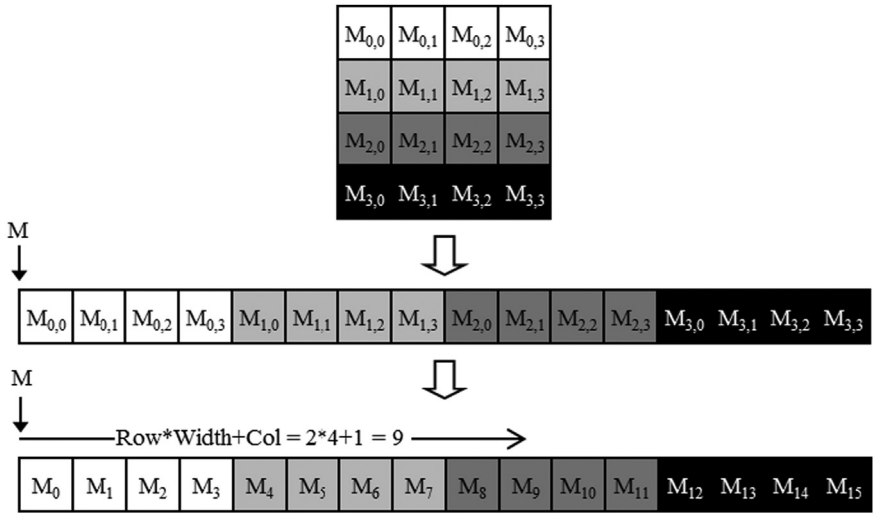


FIGURE 3.3

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 ith column of an array of Width elements in each row.

```
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] = 0.21f*r + 0.71f*g + 0.07f*b;
20      }
21  }
```

FIGURE 3.4

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

designed to be used by FORTRAN programs use the 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 programs.

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

$$L = 0.21^{*}r + 0.72^{*}g + 0.07^{*}b$$

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 width (`m` value passed in from the host code). We have at least as many threads as the number of pixels in the horizontal direction. We also know that there are at least as many threads as the number of pixels in the vertical direction. Therefore as long as we test and make sure that only the threads with both row and column values are within range, that is, `(col<width) && (row<height)`, we will be able to cover every pixel in the picture (line 07).

Since there are `width` pixels in each row, we can generate the 1D index for the pixel at row `row` and column `col` as `row*width+col` (line 10). This 1D index

grayOffset is the pixel index for Pout since each pixel in the output grayscale image is 1 byte (unsigned char). Using our 62×76 image example, the linearized 1D index of the Pout pixel calculated by thread (0,0) of block (1,0) with the following formula:

$$\begin{aligned}&\text{Pout}_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}}\\&= \text{Pout}_{1*16+0,0*16+0} = \text{Pout}_{16,0} = \text{Pout}[16*76 + 0] = \text{Pout}[1216]\end{aligned}$$

As for Pin, we need to multiply the gray pixel index by 32F2F[3] (line 13), since each colored pixel is stored as three elements (r, g, b), each of which is 1 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). In our 62×76 image example the linearized 1D 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 following formula:

$$\begin{aligned}&\text{Pin}_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}} = \text{Pin}_{1*16+0,0*16+0}\\&= \text{Pin}_{16,0} = \text{Pin}[16*76*3 + 0] = \text{Pin}[3648]\end{aligned}$$

The data that is being accessed is the 3 bytes starting at byte offset 3648.

Fig. 3.5 illustrates the execution of colorToGrayscaleConversion in processing our 62×76 example. Assuming 16×16 blocks, calling the colorToGrayscaleConversion kernel generates 64×80 threads. The grid will have $4 \times 5 = 20$ blocks: four in the vertical direction and five 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 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 $16 \times 16 = 256$ threads in each block will process pixels.

The second area, marked 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. The remaining four threads in each row will find their col values out of range and thus will 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.

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

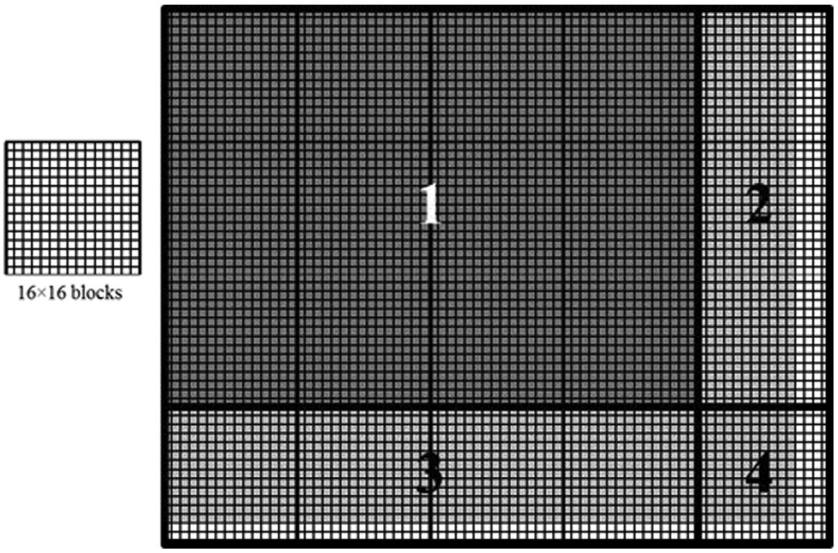


FIGURE 3.5

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

The third area, marked 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. The remaining two threads in each column will not pas 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 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 the array. 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, respectively, in a 3D array. 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 3D 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, Stencil.

3.3 Image blur: a more complex kernel

We have studied `vecAddkernel` and `colorToGrayscaleConversion`, in which 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 C program structure and data parallel execution concepts. At this point, the reader should ask the obvious question: Do all threads in CUDA C programs perform only such simple and trivial operations independently of each other? The answer is no. In real CUDA C 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 complex examples that exhibit these characteristics. We will start with an image-blurring function.

Image blurring smoothes out abrupt variation of pixel values while preserving the edges that are essential for recognizing the key features of the 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 pixel in the input image. As we will learn in Chapter 7, Convolution, the computation of such

FIGURE 3.6

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

weighted sums belongs to the *convolution* pattern. We will be using a simplified approach in this chapter by taking a simple average value of the N × N patch of pixels surrounding, and including, our 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 (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 that was used in `colorToGrayscaleConversion`, we use each thread to calculate an output pixel. That is, the thread-to-output-data mapping remains the same. Thus 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 according to the height and width of the image (line 05). Only the threads whose `col` and `row` indices are both within value ranges are allowed to participate in the execution.

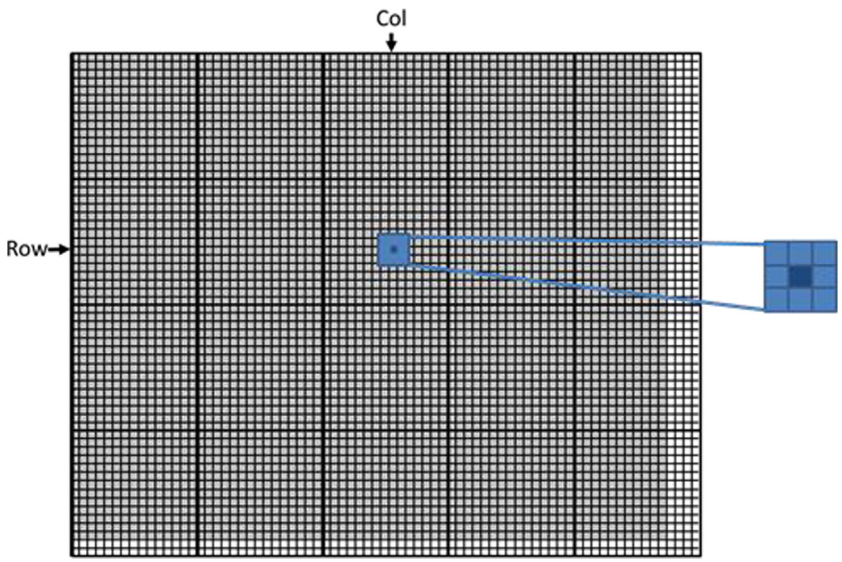


FIGURE 3.7

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)(pixVal/pixels);
23      }
24  }
```

FIGURE 3.8

An image blur kernel.

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 (radius) of the patch and `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 that are 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, in the third iteration of the outer loop, the inner loop iterates through pixels (26, 49), (26, 50), and (26, 51).

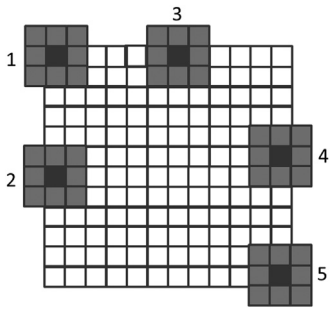


FIGURE 3.9

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

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 have been 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, in computing output pixels near the edge of the image, the patch may extend beyond the valid range of the input image. This is illustrated in Fig. 3.9 assuming 3×3 patches. In case 1, the pixel at the upper-left corner is being blurred. Five 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 $(-1, -1)$, $(-1,0)$, $(-1,1)$, $(0, -1)$, $(0,0)$, $(0,1)$, $(1, -1)$, $(1,0)$, and $(1,1)$. Note that for the five pixels that are outside the image, at least one of the values is less than 0. The `curRow < 0` and `curCol < 0` conditions of the if-statement catch these values and skip 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.

The reader 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 will find all the pixels in their assigned 3×3 patch within the input image. They will accumulate all the nine pixels. However, for the pixels on the four corners, the responsible threads will accumulate only four pixels. For other pixels on the four edges, the responsible threads will accumulate six pixels. These variations are what necessitates keeping track of the actual number of pixels that are accumulated with the variable `pixels`.

3.4 Matrix multiplication

Matrix-matrix multiplication, or matrix multiplication in short, is an important component of the Basic Linear Algebra Subprograms standard (see the "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 using convolutional neural networks, which will be discussed in detail in Chapter 16, Deep Learning.

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 $\boldsymbol{y} = \alpha\boldsymbol{x}+\boldsymbol{y}$*, where* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *are vectors and* α *is a scalar. Our vector addition example is a special case of a level 1 function with* $\alpha = 1$*. Level 2 functions perform matrix-vector operations of the form* $\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x}+\beta\boldsymbol{y}$*, where* $\boldsymbol{A}$ *is a matrix,* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *are vectors, and* α *and* β *are scalars. We will be studying a form of level 2 function in sparse linear algebra. Level 3 functions perform matrix-matrix operations in the form of* $\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$*, where* $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ *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 a matrix multiplication is performed, 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 the rowth position in the vertical direction and the colth 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 rowth row of `M` (shown as a horizontal strip in `M`) and the vector formed by the colth 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,

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

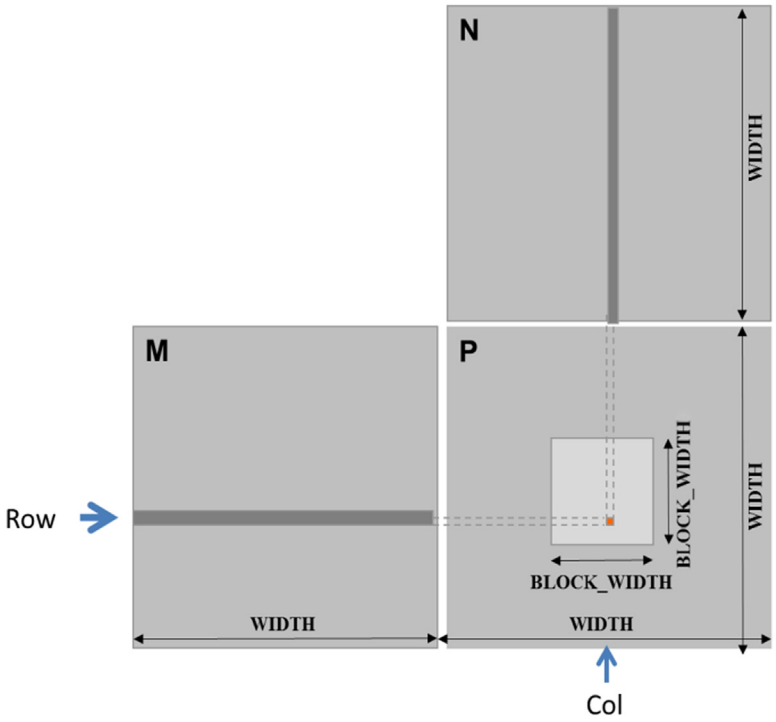


FIGURE 3.10

Matrix multiplication using multiple blocks by tiling P.

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

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

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 `colorToGrayscaleConversion`. 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

```
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.

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 to their counterparts in `colorToGrayscaleConversion`. 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 `Width`. This thread-to-data mapping effectively divides `P` into tiles, one of which is shown as a 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 rowth row of `M` and the colth column of `N`. In Fig. 3.11 we use a for-loop to perform this inner product operation. Before we enter the loop, we initialize a local variable `Pvalue` to 0 (line 06). Each iteration of the loop accesses an element from the rowth row of `M` and an element from the colth 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 rowth row is `M[row*Width]`. Since all elements of a row are placed in consecutive locations, the kth element of the rowth 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 is shown in Fig. 3.11, the beginning element of the colth column is the colth element of row 0, which is `N[col]`. Accessing the next element in the colth 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 kth element of the colth 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 1D 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.

wWe now use a small example to illustrate the execution of the matrix multiplication kernel. Fig. 3.12 shows a 4×4 P with BLOCK_WIDTH = 2. Although such small matrix and block sizes are not realistic, they allow us to fit the entire example into 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 $P_{2,0}$.

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 first 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 + col = $2*4 + 0 = 8$, 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 =

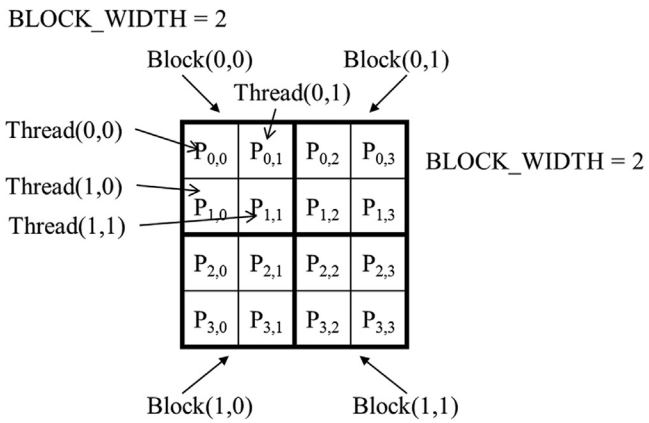


FIGURE 3.12

A small execution example of matrixMulKernel.

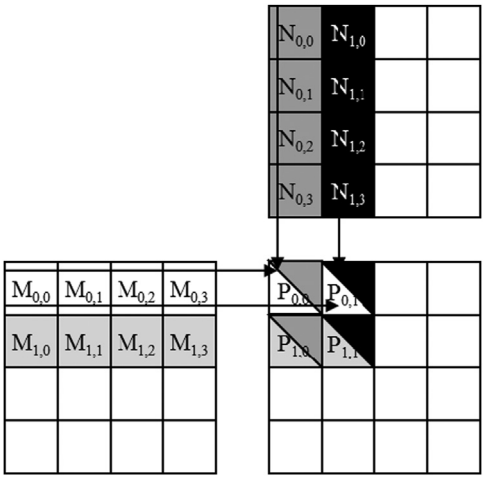


FIGURE 3.13

Matrix multiplication actions of one thread block.

3*4 + 0 = 12, which results in `M[3]` and `N[12]`, the 1D equivalent of $M_{0,3}$ and $N_{3,0}$. We have now 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` will also be limited by these constraints. In the situation in which output matrices larger than this limit are to be computed, one can divide the output matrix into submatrices whose sizes can be covered by a grid and use the host code to launch a different grid for each submatrix. Alternatively, we can change the kernel code so that each thread calculates more `P` elements. We will explore both options later in this book.

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 1D offset. The reason is that dynamically allocated multidimensional arrays in C are typically stored as 1D 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 will be foundational for understanding parallel patterns and their associated optimization techniques.

Exercises

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 produce one output matrix column. Fill in the execution configuration parameters for the design.
   c. Analyze the pros and cons of each of the two kernel designs.
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, that is, $A[i] = \sum^{j} B[i][j] + C[j]$. For simplicity we will handle only square matrices whose elements are single-precision floating-point numbers. Write a matrix-vector multiplication kernel and the host stub function that can be called with four parameters: pointer to the output matrix, pointer to the input matrix, pointer to the input vector, and the number of elements in each dimension. 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 2D 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 3D 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.

CHAPTER

Compute architecture and scheduling

4

Chapter Outline

In Chapter 1, Introduction, 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 Chapters 2, Heterogeneous Data Parallel Computing and 3, Multidimensional Grids and Data, we learned the core features of the CUDA programming interface for creating and calling kernels to launch and execute threads. In the next three chapters we will 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 will start by showing a high-level, simplified view of the compute architecture and explore the concepts of flexible resource assignment, scheduling of blocks, and occupancy. We will 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 that are 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 will present the core concepts and programming considerations of the GPU memory architecture. In particular, Chapter 5, Memory Architecture and Data Locality, focuses on the on-chip memory architecture, and Chapter 6, Performance Considerations, briefly covers the off-chip memory architecture then elaborates on various performance considerations of the GPU architecture as a whole. A CUDA C

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00003-3

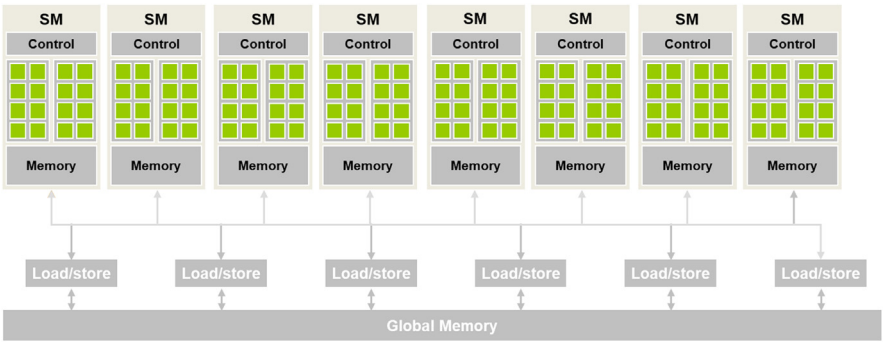


FIGURE 4.1

Architecture of a CUDA-capable GPU.

programmer who masters these concepts is well equipped to write and to understand high-performance parallel kernels.

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. It is organized into an array of highly threaded streaming multiprocessors (SMs). Each SM has several processing units called streaming processors or CUDA cores (hereinafter referred to as just *cores* for brevity), shown as small tiles inside the SMs in Fig. 4.1, that share control logic and memory resources. For example, the Ampere A100 GPU has 108 SMs with 64 cores each, totaling 6912 cores in the entire GPU.

The SMs also come with 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, Memory Architecture and Data Locality. GPUs also come with gigabytes of off-chip device memory, referred to as "Global Memory" in Fig. 4.1. While older GPUs used graphics double data rate synchronous DRAM, more recent GPUs starting with NVIDIA's Pascal architecture may use HBM (high-bandwidth memory) or HBM2, which consist of DRAM (dynamic random access memory) 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, Performance Considerations.

4.2 Block scheduling

When a kernel is called, the CUDA runtime system launches a grid of threads that execute the kernel code. These 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.

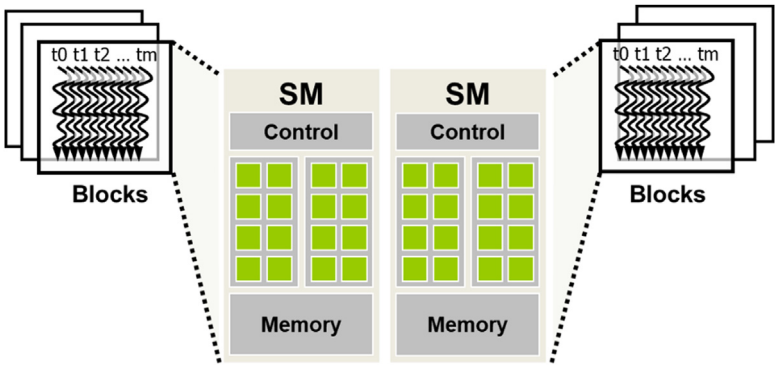


FIGURE 4.2

Thread block assignment to streaming multiprocessors (SMs).

Fig. 4.2 illustrates the assignment of blocks to SMs. Multiple blocks are likely to be simultaneously assigned to the same SM. 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. Most grids contain many more blocks than this number. To ensure that all blocks in a grid get executed, the runtime 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 are scheduled simultaneously on the same SM. This guarantee makes it possible for threads in the same block to interact with each other in ways that threads across different blocks cannot.[1] This includes *barrier synchronization*, which is discussed in Section 4.3. It also includes accessing a low-latency *shared memory* that resides on the SM, which is discussed in Chapter 5, Memory Architecture and Data Locality.

4.3 Synchronization and transparent scalability

CUDA allows threads in the same block to coordinate their activities using the barrier synchronization function `__syncthreads()`. Note that "__" consists of two

[1] Threads in different blocks can perform barrier synchronization through the Cooperative Groups API. However, there are several important restrictions that must be obeyed to ensure that all threads involved are indeed simultaneously executing on the SMs. Interested readers are referred to the CUDA C Programming Guide for proper use of the Cooperative Groups API.

"_" characters. 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. This 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.

Barrier synchronization is a simple and popular method for coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities of multiple people. 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. 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 individuals can be left in the mall when the car leaves, which could seriously damage their friendship!

Fig. 4.3 illustrates the execution of barrier synchronization. There are N threads in the block. Time goes from left to right. Some of the threads reach the barrier synchronization statement early, and some reach it much later. The ones that reach the barrier early will wait for those that arrive late. When the latest one arrives at the barrier, all threads can continue their execution. With barrier synchronization, "no one is left behind."

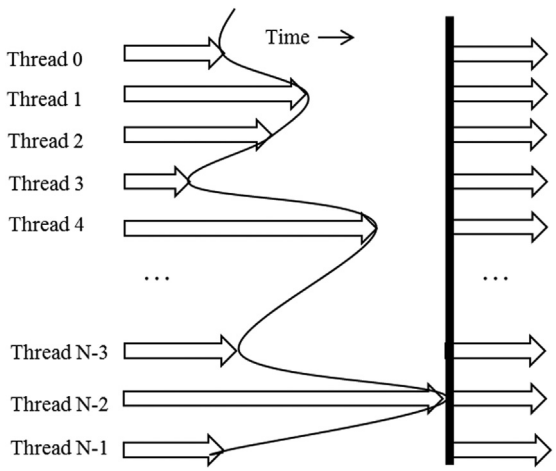


FIGURE 4.3

An example execution 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 when the last thread executes the `__syncthreads` statement, after which all threads are allowed to proceed to execute the statements after the `__syncthreads` statement.

```
01    void incorrect_barrier_example(int n) {
02       ...
03       if (threadIdx.x % 2 == 0) {
04            ...
05            __syncthreads{};
06       }
07       else {
08            ...
09            __syncthreads{};
10        }
11    }
```

FIGURE 4.4

An incorrect use of `__syncthreads()`

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-then-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. For example, in Fig. 4.4, 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 result, 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 a block. These threads should execute in close time proximity with each other to avoid excessively long waiting times. More important, 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 synchronization point can cause a deadlock. The CUDA runtime 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 begin execution only when the runtime system has secured all the resources needed by all threads in the block to complete execution. This ensures the time proximity of all threads in a block and prevents an excessive or even indefinite waiting time during barrier synchronization.

This leads us to an important tradeoff in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization

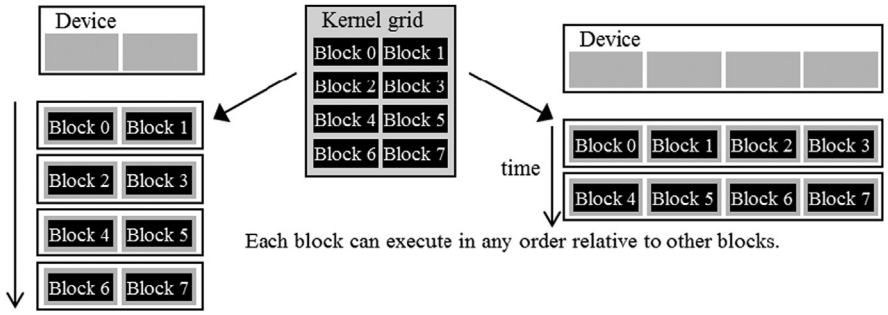


FIGURE 4.5

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

with each other, the CUDA runtime system can execute blocks in any order relative to each other, since none of them need to wait for 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, 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 of blocks simultaneously.

The ability to execute the same application code with a wide range of speeds allows the production of a wide range of implementations 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 blocks 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 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 the use of barrier synchronizations.

Thread scheduling in CUDA GPUs is a hardware implementation concept and therefore must be discussed in the context of specific hardware implementations. In most implementations to date, once a block has been assigned to an SM, 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 block has 256/32 or 8 warps. With three blocks in the SM, we have $8 \times 3 = 24$ warps in the SM.

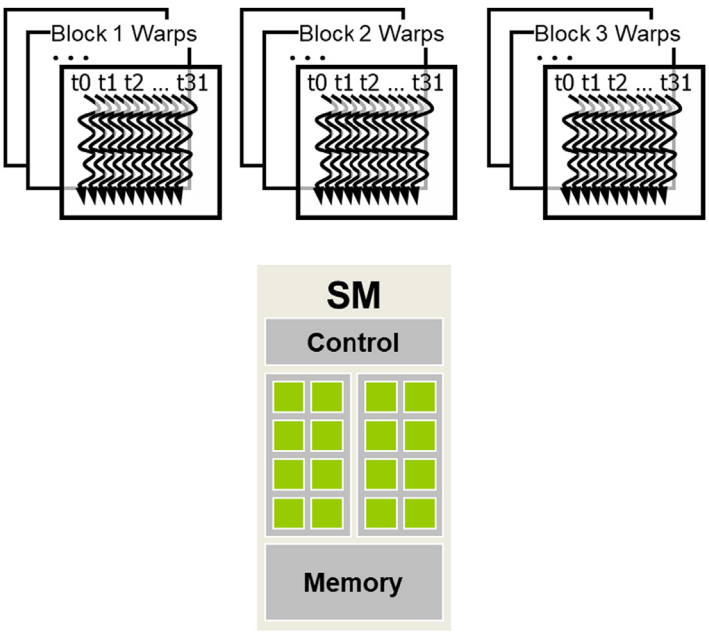


FIGURE 4.6

Blocks are partitioned into warps for thread scheduling.

Blocks are partitioned into warps on the basis of thread indices. If a block is organized into a one-dimensional array, that is, 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 a linear layout. The upper part shows the two-dimensional view of the block. The reader should recognize the similarity to 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 lower part of Fig. 4.7 shows the linearized view of the block. The first four threads are the threads whose `threadIdx.y` value is 0; they are ordered with increasing `threadIdx.x` values. The next four threads are the threads whose `threadIdx.y` value is 1. They are also placed with increasing `threadIdx.x` values. In 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

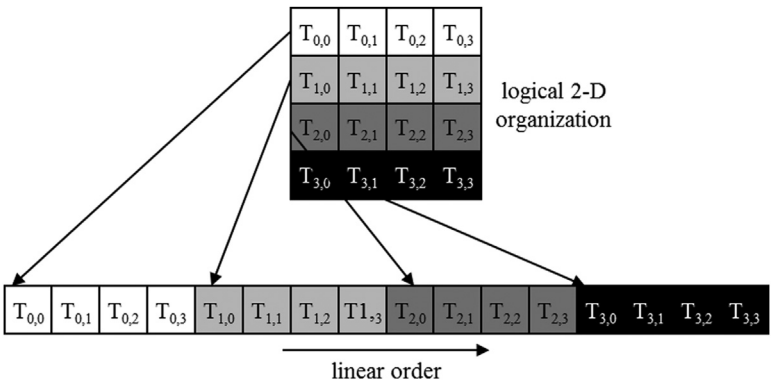


FIGURE 4.7

Placing 2D threads into a linear layout.

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. These threads 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 for all threads in the warp (see the "Warps and SIMD Hardware" sidebar). Fig. 4.8 shows how the cores in an SM are grouped into *processing blocks* in which every 8 cores form a processing block and share an instruction fetch/dispatch unit. As a real example, the Ampere A100 SM, which has 64 cores, is organized into four processing blocks with 16 cores 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 at the same time. These threads apply the same instruction to different portions of the data. Because the SIMD hardware 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.

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

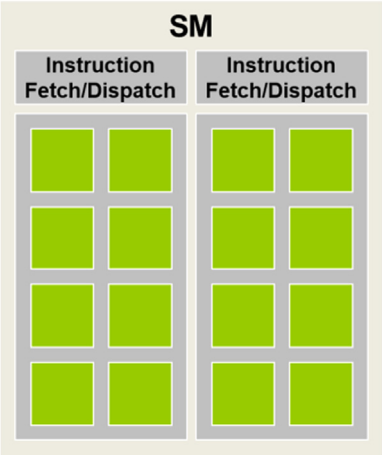


FIGURE 4.8

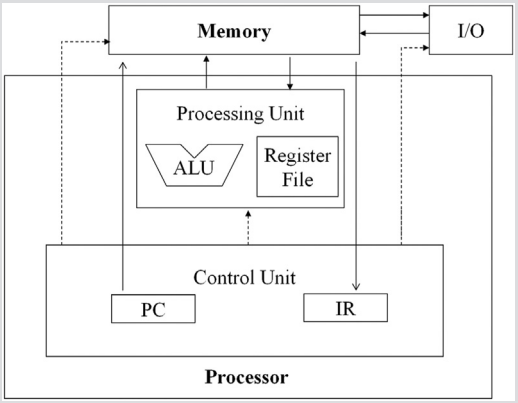
Streaming multiprocessors are organized into processing blocks for SIMD execution.

and a larger percentage to be dedicated to increasing arithmetic throughput. we expect that in the foreseeable future, warp partitioning will remain a popular implementation technique. However, the size of warp can vary from implementation to implementation. Up to this point in time, all CUDA devices have used similar warp configurations in which each warp consists of 32 threads.

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.

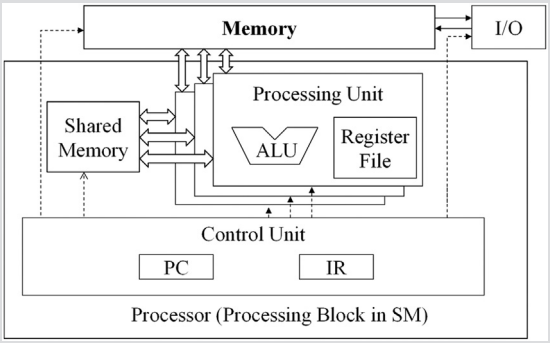
The von Neumann Model is illustrated in the following figure. The computer has an I/O (input/output) that allows both programs and data to be provided to and generated from the system. To execute a program, the computer first inputs the program and its data into the Memory.



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 Figure 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 Figure 4.8) go to multiple processing units that each correspond to a core 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.

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

When threads in the same warp follow different execution paths, we say that these threads exhibit *control divergence*, that is, they diverge in their execution. The multipass 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 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 in which threads 0–23 execute A while threads 24–31 are inactive. The warp will also do another pass through the code in which threads 24–31 execute B while threads 0–23 are inactive. The threads in the warp then reconverge and execute C. In the Pascal architecture and prior architectures, these passes are executed sequentially,

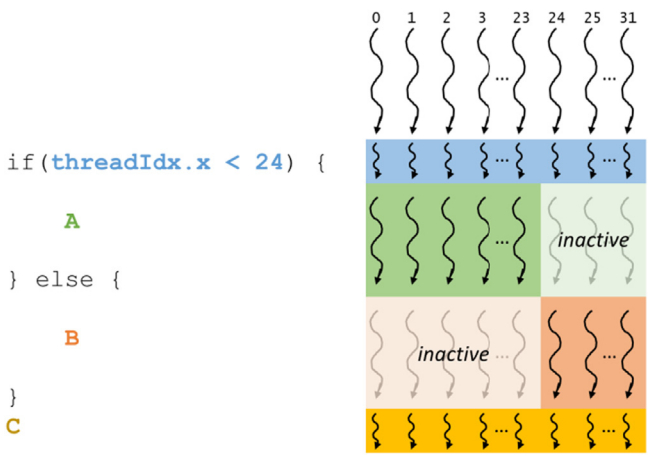


FIGURE 4.9

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

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 as *independent thread scheduling*. Interested readers are referred to the whitepaper on the Volta V100 architecture (NVIDIA, 2017) for details.

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 remaining iterations, some threads execute A, while others are inactive because they have completed their iterations.

One can determine whether 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 that of threads 3, 4, 5, and so on. 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, Heterogeneous Data Parallel Computing, 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

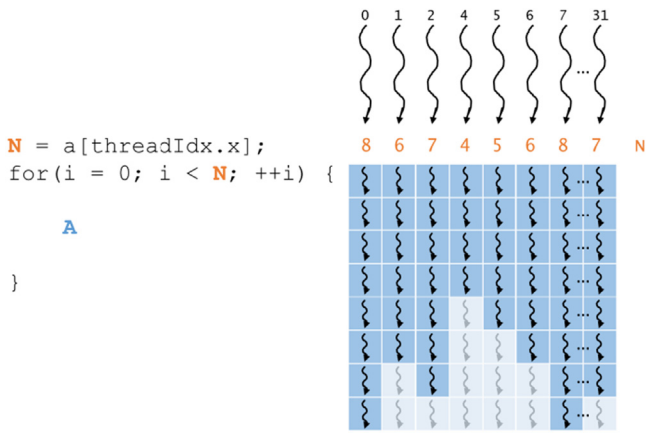


FIGURE 4.10

Example of a warp diverging at a for-loop.

and we picked 64 as the 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 that is not expected or not allowed by the original program. Keep in mind that these 16 blocks are partitioned into 32 warps. Only the last warp (i.e., the second warp in the last block) will have control divergence.

Note that the performance impact of 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 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 on 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, Multidimensional Grids and Data, 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. A total 160 warps (8 warps per block) are 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 7 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 of $130 = 13 \times 10$ 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%.

An important implication of control divergence is that one cannot assume that all threads in a warp have the same execution timing. 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 barrier synchronization mechanism such as `__syncwarp()` to ensure correctness.

4.6 Warp scheduling and latency tolerance

When threads are assigned to SMs, there are usually more threads assigned to an SM than there are cores 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. In either case, the hardware can execute instructions only for a subset of all warps in the SM. A legitimate question is why we need to have so many warps assigned to an SM if it can execute only 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. If more than one warp is ready for execution, a priority mechanism is used to select one for execution. This mechanism of filling the latency time of operations from some threads with work from other threads is often called "latency tolerance" or "latency hiding" (see the "Latency Tolerance" sidebar).

Latency Tolerance

Latency tolerance is needed in many everyday situations. For example, in post offices, each person who is 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 the clerk serves other customers. In most cases, instead of going to the end of the line, the first customer will be served as soon as he or she finishes the form and the clerk finishes serving the current customer.

We can think of these post office customers as warps and the clerk as a hardware execution unit. The customer who 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 and branch instructions. 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 the instructions of some warps wait for the results of these long-latency operations. The selection of warps that are 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 the "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 do. As a result, GPUs can dedicate more chip area to floating-point execution and memory access channel resources.

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 value 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 register 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. Traditional CPUs incur 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 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 an Ampere A100 GPU, an SM has 64 cores but can have up to 2048 threads assigned to it at the same time. Thus the SM can have up to 32 times more threads assigned to it than its cores can support at any given clock cycle. This oversubscription of threads to SMs is essential for latency tolerance. It increases the chances 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 long-latency operations. 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 of 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.

The execution resources in an SM include registers, shared memory (discussed in Chapter 5, Memory Architecture and Data Locality), thread block slots, and thread slots. These resources are dynamically partitioned across threads to support their execution. For example, an Ampere A100 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 in which each block would receive a fixed amount of resources regardless of its real needs. Fixed partitioning results in wasted thread slots when a block requires 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 Ampere A100, we saw that the block size can be varied from 1024 to 64, resulting in 2−32 blocks per SM, respectively. In all these cases, the total number of threads assigned to the SM is 2048, which maximizes occupancy. Consider, however, the case when each block has 32 threads. In this case, the 2048 thread slots

would need to be partitioned and assigned to 64 blocks. However, the Volta SM can support only 32 blocks slots at once. This means that only 1024 of the thread slots will be utilized, that is, 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 affect occupancy occurs when the maximum number of threads per block is not divisible by the block size. In the example of the Ampere A100, we saw that up to 2048 threads per SM can be supported. However, if a block size of 768 is selected, the SM will be able to accommodate only 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 are reached. The occupancy in this case is (1536 assigned threads)/(2,048 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, Memory Architecture and Data Locality, 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. Therefore 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 potential impact of register resource limitations on occupancy. For example, the Ampere A100 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. For example, if 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. However, this 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 analysis is done for the shared memory resource in Chapter 5, Memory Architecture and Data Locality.

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) $\times$ (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, bumping 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. However, this 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," in which a slight increase in resource usage can result in significant reduction in parallelism and performance achieved (Ryoo et al., 2008).

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 CUDA Occupancy Calculator (CUDA Occupancy Calculator, Web) which is a downloadable spreadsheet that calculates the actual number of threads running on each SM for a particular device implementation given the usage of resources by a kernel.

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 the "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 toothpaste and even toothbrushes, while most American hotels do not. Many American hotels provide both shampoo and conditioner, while hotels in other continents often provide only shampoo.

If the room comes with a microwave oven and a refrigerator, we can take the leftovers from dinner and expect to eat them the next 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 the properties at hotel websites, choose the hotels that best 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 resources are available in each SM. The compute capability of GPUs tends to increase from generation to generation. The Ampere A100 GPU has compute capability 8.0.

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

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

While it may not be obvious, a modern PC system often has two or more CUDA 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:

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

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 maximum 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 a sufficient number of threads in each block as far as the application is concerned.

The number of SMs in the device is given in `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 maximum hardware execution throughput of the device.

The host code can find the maximum 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 of 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 maximum number of blocks allowed along each dimension of a grid in `devProp.maxGridSize[0]` (for the *x* dimension), `devProp.maxGridSize[1]` (for the *y* dimension), and `devProp.maxGridSize[2]` (for the *z* dimension). A typical use of this information is to determine whether a grid can have enough threads to handle the entire dataset or some kind of iterative approach is needed.

The field `devProp.regsPerBlock` gives the number of registers that are available in each SM. This field can be useful in determining whether the kernel can achieve maximum occupancy on a particular device or 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 that a block can use is indeed the same as the total number of registers that are available in the SM. However, for some compute capability levels, the maximum number of registers that a block can use is less than the total that are 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 GPU is organized into SM, which consist of multiple processing blocks of cores that share control logic and memory resources. When a grid is launched, its blocks are assigned to SMs in an arbitrary order, resulting in transparent scalability of CUDA applications. The transparent scalability comes with a limitation: Threads in different blocks cannot synchronize with each other.

Threads are assigned to SMs for execution on a block-by-block basis. Once a block has been assigned to an SM, it is further partitioned into warps. Threads in a warp are executed following the SIMD model. If threads in the same warp diverge by taking different execution paths, the processing block executes these paths in passes in which each thread is active only 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 small subset of its resident warps. This allows the other warps to wait for long-latency operations without slowing down 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-latency operations.

Each CUDA device imposes a potentially different limitation on the amount of resources 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 programmers with the ability to query the resources available in a GPU at runtime.

Exercises

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

```
01    __global__ void foo_kernel(int* a, int* b) {
02        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03        if(threadIdx.x < 40 || threadIdx.x >= 104) {
04            b[i] = a[i] + 1;
05        }
06        if(i%2 == 0) {
07            a[i] = b[i]*2;
08        }
09        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 microseconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, and 2.9; they 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 can take up to 1536 threads and up to 4 thread blocks, which of the following block configurations 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 are possible. In the cases in which 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 per SM, 32 blocks per SM, and 64K (65,536) registers per SM. For each of the following kernel characteristics, specify whether the kernel can achieve full occupancy. If not, specify the limiting factor.
   a. The kernel uses 128 threads per block and 30 registers per thread.
   b. The kernel uses 32 threads per block and 29 registers per thread.
   c. The kernel uses 256 threads per block and 34 registers per 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

CUDA Occupancy Calculator, 2021. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

NVIDIA (2017). NVIDIA Tesla V100 GPU Architecture. Version WP-08608-001_v1.1.

Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., et al., Program optimization space pruning for a multithreaded GPU. In: Proceedings of the Sixth ACM/IEEE International Symposium on Code Generation and Optimization, April 6−9, 2008.

CHAPTER

Memory architecture and data locality

5

Chapter Outline

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 current GPU hardware and how threads are scheduled to execute on this hardware. 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 studied 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 in which traffic congestion in the global memory access paths prevents all but a 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 traffic to and from the global memory. In this chapter we will study the use of different memory types to boost the execution performance of CUDA kernels.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00018-5

5.1 Importance of memory access efficiency

We can illustrate the effect of memory access efficiency by calculating the expected performance level of the most executed portion of the matrix multiplication kernel code in Fig. 3.11, which is partially replicated in Fig. 5.1. The most important 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.

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 ratio of floating-point operations (FLOP) to bytes (B) accessed from global memory is 2 FLOP to 8 B, or 0.25 FLOP/B. We will refer to this ratio as the *compute to global memory access ratio*, defined as the number of FLOPs performed for each byte access from the global memory within a region of a program. This ratio is sometimes also referred to as *arithmetic intensity* or *computational intensity* in the literature.

The compute to global memory access ratio has major implications for the performance of a CUDA kernel. For example, the Ampere A100 GPU has a peak global memory bandwidth of 1555 GB/second. Since the matrix multiplication kernel performs 0.25 OP/B, the global memory bandwidth limits the throughput of single-precision FLOPs that can be performed by the kernel to 389 giga FLOPs per second (GFLOPS), obtained by multiplying 1555 GB/second with 0.25 FLOP/B. However, 389 GFLOPS is only 2% of the peak single-precision operation throughput of the A100 GPU, which is 19,500 GFLOPS. The A100 also comes with special purpose units called *tensor cores* that are useful for accelerating matrix multiplication operations. If one considers the A100's tensor-core peak single-precision floating-point throughput of 156,000 GFLOPS, 389 GFLOPS is only 0.25% of the peak. Thus the execution of the matrix multiplication kernel is severely limited by the rate at which the data can be delivered from memory to the GPU cores. We refer to programs whose execution speed is limited by memory bandwidth as *memory-bound* programs.

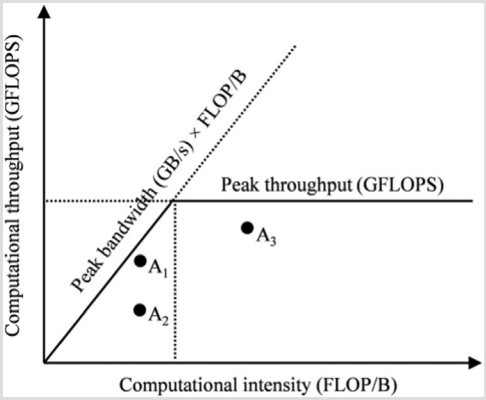
```
07    for (int k = 0; k < Width; ++k) {
08        Pvalue += M[row*Width+k] * N[k*Width+col];
09    }
```

FIGURE 5.1

The most executed part of the matrix multiplication kernel in Fig. 3.11.

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. A point in the plot represents an application with its operational 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 application's 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. The point of intersection between these two lines represents the computational intensity value at which applications transition from being memory bound to being compute bound. Applications with lower computational intensity are memory-bound and cannot achieve peak throughput because they are limited by memory bandwidth. Applications with higher computational intensity are compute-bound and are not limited by memory bandwidth.

As an example, points A_1 and A_2 both represent memory-bound applications, while A_3 represents a compute-bound application. A_1 uses resources efficiently and operates close to the peak memory bandwidth, whereas A_2 does not. For A_2, there may be room for additional optimizations to improve throughput by improving memory bandwidth utilization. However, for A_1 the only way to improve throughput is to increase the computational intensity of the application.

To achieve higher performance for this kernel, we need to increase the compute to global memory access ratio of the kernel by reducing the number of global memory accesses it performs. For example, to fully utilize the 19,500 GFLOPS that the A100 GPU provides, a ratio of at least (19,500 GOP/second)/(1555 GB/second)=12.5 OP/B is needed. This ratio means that for every 4-byte floating point value accessed, there must be about 50 floating-point operations performed! The extent to which such a ratio can be achieved depends on the intrinsic data reuse in the computation at hand. We refer the reader to the "The Roofline Model" sidebar for a useful model for analyzing a program's potential performance with respect to its compute intensity.

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.

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. Fig. 5.2 shows these CUDA device memories. At the bottom of the figure, 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 introduced global memory in Chapter 2, Heterogeneous Data Parallel Computing, and we will look at constant memory in detail in Chapter 7, Convolution.

Another type of memory is the local memory, which can also be read and written. The local memory is actually placed in global memory and has similar access latency, but it is 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 data that is private to the thread but cannot be allocated in registers. This data includes statically allocated arrays, spilled registers, and other elements of the thread's call stack.

Registers and shared memory in Fig. 5.2 are on-chip memories. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual threads; each thread can access only its own registers (see the "CPU versus GPU Register Architecture" sidebar). A kernel function typically uses registers to hold frequently accessed variables that are private to each thread. 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 by which threads can cooperate by sharing their input data and intermediate results. By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

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, Compute Architecture and Scheduling, 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, Compute Architecture and Scheduling, 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.

To fully appreciate the difference between registers, shared memory, and global memory, we need to go into a little more detail about how these different memory types are realized and used in modern processors. As we discussed in the "Warps and SIMD Hardware" sidebar in Chapter 4, Compute Architecture and Scheduling, virtually all modern processors find their root in the model proposed by John von Neumann in 1945, which is shown in Fig. 5.3. CUDA devices are no exception. The global memory in a CUDA device maps to the Memory box in Fig. 5.3. The

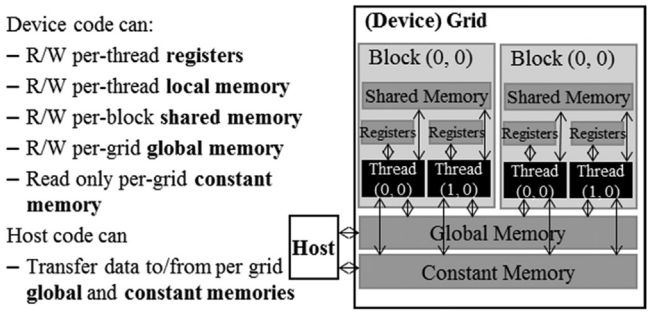


FIGURE 5.2

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

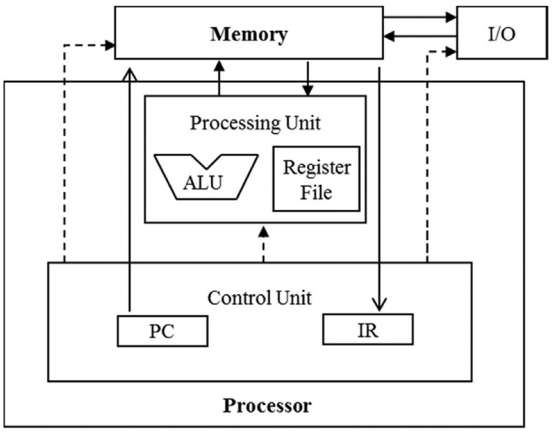


FIGURE 5.3

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

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 each access to registers involves fewer instructions than an access to the global memory. Arithmetic instructions in most modern processors have "built-in" register operands. For example, a floating-point addition instruction might be of the following form:

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

where r2 and r3 are the register numbers that specify 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, no additional instruction is 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 that are 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 fetch and execute only 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 that is consumed for accessing a value 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 that are available to each thread is quite limited in today's GPUs. As we saw in Chapter 4, Compute Architecture and Scheduling, the occupancy that is achieved for an application can be reduced if the register usage in full-occupancy scenarios exceeds the limit. Therefore we also need to avoid oversubscribing to this limited resource whenever possible.

Fig. 5.4 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 resides in the shared memory, it needs to perform a memory load operation, just as in 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 a block. As shown in Fig. 5.4, a CUDA device SM typically employs multiple processing units to allow multiple threads to make simultaneous progress (see the "Threads" sidebar) in Chapter 2, Heterogeneous Data Parallel Computing. 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 a block. We will be learning several important types of parallel algorithms that can greatly benefit from such efficient data sharing among threads.

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. Table 5.1 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

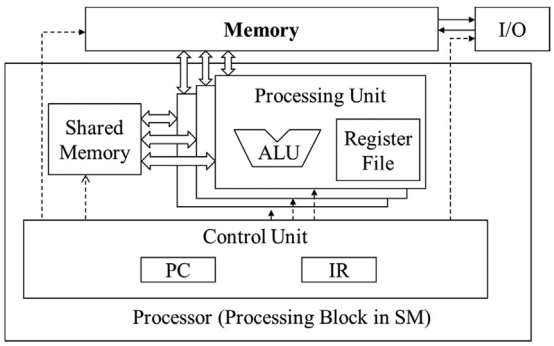


FIGURE 5.4

Shared memory versus registers in a CUDA device SM.

Table 5.1 CUDA variable declaration type qualifiers and the properties of each type.

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

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 access only 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 available to all kernels.

We refer to variables that are not arrays as *scalar* variables. As shown in Table 5.1, all automatic scalar variables that are 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 cease to exist. In Fig. 5.1, 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 affect the occupancy of each SM, as we saw in Chapter 4, Compute Architecture and Scheduling.

Automatic array variables are not stored in registers.[1] Instead, they are stored into the thread's local memory and may incur long access delays and potential

[1]There are some exceptions to this rule. The compiler may decide to store an automatic array into registers if all accesses are done with constant index values.

access congestions. The scope of these arrays, like that of 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 cease to exist. From our experience, one seldom needs to use automatic array variables in kernel functions and device functions.

If a variable declaration is preceded by the `__shared__` keyword (each "__'' consists of two "_'' characters), it declares a shared 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. Shared variables reside in the shared memory. The scope of a shared variable is within a thread block; that is, all threads in a block see the same version of a shared variable. A private version of the shared variable is created for and used by each block during kernel execution. The lifetime of a shared variable is within the duration of the kernel execution. When a kernel terminates its grid's execution, the contents of its shared variables cease to exist. As we discussed earlier, shared variables are an efficient means for threads within a block to collaborate with each other. Accessing shared variables from the shared memory is extremely fast and highly parallel. CUDA programmers often use shared variables to hold the portion of global memory data that is frequently used and reused in an execution phase of the kernel. One may need to adjust the algorithms that are 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 keyword `__constant__`' (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 for variables that 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 to 65,536 bytes. One may need to break up the input data volume to fit within this limitation. We will demonstrate the usage of constant memory in Chapter 7, Convolution.

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 global memory. Accesses to a global variable are slow. Latency and throughput of accessing global variables have been improved with caches in more recent devices. One important advantage of global variables is that they are

visible to all threads of all kernels. Their contents also persist through the entire execution. Thus global variables can be used as a means for threads to collaborate across blocks. However, one must be aware that there is currently no easy way to synchronize between threads from different thread blocks or to ensure data consistency across threads in accessing global memory other than using atomic operations or terminating the current kernel execution.[2] Therefore global variables are often used to pass information from one kernel invocation to another kernel invocation.

In CUDA, pointers can be used to point to data objects in the global memory. There are two typical ways in which pointer use arises in kernel and device functions. First, if an object is allocated by a host function, the pointer to the object is initialized by memory allocation API functions such as `cudaMalloc` and can be passed to the kernel function as a parameter, as we saw in Chapter 2, Heterogeneous Data Parallel Computing, and Chapter 3, Multidimensional Grids and Data. The second type of use is to assign the address of a variable that is declared in the global memory to a pointer variable. For example, the statement `{float* ptr=&GlobalVar;}` in a kernel function assigns the address of `GlobalVar` into an automatic pointer variable `ptr`. The reader should refer to the CUDA Programming Guide for using pointers in other memory types.

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 can each 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, Multidimensional Grids and Data. 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. For brevity we abbreviate P[y*Width+x], M[y*Width+x], and N[y*Width+x] into $P_{y,x}$, $M_{y,x}$, and $N_{y,x}$, respectively. This example assumes that we use four $\times$ 2 blocks to compute the P matrix. The heavy boxes in the P matrix define the P elements that are 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

[2]One can use CUDA memory fencing to ensure data coherence between thread blocks if the number of thread blocks is smaller than the number of SMs in the CUDA device. See the CUDA Programming Guide for more details.

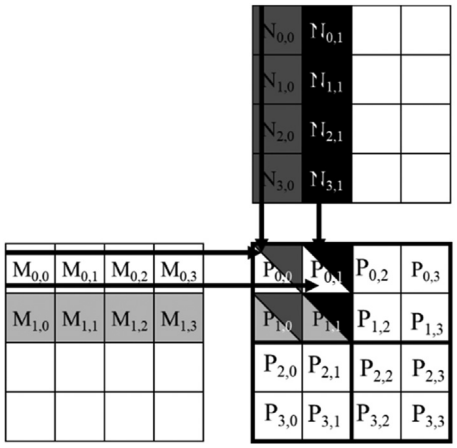


FIGURE 5.5

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

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 that 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. 3.11 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 $thread_{0,1}$ collaborate so that these M elements are loaded from global memory only 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 collaborate in their accesses to global memory, we can reduce the 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 dimension of the blocks that are used. With Width $\times$ Width blocks, the potential reduction of global memory traffic would be Width. That is, if we use 16×16 blocks, we can potentially reduce the global memory traffic to 1/16 of the original level through collaboration between threads.

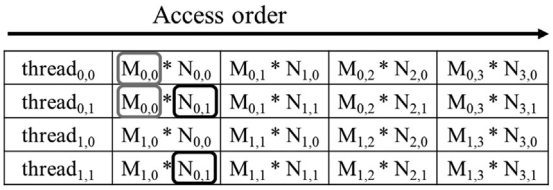


FIGURE 5.6

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

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

In Fig. 5.7 we divide M and N into 2×2 tiles, as delineated by the thick lines. The dot product calculations that are 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 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 need to show only 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 accesses to the global memory by a factor of 2. 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

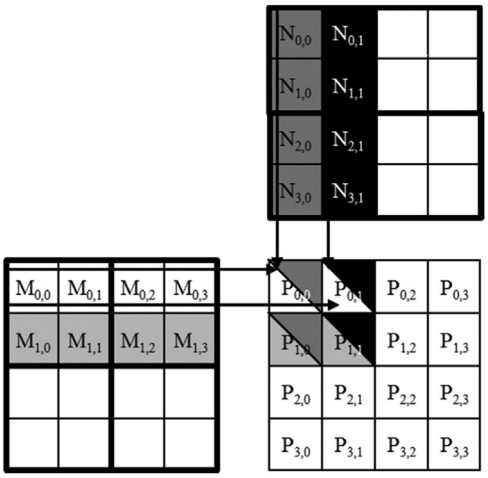


FIGURE 5.7

Tiling M and N to utilize shared memory.

	Phase 0			Phase 1		
$thread_{0,0}$	$\mathbf{M_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$	$\mathbf{M_{0,2}}$ ↓ $Mds_{0,0}$	$\mathbf{N_{2,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{0,1}$*$Nds_{1,0}$
$thread_{0,1}$	$\mathbf{M_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{0,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$	$\mathbf{M_{0,3}}$ ↓ $Mds_{0,1}$	$\mathbf{N_{2,1}}$ ↓ $Nds_{0,1}$	$PValue_{0,1}$ += $Mds_{0,0}$*$Nds_{0,1}$ + $Mds_{0,1}$*$Nds_{1,1}$
$thread_{1,0}$	$\mathbf{M_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$	$\mathbf{M_{1,2}}$ ↓ $Mds_{1,0}$	$\mathbf{N_{3,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{1,0}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{1,0}$
$thread_{1,1}$	$\mathbf{M_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{1,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{M_{1,3}}$ ↓ $Mds_{1,1}$	$\mathbf{N_{3,1}}$ ↓ $Nds_{1,1}$	$PValue_{1,1}$ += $Mds_{1,0}$*$Nds_{0,1}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

FIGURE 5.8

Execution phases of a tiled matrix multiplication.

that Pvalue is an automatic variable, so a private version is generated for each thread. We added subscripts just to 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 is shown 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 reused 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 multicore CPUs as in many-thread GPUs We will return to the concept of locality in Chapter 6, Performance Considerations.

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, lines 04 and 05 declare `Mds` and `Nds`, respectively, 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 because all threads in a block must have access to the M and N elements that are 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 by the runtime system 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, respectively, 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 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

```
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.

`bx*TILE_WIDTH` elements of P. Another `tx` thread within the same block would cover another `tx` elements. Thus the thread with `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 `Col`th column and the `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

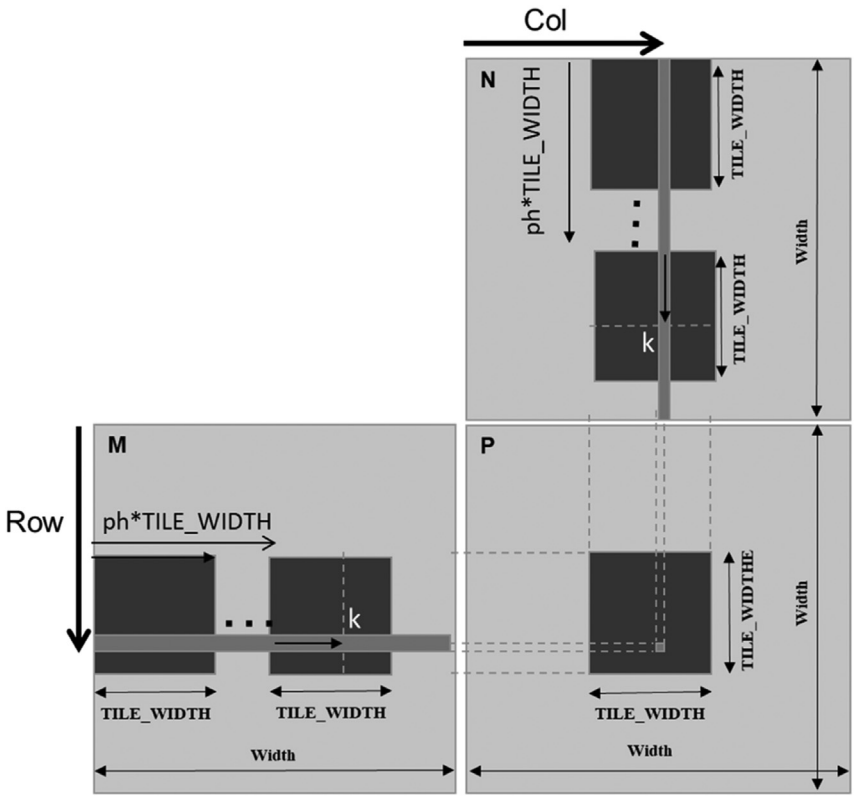


FIGURE 5.10

Calculation of the matrix indices in tiled multiplication.

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, respectively, 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} `M` elements and `TILE_WIDTH`^{2} `N` elements 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 by 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` (the `threadIdx.x` value) 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` (the `threadIdx.y` value) 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, in 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 Figs. 5.7 and 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, Compute Architecture and Scheduling, 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. Thus none of the threads would load the elements too early and corrupt the input values of other threads.

The two `__syncthreads()` calls in lines 21 and 26 demonstrate two different types of data dependence that parallel programmers often have to reason about when they are 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.

The loop nest from line 16 to line 28 illustrates a technique called *strip-mining*, which takes a long-running loop and break it into phases. Each phase involves 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 in the same block to focus their work on the same section of input data during each phase. Strip-mining is an important means to creating the phases that are needed by tiling in data parallel programs.[3]

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 16×16 tiles, one can reduce the global memory accesses by a factor of 16. This increases the compute to global memory access ratio from 0.25 OP/B to 4 OP/B. This improvement allows the memory bandwidth of a CUDA device to support a higher computation rate. For example, in the A100 GPU which has a global memory bandwidth of 1555 GB/second, this improvement allows the device to achieve (1555 GB/second)*(4 OP/B)=6220 GFLOPS, which is substantially higher than the 389 GFLOPS achieved by the kernel that did not use tiling.

Although tiling improves throughput substantially, 6220 GFLOPS is still only 32% of the device's peak throughput of 19,500 GFLOPS. One can further optimize the code to reduce the number of global memory accesses and improve throughput. We will see some of these optimizations later in the book, while other advanced optimizations will not be covered. Because of the importance of matrix multiplication in many domains, there are highly optimized libraries, such as cuBLAS and CUTLASS, that already incorporate many of these advanced optimizations. Programmers can use these libraries to immediately achieve close to peak performance in their linear algebra applications.

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 that is 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 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.

[3]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.

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 removes 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 3×3 M, N, and P matrices. 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 (which is 2). Fig. 5.11 shows the memory access pattern during the second phase of $block_{0,0}$. We see that $thread_{0,1}$ and $thread_{1,1}$ will attempt to load M elements that do not exist. Similarly, we see that $thread_{1,0}$ and $thread_{1,1}$ will attempt to access N elements that do not exist.

Accessing nonexisting elements is problematic in two ways. In the case of accessing a nonexisting element that is 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 two-dimensional matrices.

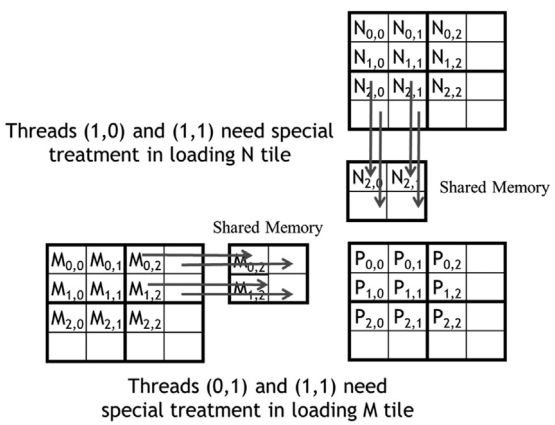


FIGURE 5.11

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

The element after $M_{0,2}$ in the linearized layout is $M_{1,0}$. Although $thead_{0,1}$ 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 in accessing an element that is 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. In other systems these accesses will be rejected, causing 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 arise only 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 nonexisting 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. For example, as we saw 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 nonexisting $M_{0,3}$ 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/

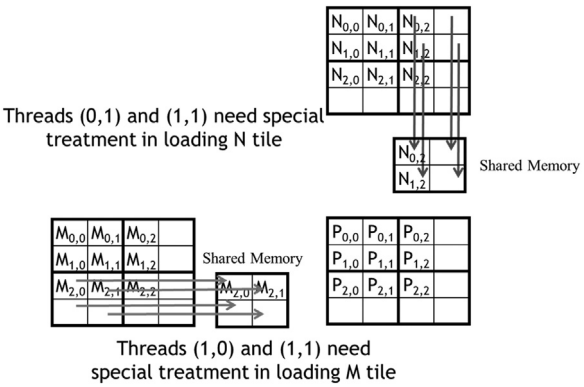


FIGURE 5.12

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

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, in 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 0.0, a value that will not cause any harm if it is used in the inner product calculation. If any thread uses this 0.0 value in the calculation of its inner product, there will not be any change in the inner product value.

Finally, a thread should store its final inner product value only 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 tile matrix multiplication kernel is just one 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 × k M matrix multiplied with a k × l N matrix results in a j × l P matrix. Our kernel can handle only square matrices so far.

Fortunately, it is quite easy to extend our kernel further 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`, `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, Compute Architecture and Scheduling, 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 careful to stay within the SM's capacity of these memories. Each CUDA device offers limited resources, which limits 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.

We saw in Chapter 4, Compute Architecture and Scheduling, how register usage can be a limiting factor for occupancy. Shared memory usage can also limit the number of threads that can be assigned to each SM. For example, the A100 GPU can be configured to have up to 164 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 (164 KB)/(2048 threads)=82 B/thread. In the tiled matrix multiplication example, every block has `TILE_WIDTH`^{2} threads, and uses `TILE_WIDTH`^{2}*4B of shared memory for `Mds` and `TILE_WIDTH`^{2}*4B of shared memory for `Nds`. Thus the thread block uses an average of (`TILE_WIDTH`^{2}*4B + `TILE_WIDTH`^{2}*4B)/(`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 a kernel that has thread blocks that use 32 KB of shared memory, each of which has 256 threads. In this case, the kernel uses an average of (32 KB)/(256 threads)=132 B/thread of shared memory. With such shared memory usage, the kernel cannot achieve full occupancy. Each SM can host a maximum of only (164 KB)/(132 B/thread)=1272 threads. Therefore the maximum achievable occupancy of this kernel will be (1272 assigned threads)/(2048 maximum threads)=62%.

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. That is, we may want a host code to dynamically determine the size of the shared memory and adjust the amount of shared memory that is 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 that is available in each SM. The programmer can then determine the amount of shared memory that should be used by each block.

Unfortunately, the kernels in Figs. 5.9 and 5.13 do not support any dynamic adjustment of shared memory usage by the host code. The declarations that are 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 16
```

both `Mds` and `Nds` will have 256 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 such adjustment with a different style of declaration in CUDA by adding a `C 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:

```
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 by using a linearized index based on the vertical and horizontal indices.

At runtime, when we call the kernel, we can dynamically configure the amount of shared memory to be used for each block according to the device query result and supply that as a third configuration parameter to the kernel call. For example, the revised kernel could be launched with the following statements:

```
  size_t size =
calculate_appropriate_SM_usage(devProp.sharedMemPerBlock,
...);

  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 16×16 tile, we have a size of $2 \times 16 \times 16 \times 4$=2048 bytes to accommodate both `Mds` and `Nds`. We have omitted the details of the calculation for setting the value of size at runtime 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 argument is the size of the `Mds` section, and the second argument is the size of the `Nds` section, both in terms of bytes. Note that in the host code above, we passed 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]`.

```
01    #define TILE_WIDTH 16
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.

5.7 Summary

In summary, the execution speed of a program in modern processors can be severely limited by the speed of the memory. To achieve good utilization of the execution throughput of a CUDA devices, one needs to strive for a high compute to global memory access ratio in the kernel code. If the ratio is low, the kernel is memory-bound. That is, 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. Using these memories effectively requires redesign 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.

However, it is 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 have been exceeded, they limit the number of threads that can be executing simultaneously in each SM and can negatively affect 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 C programming, 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 multicore 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 was 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 in 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 that is used by each block according to the hardware capability. We did not discuss the use of registers in tiling. We will explain the use of registers in tiled algorithms when we discuss parallel algorithm patterns in Part II of the book.

Exercises

1. Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: Analyze the elements that are accessed by each thread and see whether 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 that capacity is 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 32×32 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 $N \times N$. How many times is each element in the input matrices requested from global memory when:
   a. There is no tiling?
   b. Tiles of size $T \times T$ are used?
9. A kernel performs 36 floating-point operations and seven 32-bit global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute-bound or memory-bound.
   a. Peak FLOPS=200 GFLOPS, peak memory bandwidth=100 GB/second
   b. Peak FLOPS=300 GFLOPS, peak memory bandwidth=250 GB/second
10. To manipulate tiles, a new CUDA programmer has written a device kernel that 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 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 bytes)?

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, 64K (65,536) registers/SM, and 96 KB of shared memory/SM. For each of the following kernel characteristics, specify whether 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.

CHAPTER

Performance considerations 6

Chapter Outline

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 through hands-on exercises in a parallel programming model that is designed for high performance.

So far, we have learned about various aspects of the GPU architecture and their implications for performance. In Chapter 4, Compute Architecture and Scheduling, we learned about the compute architecture of the GPU and related performance considerations, such as control divergence and occupancy. In Chapter 5, Memory Architecture and Data Locality, we learned about the on-chip memory architecture of the GPU and the use of shared memory tiling to achieve more data reuse. In this chapter we will briefly present the off-chip memory (DRAM) architecture and discuss related performance considerations such as memory coalescing and memory latency hiding. We then discuss an important type of optimization—thread granularity coarsening—that may target any of the different aspects of the architecture, depending on the application. Finally, we wrap up this part of the book with a checklist of common performance optimizations that will serve as a guide for optimizing the performance of the parallel patterns that will be discussed in the second and third parts of the book.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00016-1

In different applications, 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 that is thus alleviated was the dominating constraint before the strategy was applied and the constraint that is thus exacerbated does not have negative effects on parallel execution. Without such an understanding, performance tuning would be guesswork; plausible strategies may or may not lead to performance enhancements.

6.1 Memory coalescing

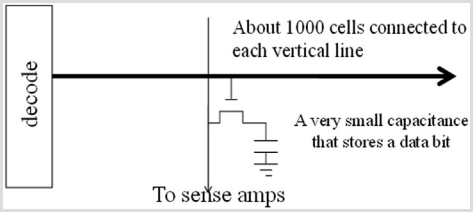
One of the most important factors of CUDA kernel performance is accessing data in the global memory, the limited bandwidth of which can become the bottleneck. CUDA applications extensively exploit data parallelism. Naturally, CUDA applications tend to process a massive amount of data from the global memory within a short period of time, In Chapter 5, Memory Architecture and Data Locality, 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.[1]

The global memory of a CUDA device is implemented with DRAM. Data bits are stored in DRAM cells that are small capacitors, in which 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 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 the "Why is DRAM So Slow?" sidebar). This is in sharp contrast to the subnanosecond clock cycle time of modern computing devices. Because this process is very slow relative to the desired data access speed (subnanosecond access per byte), modern DRAM designs use parallelism to increase their rate of data access, commonly referred to as memory access throughput.

[1] Recent CUDA devices use on-chip caches for global memory data. Such caches automatically coalesce more of the kernel access patterns and somewhat reduce the need for programmers to manually rearrange their access patterns. However, even with caches, coalescing techniques will continue to have significant effects on kernel execution performance in the foreseeable future.

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 uses a transistor to drive a line connected to the outlet gates of thousands of cells. It can take a long time for the line to be fully charged or discharged to the desired level.



A more formidable challenge is for the cell 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 that is stored in the cell. If the cell content is "1," the tiny amount of charge must raise the electrical potential of the large capacitance of the long bit line to a sufficiently high level that it can 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 and a person at the other end of the hallway to use the aroma propagated along 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. Many sensors are provided in each DRAM chip, and they all work in parallel. Each 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 these bursts, the DRAMs can supply the data at a much higher rate than would be the case if a truly random sequence of locations were accessed.

Recognizing the burst organization of modern DRAMs, current CUDA devices employ a technique that allows programmers to achieve high global memory

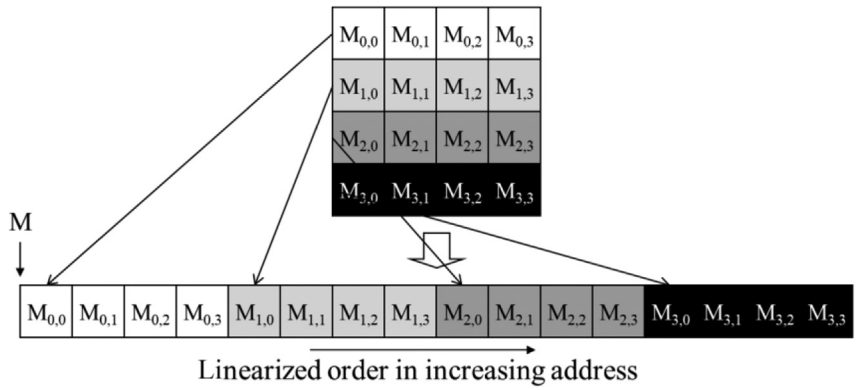


FIGURE 6.1

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

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. In other words, the most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations. In this case, the hardware combines, or *coalesces*, all these accesses into a consolidated access to consecutive DRAM locations. 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 single request for consecutive locations when accessing the DRAM.[2] Such coalesced access allows the DRAM to deliver data as a burst.[3]

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

[2] Different CUDA devices may also impose alignment requirements on the global memory address X. For example, in some CUDA devices, X is required to be aligned to 16-word (i.e., 64-byte) boundaries. That is, the lower six bits of X should all be 0 bits. Such alignment requirements have been relaxed in recent CUDA devices, owing to the presence of second-level caches.

[3] Note that modern CPUs also recognize the DRAM burst organization in their cache memory design. A CPU cache line typically maps to one or more 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 that we present in this chapter can be adapted to help CPU programs to achieve high performance.

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 they appear to be consecutive in the two-dimensional matrix, are placed four locations apart in the linearly addressed memory.

Let's assume that the multidimensional array in Fig. 6.1 is a matrix that 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 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 `M` can be coalesced. The index of the array `M` 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 `M`.

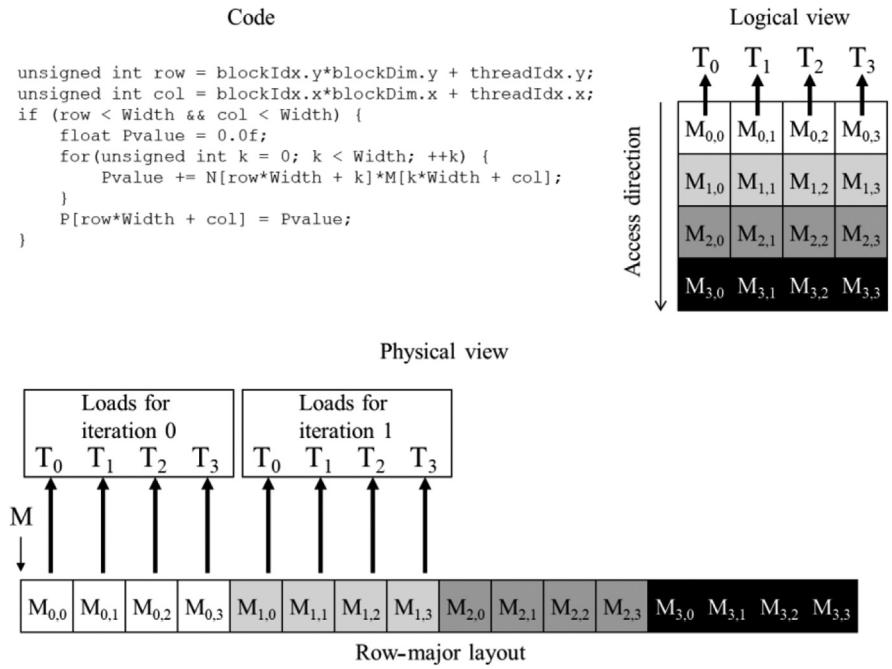


FIGURE 6.2

A coalesced access pattern.

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 that 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 that 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 that is formed by the threads during this process is a favorable one that can be coalesced. Indeed, in all the kernels that we have implemented 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 multiplying by 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 by switching the roles of the row and column indices. In C this is equivalent to viewing the transposed matrix as a column-major layout of the original matrix. Regardless of the reason, let's observe the memory access pattern that is 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 access a column of matrix M. One can tell by inspecting the code that the accesses to `M` are not favorable for coalescing. The index of the array `M` 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 `M`, `col` is multiplied by `Width`, which means that consecutive threads will access elements of `M` 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 that are also not adjacent in memory. For a realistic matrix there are typically hundreds or even thousands of elements in each dimension. The M elements that are 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

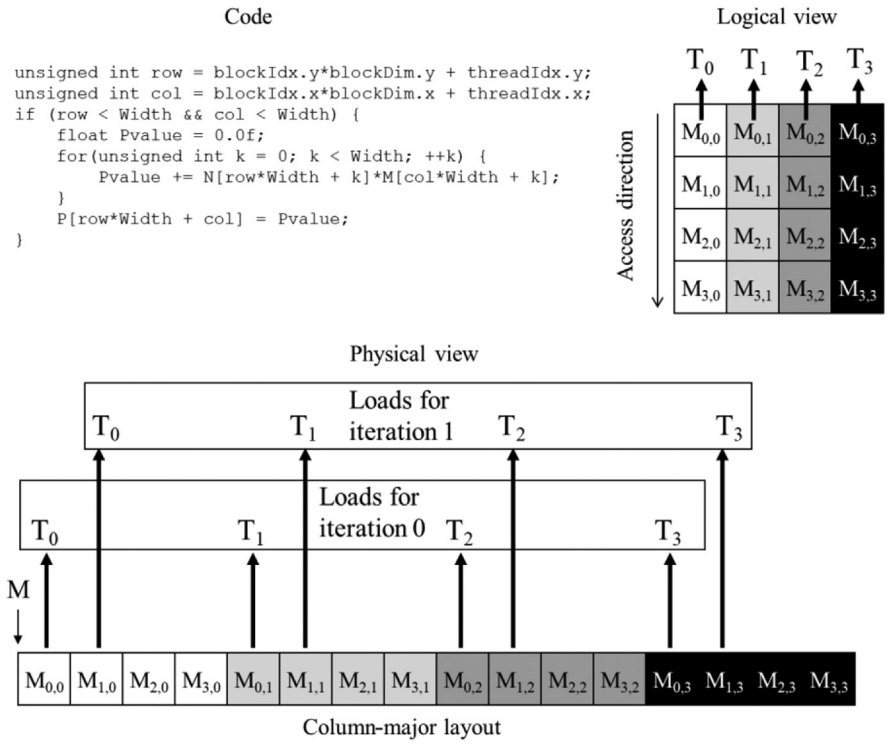


FIGURE 6.3

An uncoalesced access pattern.

how threads are mapped to the data; another strategy is to rearrange the layout of the data itself. We will discuss these strategies in Section 6.4 and see examples throughout this book of how they can be applied. 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 provides faster access latency. We will also see example optimizations that use this strategy throughout this book, including an optimization that we will apply now to matrix-matrix multiplication in which 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, A is an input matrix that is stored in row-major layout in global memory, and B is an input matrix that is stored in column-major layout in global memory. They are multiplied to produce an output matrix C that 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.

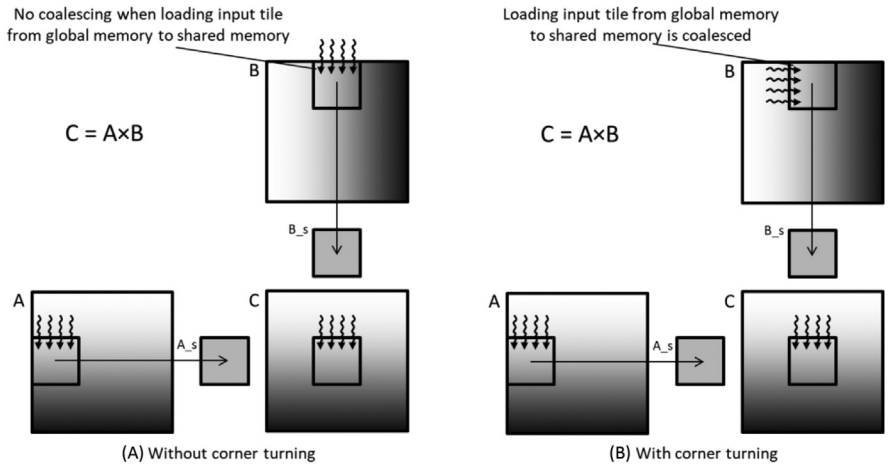


FIGURE 6.4

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

The access to the input tile in matrix A is similar to that in Chapter 5, Memory Architecture and Data Locality. 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 those 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 A that are adjacent in memory according to the row-major layout.

On the other hand, the access to the input tile in matrix B needs to be different from that in Chapter 5, Memory Architecture and Data Locality. Fig. 6.4(A) shows what the access pattern would be like if we used the same arrangement as in Chapter 5, Memory Architecture and Data Locality. Even though the four threads are logically loading the four consecutive elements at the top edge of the input tile, the elements that are loaded by consecutive threads are far away from each other in the memory because of the column-major layout of the B elements. In other words, consecutive threads that are responsible for consecutive elements in the same row of the output tile load nonconsecutive locations in memory, which results in uncoalesced memory accesses.

This problem can be solved by assigning the four consecutive threads 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 B input tile. Since B 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 B 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.

The main advantage of memory coalescing is that it reduces global memory traffic by combining multiple memory accesses into a single access. Accesses can be combined when they take place at the same time and access adjacent memory locations. Traffic congestion does not arise only 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 trying to travel on 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 when there is traffic congestion.

Most solutions for reducing traffic congestion involve the reduction of the number 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, in which the members of a group of commuters take turns driving the group to work in one vehicle. Governments usually need to 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 plate numbers 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

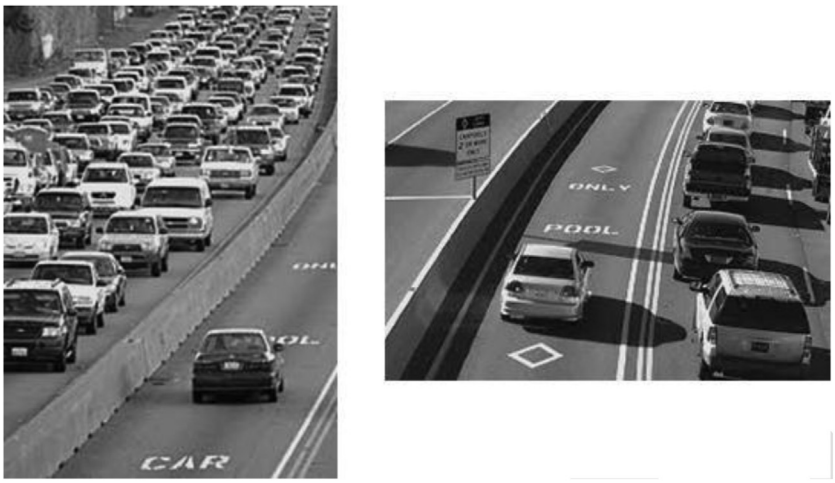


FIGURE 6.5

Reducing traffic congestion in highway systems.

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 schedules 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 not the case for the schedules shown in the bottom half of Fig. 6.6. Worker A and Worker B have very different schedules 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, and returns home for dinner at 6:00 pm. The schedules are so wildly different that these two workers cannot possibly 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 bandwidth 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. However, this requires the threads to have similar execution schedules so that their data accesses can be combined into one. Threads in the same warp are

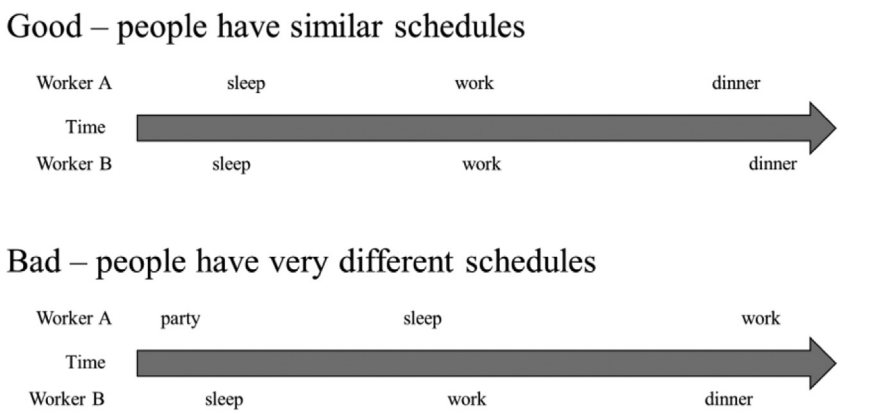


FIGURE 6.6

Carpooling requires synchronization among people.

perfect candidates because they all execute a load instruction simultaneously by virtue of SIMD execution.

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 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 a large number of banks is connected to each channel.

The data transfer bandwidth of a bus is defined by its width and clock frequency. Modern *double data rate* (DDR) busses 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 DDR bus with a clock frequency of 1 GHz has a bandwidth of 8B*2*1 GHz=16GB/s. This seems to be a large number but is often too small for modern CPUs and GPUs. A modern CPU might require a memory bandwidth of at least 32 GB/s, whereas a modern GPU might require 256 GB/s. For this example the CPU would require 2 channels, and the GPU would require 16 channels.

For each channel, the number of banks that is 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).

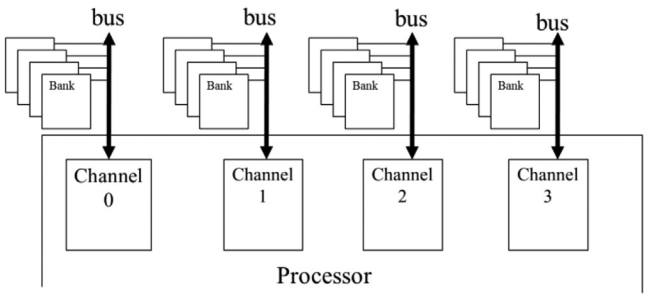


FIGURE 6.7

Channels and banks in DRAM systems.

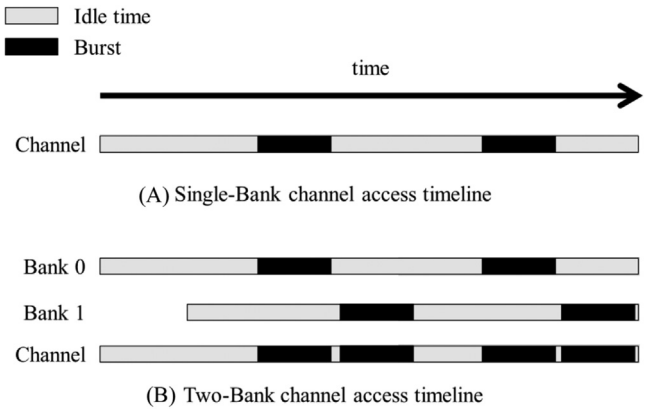


FIGURE 6.8

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

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 gray section at the left end of the time frame. Once the sensing amplifier has completed 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. The second memory read access will incur a similar long access latency (the gray section between the dark sections of the time frame) before its burst data can be transferred (the right dark section).

In reality, the access latency (the gray sections) is much longer than the data transfer time (the 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 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 to access its cell array. When the access in bank 0 is complete, it transfers the burst data (the leftmost dark section of the

time frame). Once bank 0 completes its data transfer, bank 1 can transfer its burst data (the 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 the probability of multiple simultaneous accesses targeting the same bank, a phenomenon called *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 size that is required.

There is an important connection between the parallel execution of threads and the parallel organization of the DRAM system. To achieve the memory access bandwidth specified for device, there must be a sufficient number of threads making simultaneous memory accesses. This observation reflects another benefit of maximizing occupancy. Recall that in Chapter 4, Compute Architecture and Scheduling, we saw that maximizing occupancy ensures that there are enough threads resident on the streaming multiprocessors (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 enough memory access requests are 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.

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 (8 bytes). The distribution is done by hardware design. The addressing of the channels and backs are such that the first 8 bytes of the array (M[0] and M[1]) are stored in bank 0 of channel 0, the next 8 bytes (M[2] and M[3]) in bank 0 of channel 1, the next 8 bytes (M[4] and M[5]) in bank 0 of channel 2, and the next 8 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 8 bytes (M[8] and M[9]). Thus elements M[10] and M[11] will be in bank 1 of channel 1, M[12] and M[13] will be in bank 1 of channel 2, and M[14] and M[15] will be 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, if there are more elements, M[16] and M[17] will be stored in bank 0 of channel 0, M[18] and M[19] will be stored in bank 0 of channel 1, and so on.

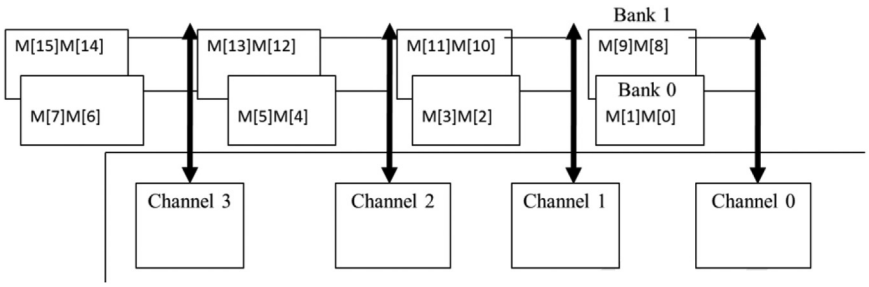


FIGURE 6.9

Distributing array elements into channels and banks.

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. Thus we assign only 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.

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 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 will be loading their first tile. The M elements that are 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 will be made to the two banks in channel 0 as well as the two banks in channel 2. These four 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 $\text{Block}_{0,0}$ and $\text{Block}_{0,1}$ will load the same M elements. Most modern devices are equipped with caches that will combine these accesses into one as long as the execution timing 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 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

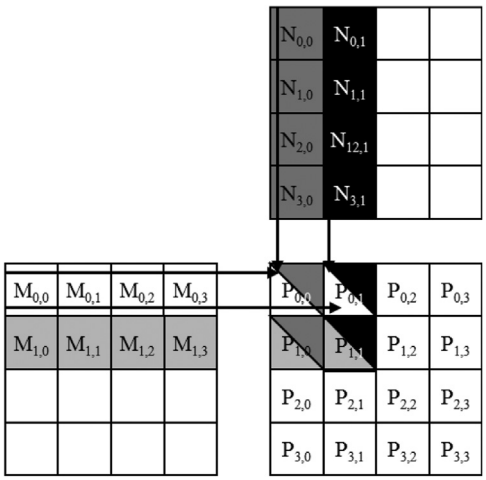


FIGURE 6.10

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

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[19], 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.

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, that is, 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 of 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.

6.3 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, each thread was assigned one output element. In the RGB-to-grayscale conversion and the image blur kernels, each thread was assigned one pixel in the output image. In the matrix multiplication kernels, 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, Compute Architecture and Scheduling. 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 the thread blocks one after the other.

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

We demonstrate the thread coarsening optimization using the tiled matrix multiplication example from Chapter 5, Memory Architecture and Data Locality. Fig. 6.12 depicts the memory access pattern of computing two horizontally adjacent output tiles of the output matrix `P`. For each of these output tiles, we observe that different input tiles of the matrix `N` need to be loaded. However, the same input tiles of the matrix `M` are loaded for both the output tiles.

In the tiled implementation in Chapter 5, Memory Architecture and Data Locality, each output tile is processed by a different thread block. Because the shared memory contents cannot be shared across blocks, each block must load its own copy of the input tiles of matrix `M`. Although having different thread blocks load the same input tile is redundant, it is a price that we pay to be able to process the two output tiles in parallel using different blocks. If these thread blocks run in

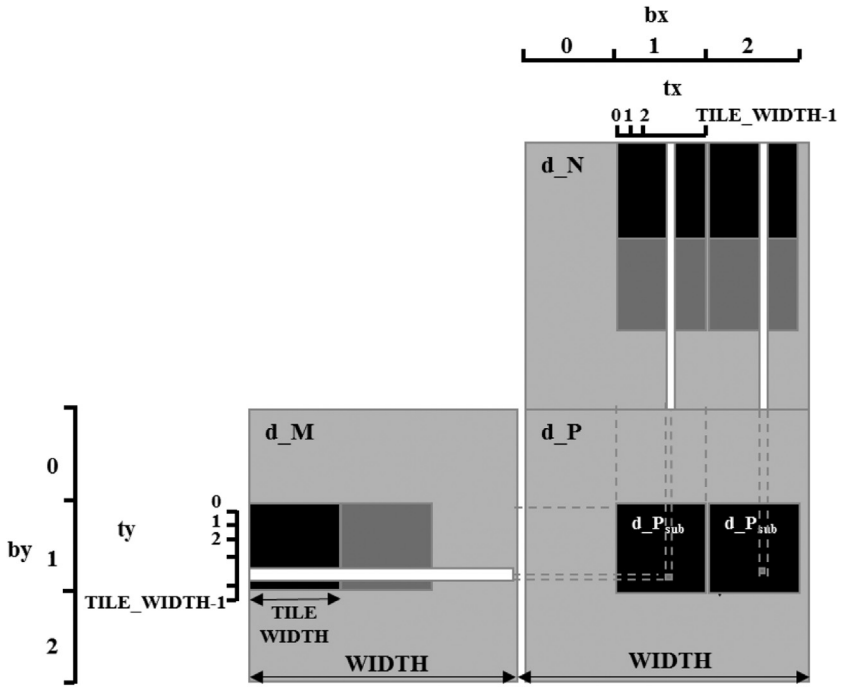


FIGURE 6.12

Thread coarsening for tiled matrix multiplication.

parallel, this price may be worth paying. On the other hand, if these thread blocks are serialized by the hardware, the price is paid in vain. In the latter case, it is better for the programmer to have a single thread block process the two output tiles, whereby each thread in the block processes two output elements. This way, the coarsened thread block would load the input tiles of `M` once and reuse them for multiple output tiles.

Fig. 6.13 shows how thread coarsening can be applied to the tiled matrix multiplication code from Chapter 5, Memory Architecture and Data Locality. On line 02 a constant `COARSE_FACTOR` is added to represent the *coarsening factor*, which is the number of original units of work for which each coarsened thread is going to be responsible. On line 13 the initialization of the column index is replaced with an initialization of `colStart`, which is the index of the first column for which the thread is responsible, since the thread is now responsible for multiple elements with different column indices. In calculating `colStart`, the block index `bx` is multiplied by `TILE_WIDTH*COARSE_FACTOR` instead of just `TILE_WIDTH`, since each thread block is now responsible for `TILE_WIDTH*COARSE_FACTOR` columns. On lines 16–19, multiple instances of `Pvalue` are declared and initialized, one for each element for which the

```
01    #define TILE_WIDTH    32
02    #define COARSE_FACTOR 4
03     __global__ void matrixMulKernel(float* M, float* N, float* P, int width)
{
04
05        __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
06        __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
07
08        int bx = blockIdx.x;  int by = blockIdx.y;
09        int tx = threadIdx.x; int ty = threadIdx.y;
10
11        // Identify the row and column of the P element to work on
12        int row = by*TILE_WIDTH + ty;
13        int colStart = bx*TILE_WIDTH*COARSE_FACTOR + tx;
14
15        // Initialize Pvalue for all output elements
16        float Pvalue[COARSE_FACTOR];
17        for(int c = 0; c < COARSE_FACTOR; ++c) {
18            Pvalue[c] = 0.0f;
19        }
20
21        // Loop over the M and N tiles required to compute P element
22        for(int ph = 0; ph < width/TILE_WIDTH; ++ph) {
23
24            // Collaborative loading of M tile into shared memory
25            Mds[ty][tx] = M[row*width + ph*TILE_WIDTH + tx];
26
27            for(int c = 0; c < COARSE_FACTOR; ++c) {
28
29                int col = colStart + c*TILE_WIDTH;
30
31                // Collaborative loading of N tile into shared memory
32                Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*width + col];
33                __syncthreads();
34
35                for(int k = 0; k < TILE_WIDTH; ++k) {
36                    Pvalue[c] += Mds[ty][k]*Nds[k][tx];
37                }
38                __syncthreads();
39
40            }
41
42        }
43
44        for(int c = 0; c < COARSE_FACTOR; ++c) {
45            int col = colStart + c*TILE_WIDTH;
46            P[row*width + col] = Pvalue[c];
47        }
48
49    }
```

FIGURE 6.13

Code for thread coarsening for tiled matrix multiplication.

coarsened thread is responsible. The loop on line 17 that iterates over the different units of work for which the coarsened thread is responsible is sometimes referred to as a *coarsening loop*. Inside the loop on line 22 that loops over the input tiles, only one tile of `M` is loaded in each loop iteration, as with the original code. However, for each tile of `M` that is loaded, multiple tiles of `N` are loaded and used by the coarsening loop on line 27. This loop first figures out which column of the current tile the coarsened thread is responsible

for (line 29), then loads the tile of `N` (line 32) and uses the tile to compute and update a different `Pvalue` each iteration (lines 35–37). At the end, on lines 44–47, another coarsening loop is used for each coarsened thread to update the output elements for which it is responsible.

Thread coarsening is a powerful optimization that can result in substantial performance improvement for many applications. It is an optimization that is commonly applied. However, there are several pitfalls to avoid in applying thread coarsening. First, one must be careful not to apply the optimization when it is unnecessary. Recall that thread coarsening is beneficial when there is a price paid for parallelization that can be reduced with coarsening, such as redundant loading of data, redundant work, synchronization overhead, or others. Not all computations have such a price. For example, in the vector addition kernel in Chapter 2, Heterogeneous Data Parallel Computing, no price is paid for processing different vector elements in parallel. Therefore applying thread coarsening to the vector addition kernel would not be expected to make a substantial performance difference. The same applies to the RGB-to-grayscale conversion kernel in Chapter 3, Multidimensional Grids and Data.

The second pitfall to avoid is not to apply so much coarsening 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 that is 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. 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.

The third pitfall of applying thread coarsening is to avoid increasing resource consumption to such an extent that it hurts occupancy. Depending on the kernel, thread coarsening may require using more registers per thread or more shared memory per thread block. If this is the case, programmers must be careful not to use too many registers or too much shared memory such that the occupancy is reduced. The performance penalty from reducing occupancy may be more detrimental than the performance benefit that thread coarsening may offer.

6.4 A checklist of optimizations

Throughout this first part of the book, we have covered various common optimizations that CUDA programmers apply to improve the performance of their code. We consolidate these optimizations into a single checklist, shown in Table 6.1. This checklist is not an exhaustive one, but it contains many of the universal

Table 6.1 A checklist of optimizations.

Optimization	Benefit to compute cores	Benefit to memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/ cache lines	Transfer between global memory and shared memory in a coalesced manner and performing uncoalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (fewer idle cores during SIMD execution)	–	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data and then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread to reduce the price of parallelism when it is incurred unnecessarily

optimizations that are common across different applications and that programmers should first consider. In the second and third parts of the book, we will apply the optimizations in this checklist to various parallel patterns and applications to understand how they operate in different contexts. In this section we will provide a brief review of each optimization and the strategies for applying it.

The first optimization in Table 6.1 is maximizing the occupancy of threads on SMs. This optimization was introduced in Chapter 4, Compute Architecture and Scheduling, where the importance of having many more threads than cores was emphasized as a way to have 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, Memory Architecture and Data Locality, shared memory was introduced as another resource whose usage should be carefully tuned so as not to limit occupancy. In 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.

The second optimization in Table 6.1 is using coalesced global memory accesses by ensuring that threads in the same warp access adjacent memory locations. This optimization was introduced in 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 the utilization of DRAM bursts. So far, the kernels that we have looked at in this part of the book have exhibited coalesced accesses naturally. However, we will see many examples in the second and third parts of the book in which 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 and then perform the irregular accesses on shared memory. We have already seen an example of this strategy in this chapter, namely, corner turning. We will see another example of this strategy in Chapter 12, Merge, 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 performs the binary search in shared memory. We will also see an example of this strategy in Chapter 13, Sorting, which covers the sort pattern. In this pattern, threads write out results to an array in a scattered manner, so 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.

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, Reduction and Minimizing Divergence, which covers the reduction pattern. 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 14,

Sparse Matrix Computation which covers sparse matrix computation and storage formats, particularly in discussing the ELL and JDS formats.

The third optimization in Table 6.1 is minimizing control divergence. Control divergence was introduced in Chapter 4, Compute Architecture and Scheduling, 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 that we have looked at in this part of the book have not exhibited control divergence, except for the inevitable divergence at boundary conditions. However, we will see many examples in the second and third parts of the book in which control divergence can be a significant detriment to performance.

There are multiple strategies that can be employed for minimizing 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 threads in other warps are used. We will see an example of this strategy in Chapter 10, Reduction and Minimizing Divergence, which covers the reduction pattern, and Chapter 11, Prefix Sum (Scan), which covers the scan 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 workloads. We will see an example of this in Chapter 15, Graph Traversal, which covers graph traversal, in which we will discuss the tradeoffs between vertex-centric and edge-centric parallelization schemes. Another strategy for minimizing control divergence is to rearrange how the data is laid out to ensure that threads in the same warp that process adjacent data have similar workloads. We will see an example of this strategy in Chapter 14, Sparse Matrix Computation, which covers sparse matrix computation and storage formats, particularly when discussing the JDS format.

The fourth optimization in Table 6.1 is tiling data that is reused within a block by placing it in the shared memory or registers and accessing it repetitively from there, such that it needs to be transferred between global memory and the SM only once. Tiling was introduced in Chapter 5, Memory Architecture and Data Locality, in the context of matrix multiplication, in which threads processing the same output tile collaborate to load the corresponding input tiles to the shared memory and then access these input tiles repetitively from the shared memory. We will see this optimization applied again in most of the parallel patterns in the second and third parts of the book. We will observe the challenges of applying tiling when the input and output tiles have different dimensions. This challenge arises in Chapter 7, Convolution, which covers the convolution pattern, and in Chapter 8, Stencil, which covers the stencil pattern. We will also observe that tiles of data can be stored in registers, not just shared memory. This observation is most pronounced in Chapter 8, Stencil, which covers the stencil pattern. We will additionally observe that tiling is applicable to output data that is accessed repeatedly, not just input data.

The fifth optimization in Table 6.1 is privatization. This optimization has not yet been introduced, but we mention it here for completeness. Privatization relates to the situation in which multiple threads or blocks need to update a universal

output. To avoid the overhead of updating the same data concurrently, a private copy of the data can be created and partially updated, and then a final update can be made to the universal copy from the private copy when done. We will see an example of this optimization in Chapter 9, Parallel Histogram, which covers the histogram pattern, in which multiple threads need to update the same histogram counters. We will also see an example of this optimization in Chapter 15, Graph Traversal, which covers graph traversal, in which multiple threads need to add entries to the same queue.

The sixth optimization in Table 6.1 is thread coarsening, in which multiple units of parallelism are assigned to a single thread to reduce the price of parallelism if the hardware was going to serialize the threads anyway. Thread coarsening was introduced in this chapter in the context of tiled matrix multiplication, in which the price of parallelism was loading of the same input tile redundantly by multiple thread blocks that process adjacent output tiles. In this case, assigning one thread block to process multiple adjacent output tiles enables loading an input tile once for all the output tiles. In the second and third parts of the book, we will see thread coarsening applied in different contexts with a different price of parallelism each time. In Chapter 8, Stencil, which covers the stencil pattern, thread coarsening is applied to reduce the redundant loading of input data, as in this chapter. In Chapter 9, Parallel Histogram, which covers the histogram pattern, thread coarsening helps to reduce the number of private copies that need to be committed to the universal copy in the context of the privatization optimization. In Chapter 10, Reduction and Minimizing Divergence, which covers the reduction pattern, and in Chapter 11, Prefix Sum (Scan), which covers the scan pattern, thread coarsening is applied to reduce the overhead from synchronization and control divergence. Also in Chapter 11, Prefix Sum (Scan), which covers the scan pattern, thread coarsening also helps reduce the redundant work that is performed by the parallel algorithm compared to the sequential algorithm. In Chapter 12, Merge, which covers the merge pattern, thread coarsening reduces the number of binary search operations that need to be performed to identify each thread's input segment. In Chapter 13, Sorting, which covers the sort pattern, thread coarsening helps improve memory coalescing.

Again, the checklist in Table 6.1 is not intended to be an exhaustive one, but it contains major types of the optimizations that are common across different computation patterns. These optimizations appear in multiple chapters in the second and third parts of the book. We will also see other optimizations that appear in specific chapters. For example, in Chapter 7, Convolution, which covers the convolution pattern, we will introduce the use of constant memory. In Chapter 10, Reduction and Minimizing Divergence, which covers the scan pattern, we will introduce the double-buffering optimization.

6.5 Knowing your computation's bottleneck

In deciding what optimization to apply to a specific computation, it is important first to understand what resource is limiting the performance of that computation.

The resource that limits the performance of a computation 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 that is applied does not target the bottleneck resource, there may be no benefit from the optimization. Worse yet, the optimization attempt 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 is the global memory bandwidth and the data being loaded is reused. However, if, for example, the performance is limited by occupancy and occupancy is constrained by the use of too much shared memory already, then applying shared memory tiling is likely to make things worse.

To understand what resource is limiting the performance of a computation, 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 computations (NVIDIA, Profiler). Performance bottlenecks may be hardware-specific, meaning that the same computation 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.

6.6 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 an important optimization: thread granularity coarsening. With the insights that were presented in this chapter and earlier chapters, readers should be able to reason about the performance of any kernel code that they come across. We concluded this part of the book 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.

Exercises

1. Write a matrix multiplication kernel function that corresponds to the design illustrated in Fig. 6.4.
2. For tiled matrix multiplication, 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.)

3. 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 or uncoalesced or 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

4. What is the floating point to global memory access ratio (in OP/B) of each of the following matrix-matrix multiplication kernels?
   a. The simple kernel described in Chapter 3, Multidimensional Grids and Data, without any optimizations applied.
   b. The kernel described in Chapter 5, Memory Architecture and Data Locality, with shared memory tiling applied using a tile size of 32×32.
   c. The kernel described in this chapter with shared memory tiling applied using a tile size of 32×32 and thread coarsening applied using a coarsening factor of 4.

References

NVIDIA Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide.

CHAPTER

7 Convolution

An introduction to constant memory and caching

Chapter Outline

In the next several chapters we will 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 will 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 smoothes 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 datasets such as high-definition images and videos in which there are many output elements (pixels), the amount of computation can be huge. On one hand, each output data element of convolution 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 in processing different output data elements with somewhat challenging boundary conditions. This makes convolution an important use case for sophisticated tiling methods and input data staging methods, which are the focus of this chapter.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00008-2

7.1 Background

Convolution is an array operation in which each output data element is a weighted sum of the corresponding input element and a collection of input elements that are centered on it. The weights that are 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 refer to these filter arrays as *convolution filters* to avoid confusion.

Convolution can be performed on input data of different dimensionality: one-dimensional (1D) (e.g., audio), two-dimensional (2D) (e.g., photo), three-dimensional (3D) (e.g., video), and so on. 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 $[x_0, x_1, \ldots, x_{n-1}]$ and a filter array of $2r + 1$ elements $[f_0, f_1, \ldots, f_{2r}]$ and returns an output data array y:

$$y_i = \sum_{j=-r}^{r} f_{i+j} \times x_i$$

Since the size of the filter is an odd number $(2r + 1)$, the weighted sum calculation is symmetric around the element that is being calculated. That is, the weighted sum involves r input elements on each side of the position that is 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 in which a five-element ($r = 2$) convolution filter f is applied to a seven-element input array x. We will follow the C language convention by which x and y elements are indexed from 0 to 6 and f elements are indexed from 0 to 4. Since the filter radius is 2, 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.

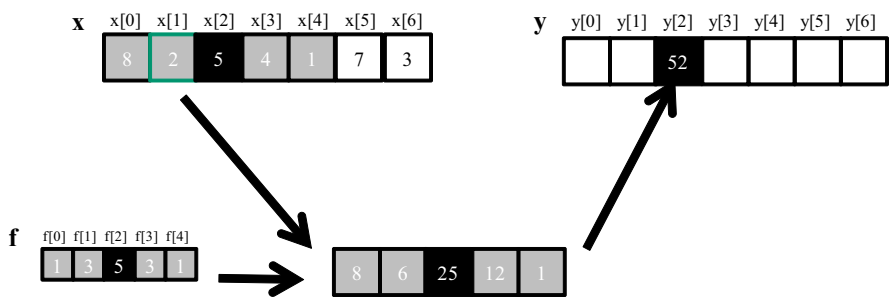


FIGURE 7.1

A 1D convolution example, inside elements.

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:

$$\begin{aligned} y[2] &= f[0]*x[0] + f[1]*x[1] + f[2]*x[2] + f[3]*x[3] + f[4]*x[4] \\ &= 1*8 + 3*2 + 5*5 + 3*4 + 1*1 \\ &= 52 \end{aligned}$$

In Fig. 7.1 the calculation for y[i] can be viewed as an inner product between the subarray of x that starts at x[I − 2] and the f array. 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] is as following inner product:

$$\begin{aligned} y[3] &= f[0]*x[1] + f[1]*x[2] + f[2]*x[3] + f[3]*x[4] + f[4]*x[5]y[3] \\ &= f[0]*x[1] + f[1]*x[2] + f[2]*x[3] + f[3]*x[4] + f[4]*x[5] \\ &= 1*2 + 3*5 + 5*4 + 3*1 + 1*7 \\ &= 47 \end{aligned}$$

Because convolution is defined in terms of neighboring elements, boundary conditions naturally arise in 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 conditions is to assign 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

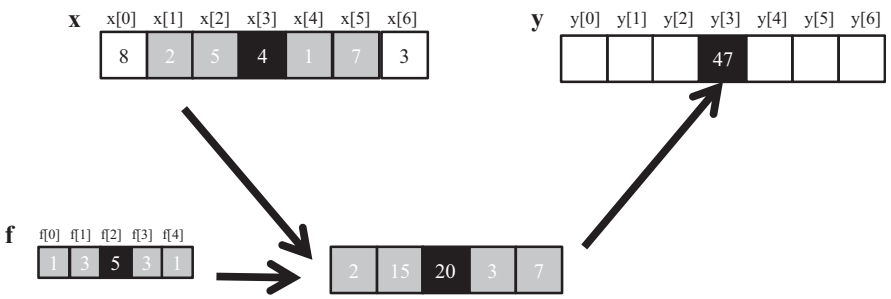


FIGURE 7.2

1D convolution, calculation of y[3].

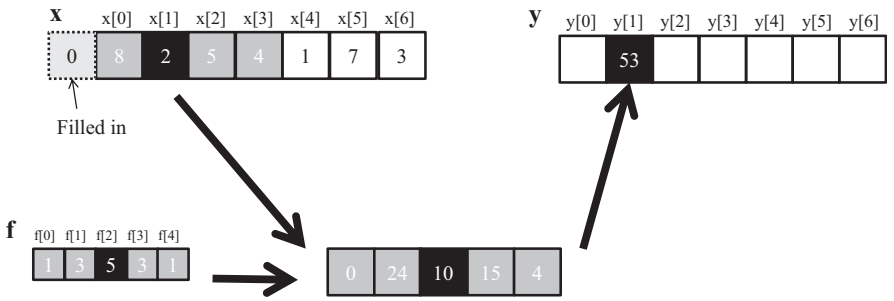


FIGURE 7.3

A 1D convolution boundary condition.

before the recording starts and after it ends. In this case, the calculation of y[1] is as follows:

$$\begin{aligned} y[1] &= f[0]*0 + f[1]*x[0] + f[2]*x[1] + f[3]*x[2] + f[4]*x[3] \\ &= 1*0 + 3*8 + 5*2 + 3*5 + 1*4 \\ &= 53 \end{aligned}$$

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] will involve two missing x elements, both of which will 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.

Also, not all applications assume that the ghost cells contain 0. For example, some applications might assume that the ghost cells contain the same value as the closest valid data element on the edge.

For image processing and computer vision, input data is typically represented as 2D 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. If we 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 follows:

$$P_{y,x} = \sum_{j=-r_y}^{r_y} \sum_{k=-r_x}^{r_x} f_{y+j,x+k} \times N_{y,x}$$

In Fig. 7.4 we use a 5 × 5 filter for simplicity; that is, $r_y = 2$ 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 subarray whose center is at the corresponding location in the input array N. We then perform pairwise multiplication between

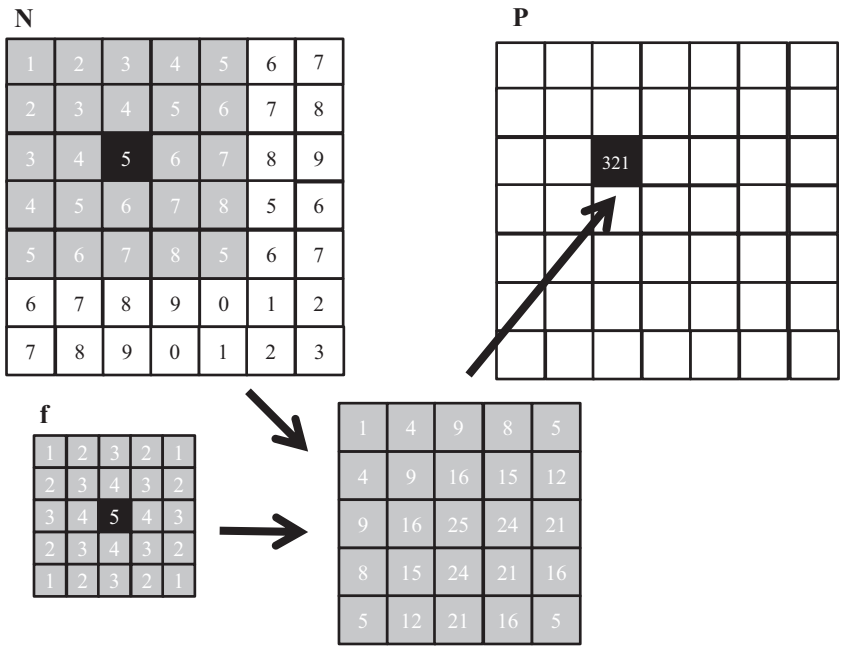


FIGURE 7.4

A 2D convolution example.

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.

The example in Fig. 7.4 shows the calculation of P2,2. For brevity, we will use *Ny,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:

$$
\begin{aligned}
P_{2,2} &= N_{0,0}*M_{0,0} + N_{0,1}*M_{0,1} + N_{0,2}*M_{0,2} + N_{0,3}*M_{0,3} + N_{0,4}*M_{0,4} \\
&\quad + N_{1,0}*M_{1,0} + N_{1,1}*M_{1,1} + N_{1,2}*M_{1,2} + N_{1,3}*M_{1,3} + N_{1,4}*M_{1,4} \\
&\quad + N_{2,0}*M_{2,0} + N_{2,1}*M_{1,1} + N_{2,2}*M_{2,2} + N_{2,3}*M_{2,3} + N_{2,4}*M_{2,4} \\
&\quad + N_{3,0}*M_{3,0} + N_{3,1}*M_{3,1} + N_{3,2}*M_{3,2} + N_{3,3}*M_{3,3} + N_{3,4}*M_{3,4} \\
&\quad + N_{4,0}*M_{4,0} + N_{4,1}*M_{4,1} + N_{4,2}*M_{4,2} + N_{4,3}*M_{4,3} + N_{4,4}*M_{4,4} \\
&= 1*1 + 2*2 + 3*3 + 4*2 + 5*1 \\
&\quad + 2*2 + 3*3 + 4*4 + 5*3 + 6*2 \\
&\quad + 3*3 + 4*4 + 5*5 + 6*4 + 7*3 \\
&\quad + 4*2 + 5*3 + 6*4 + 7*3 + 8*2 \\
&\quad + 5*1 + 6*2 + 7*3 + 8*2 + 5*1 \\
&= 1 + 4 + 9 + 8 + 5 \\
&\quad + 4 + 9 + 16 + 15 + 12 \\
&\quad + 9 + 16 + 25 + 24 + 21 \\
&\quad + 8 + 15 + 24 + 21 + 16 \\
&\quad + 5 + 12 + 21 + 16 + 5 \\
&= 321
\end{aligned}
$$

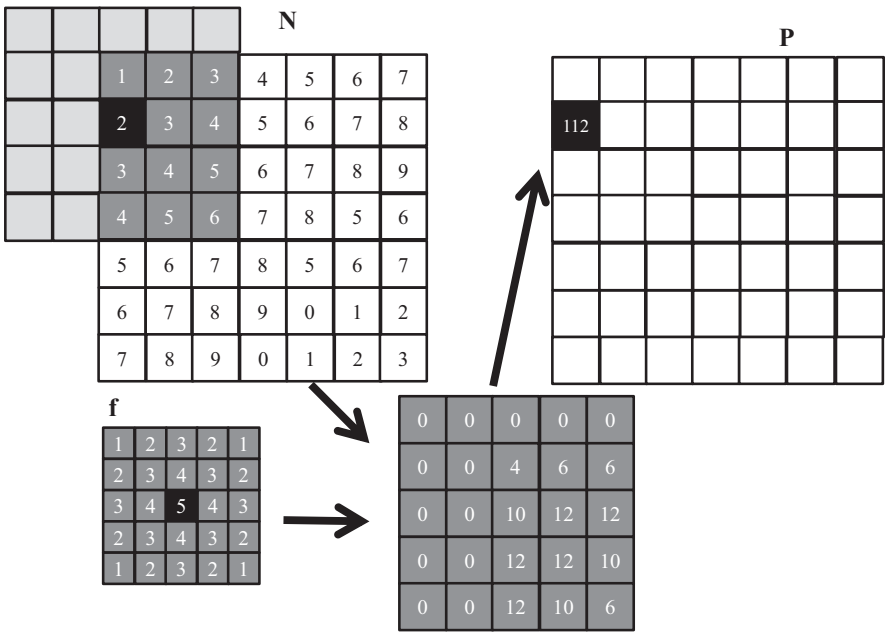


FIGURE 7.5

A 2D convolution boundary condition.

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 subarray of N. As 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 algorithm

The fact that the calculation of all output elements can be done in parallel in a convolution makes convolution an ideal use case for parallel computing. Based on our experience in matrix multiplication, we can quickly write a simple parallel convolution kernel. We will show code examples for 2D convolution, and the

reader is encouraged to adapt these code examples to 1D and 3D as exercises. Also, for simplicity we will assume square filters.

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 the input array, *N*; a pointer to the filter, *F*; a pointer to the 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 setup:

```
__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 calculate one output element. Each block, with up to 1024 threads in a block, can calculate up to 1024 output elements. Fig. 7.6 shows a toy example in which 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—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 the thread's *x* and *y* indices.

The reader should recognize that the parallelization arrangement in Fig. 7.6 is the same as the `ColorToGrayScaleConversion` example in Chapter 3, Multidimensional

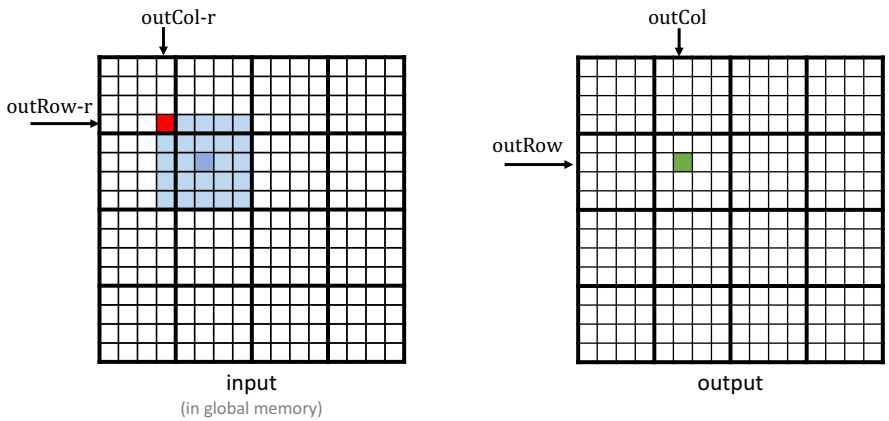


FIGURE 7.6

Parallelization and thread organization for 2D convolution.

Grids and Data. Therefore we can use the statements in lines 02 and 03 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. 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 green square in Fig. 7.6.

Once we have determined the output element indices for each thread, we can identify the input *N* elements that are needed for calculating the output element. As illustrated in Fig. 7.6, the calculation of P[5][5] (green square) by $thread_{1,1}$ of $block_{1,1}$ will use the input elements whose *x* indices range from `outCol - r=3 to outCol + r=7` and whose y indices range from `outRow - r=3 to outRow + r=7`. For all threads, `outCol - r and outRow - r` define the upper-left corner (heavily shaded square) of the patch of input elements (lightly shaded area) 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 will accumulate all intermediate results to save DRAM bandwidth. The if-statement in the inner for-loop tests whether any of the input *N* elements that are 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 on the kernel in Fig. 7.7. First, there will be control flow divergence. The threads that calculate the output elements near the four edges of the P array will need to handle ghost cells. As we showed in Section 7.1, each of these threads will encounter a different number of ghost cells. Therefore they will all be somewhat different decisions in the if-statement (line 09). The thread that calculates P[0][0] will skip the multiply-accumulate statement most of the time, whereas the one that calculates P[0][1] will skip fewer times, and so on. The cost of control divergence will depend

```
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.

on the value of width and height of the input array and the radius of the filter. For large input arrays and small filters, control divergence occurs only in computing a small portion of the output elements, which will keep the effect of control divergence small. Since convolution is often applied to large images, we expect the effect of control divergence to range from modest to insignificant.

A more serious problem is memory bandwidth. The ratio of floating-point arithmetic calculation to global memory accesses is only about 0.25 OP/B (2 operations for every 8 bytes loaded on line 10). As we saw in the matrix multiplication example, this simple kernel can be expected to run only at a tiny fraction of the peak performance. We will discuss two key techniques for reducing the number of global memory accesses in the next two sections.

7.3 Constant memory and caching

There are three interesting properties in 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 F elements in the same order, starting from `F[0][0]` and moving by one element at a time through the iterations of the doubly nested for-loop in Fig. 7.7. These three properties make the filter an excellent candidate for constant memory and caching (Fig. 7.8).

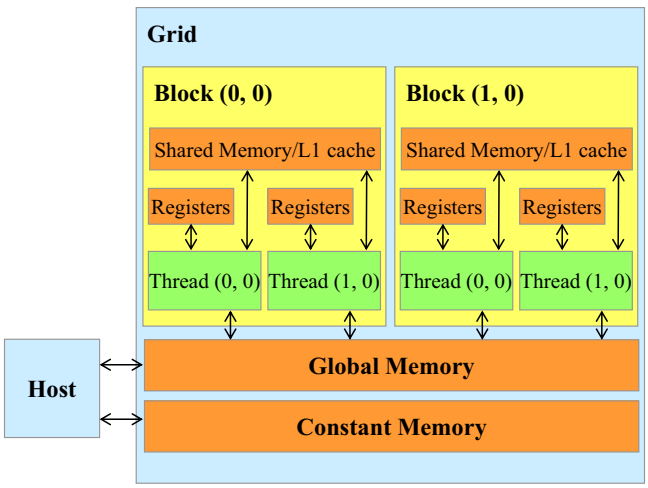


FIGURE 7.8

A review of the CUDA memory model.

As we discussed in Chapter 5, Memory Architecture and Data Locality (Table 5.1), the CUDA C allows programmers to declare variables to reside in the constant memory. Like global memory variables, constant memory variables are 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 an F array in constant memory, the host code declares it a global variable as follows:

```
#define FILTER_RADIUS 2
__constant__ float F[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
```

Note that this is a global variable declaration and 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 mask in a filter `F_h` array in the host memory with `(2*FILTER_RADIUS+1)2` elements. The contents of the `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));
```

Note that this 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 `cudaMemcpyToSymble()` 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.[1]

Kernel functions access constant memory variables as global variables. Therefore 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 that is passed in as a parameter. It is now

[1]The function can take two more arguments, namely, `offset` and `kind`, but these are seldom used and are 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.

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 constant memory usage, we need to first understand more about modern processor memory and cache hierarchies.

As we discussed in Chapter 6, Performance Considerations, 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, in using caches, the program simply accesses the original global memory variables. The processor hardware will automatically retain the most recently or frequently used variables in the cache and remember their original global memory address. When one of the retained variables is used later, the hardware will detect from their addresses that a copy of the variable is available in cache. The value of the variable will then be 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

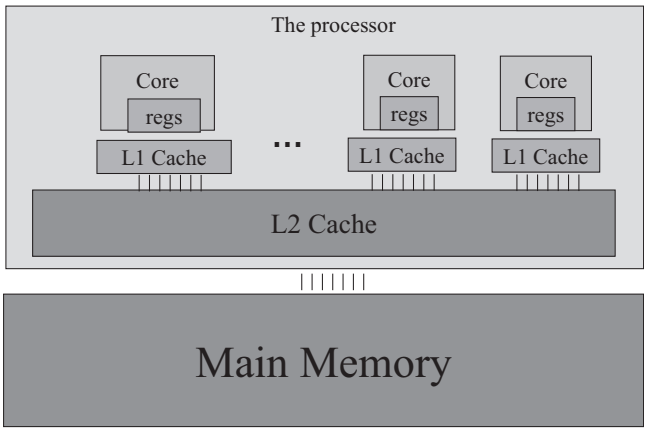


FIGURE 7.10

A simplified view of the cache hierarchy of modern processors.

level, L1 or level 1, is the cache that is directly attached to a processor core, as shown in Fig. 7.10. It runs at a speed close to that of the processor in both latency and bandwidth. However, an L1 cache is small, typically between 16 and 64 KB in capacity. L2 caches are larger, in the range of a few hundred kilobytes to a small number of MBs but can take tens of cycles to access. They are typically shared among multiple processor cores, or streaming multiprocessors (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 megabytes 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-throughput 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, 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. As a result, 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 the use of constant memory and caching, we have effectively

doubled the ratio of floating-point arithmetic to memory access to around 0.5 OP/B (2 operations for every 4 bytes loaded on line 10).

As it turns out, the accesses to the input N array elements can also benefit from caching. We will come back to this point in Section 7.5.

7.4 Tiled convolution with halo cells

We can address the memory bandwidth bottleneck of convolution with a tiled convolution algorithm. Recall that in a tiled algorithm, threads collaborate to load input elements into an on-chip memory for subsequent use of these elements. We will first establish the definitions of input and output tiles, since these definitions are important for understanding the design of the algorithm. We will 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 16 blocks of 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 many more to achieve good occupancy and data reuse. From this point on, we will assume that the F elements are in the constant memory.

We define an input tile as the collection of input N elements that are needed to calculate the P elements in an output tile. Fig. 7.11 shows the input tile (the shaded patch on the left side) that corresponds to an output tile (the shaded patch on the right side). 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 that are needed for calculating the P elements at the edges of the output tile. This extension can make the input tiles significantly larger than output tiles.

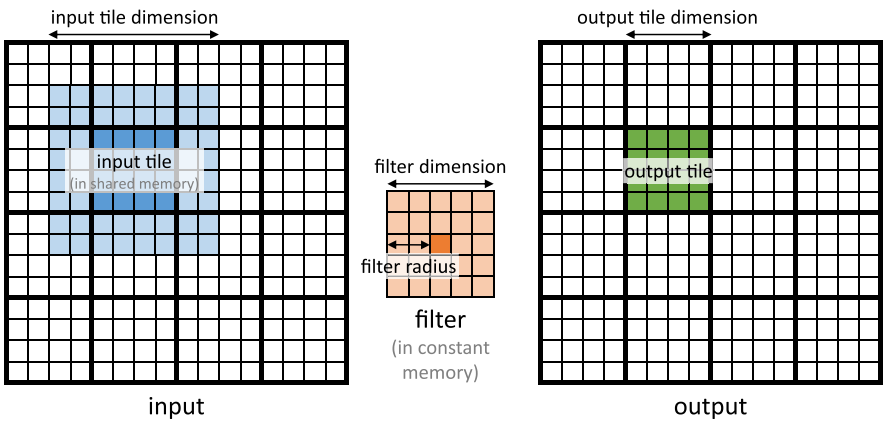


FIGURE 7.11

Input tile versus output tile in a 2D convolution.

In this toy example, each output tile consists of $4^2 = 16$ P-elements, whereas each input tile consists of $(4 + 4)^2 = 8^2 = 64$ elements. In this case, the input tiles are $4\times$ 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 closer to 1.0. For example, if the output size is $16 \times 16 = 256$, with the same 5×5 filter, the input tile size would be $(16 + 4)^2 = 400$. The ratio between the input tile size and the output size would be 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 a class of tiled convolution algorithms in which all threads in a block first collaboratively load the input tile into the shared memory before they calculate the elements of the output tile by accessing the input elements from the shared memory. This should sound familiar to the reader; the strategy resembles that of the tiled matrix multiplication algorithms that were discussed in Chapter 5, Memory Architecture and Data Locality. The main difference is that the tiled matrix multiplication algorithms in Chapter 5, Memory Architecture and Data Locality, assume 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, which can reduce the efficiency of execution resource utilization. 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 will present the design of a kernel based on the first thread organization and leave the second organization as an exercise.

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 or output element that it is responsible for loading or 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 in line 10 are used by each thread to check whether the input tile element that 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

```
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.

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_ds` array, each thread can calculate their output P element value using the `N_ds` elements. Keep in mind that the output tile is smaller than the input tile and that the blocks are of the same size as the input tiles, so only a subset of the threads in each block will be used to calculate the output tile elements. There are multiple ways in which we can select the threads for this calculation. We use a design that 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 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 overlaid on top of each other. With our design, we deactivate `FILTER_RADIUS`=1 exterior layer of threads. The heavy-line box at the center of the left side of Fig. 7.13 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) will calculate output element (tx - `FILTER_RADIUS`, ty

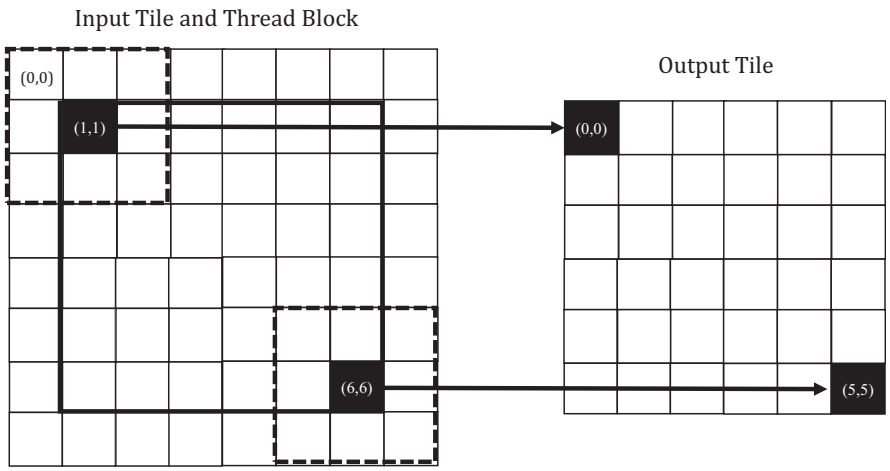


FIGURE 7.13

A small example that illustrates the thread organization for using the input tile elements in the shared memory to calculate the output tile elements.

- 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) 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 fRow-fCol loop nest on lines 24–28 of Fig. 7.12 iterates through the patch and generates the output element. Thread (1,1) in the block will iterate through the patch whose upper-left corner is N_s[0][0], whereas thread (5,5) will iterate 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, respectively, of the beginning of the output tile assigned to the block. As we discussed earlier, threadIdx.x-r and threadIdx.y-r 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-to-global memory access ratio so that the achieved performance is not limited or less limited by the DRAM bandwidth. Recall from

Section 7.4 that the arithmetic-to-global memory access ratio of the kernel in Fig. 7.9 is 0.5 OP/B. Let us now derive this ratio 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 mask sizes will be insignificant. Therefore we will ignore the effect of ghost cells when we calculate the arithmetic-to-global memory access ratio in tiled convolution kernels and consider only the internal thread blocks whose halo cells are not ghost cells.

We now calculate the arithmetic-to-global memory access ratio for the tiled kernel in Fig. 7.12. Every thread that is 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 `OUT_TILE_DIM^2*(2*FILTER_RADIUS + 1)^2*2` arithmetic operations. As for the global memory accesses, all the global memory accesses have been shifted to the code that loads the N elements into the shared memory. Each thread that is assigned to an input tile element loads one 4-byte input value. Therefore `IN_TILE_DIM^2*4=(OUT_TILE_DIM+2*FILTER_RADIUS)^2*4` bytes are loaded by each internal block. Therefore the arithmetic-to-global memory access ratio for the tiled kernel is

$$\frac{\text{OUT\_TILE\_DIM}^2*(2*\text{FILTER\_RADIUS}+1)^2*2}{(\text{OUT\_TILE\_DIM}+2*\text{FILTER\_RADIUS})^2*4}$$

For our example with a 5 × 5 filter and 32 × 32 input tiles (28 × 28 output tiles), the ratio is $\frac{28^2 \times 5^2*2}{32^2*4} = 9.57 OP/B$. An input tile size of 32 × 32 is the largest that is achievable on current GPUs. However, we can perform an asymptotic analysis on the tile size to get an upper bound on the arithmetic-to-global memory access ratio that is achievable for this computation. If `OUT_TILE_DIM` is much larger than `FILTER_RADIUS`, we can consider `OUT_TILE_DIM+2*FILTER_RADIUS` to be approximately `OUT_TILE_DIM`. This simplifies the expression to `(2*FILTER_RADIUS+1)^2*2/4`. This should be quite an intuitive result. In the original algorithm, each N element is redundantly loaded by approximately `(2*FILTER_RADIUS+1)^2` threads, each of which performs two arithmetic operations with it. Thus if the tile size is infinitely large and each 4-byte element is loaded only into the shared memory once, the ratio should be `(2*FILTER_RADIUS+1)^2*2/4`.

Fig. 7.14 shows how the arithmetic-to-global memory access ratio of the tiled convolution kernel for different filter sizes varies with tile dimension, including an asymptotic bound. The bound on the ratio with a 5 × 5 filter is 12.5 OP/B. However, the ratio that is actually achievable with the 32 × 32 limit on thread block size is 9.57 OP/B. For a larger filter, such as 9 × 9 in the bottom row of Fig. 7.14, the bound on the ratio is 40.5 OP/B. However, the ratio that is actually achievable with the 32 × 32 limit on thread block size is 22.78 OP/B. Therefore we observe that a larger filter size has a higher ratio because each input element

IN_TILE_DIM		8	16	32	Bound
5x5 filter (FILTER_RADIUS = 2)	OUT_TILE_DIM	4	12	28	-
	Ratio	3.13	7.03	9.57	12.5
9x9 filter (FILTER_RADIUS = 4)	OUT_TILE_DIM	-	8	24	-
	Ratio	-	10.13	22.78	40.5

FIGURE 7.14

Arithmetic-to-global memory access ratio as a function of tile size and filter size for a 2D tiled convolution.

is used by more threads. However, the larger filter size also has a higher disparity between the bound and the ratio that is actually achieved because of the larger number of halo elements that force a smaller output tile.

The reader should always be careful when using small block and tile sizes. They may result in significantly less reduction in memory accesses than expected. For example, in Figs. 7.14, 7.8 $\times$ 8 blocks (input tiles) result in only a ratio of 3.13 OP/B for 5 $\times$ 5 filters. In practice, smaller tile sizes are often used because of an insufficient amount of on-chip memory, especially in 3D convolution, in which the amount of on-chip memory that is needed grows quickly with the dimension of the tile.

7.5 Tiled convolution using caches for halo cells

In Fig. 7.12, much of the complexity of the code has to do with the fact that the input tiles and blocks are larger than the output tiles because of the loading of 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 its halo cells, they are already in L2 cache because of 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_ds`. We now present a tiled convolution algorithm that uses the same dimension for input and output tiles and loads only the internal elements of each 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_ds` array needs to hold only 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 x and y dimensions (line 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.

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 x and y coordinates as its assigned output element (lines 4−5 and 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, there is no danger of loading ghost cells. Thus the condition needs to check only for the usual boundary condition that a tile may extend beyond the valid range of the input data.

However, the body of the loop that calculates P elements becomes more complex. It needs to add conditions to check for use of both halo cells and ghost cells. The handling of halo cells is done with the conditions in lines 17−20, which tests whether 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 in lines 24−27

check whether 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 2. 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.

7.6 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 solvers as a special case of convolution; this will be the subject of Chapter 8, Stencil. For another example, one can also view the calculation of grid point force or potential value as a special case of convolution, which will be presented in Chapter 17, Iterative Magnetic Resonance Imaging Reconstruction. We will also apply much of what we learned in this chapter on convolutional neural networks in Chapter 16, Deep Learning.

We presented a basic parallel convolution algorithm whose implementation will be 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 algorithm 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 algorithm 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-to-global memory access ratio. The analysis is an important skill and will be useful in understanding the benefit of tiling for other patterns. Through the analysis we can learn about the limitation of small tile sizes, which is especially pronounced for large filters and 3D convolutions.

Although we have shown kernel examples for only 1D and 2D convolutions, the techniques are directly applicable to 3D convolutions as well. In general, the index calculation for the input and output arrays are more complex, owing to higher dimensionality. Also, one will have more loop nesting for each thread, since multiple dimensions need to be traversed in loading tiles and/or calculating output values. We encourage the reader to complete these higher-dimension kernels as homework exercises.

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. [− 1/2 0 1/2]
   e. [1/3 1/3 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 mask 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.

CHAPTER

Stencil 8

Chapter Outline

Stencils are foundational to numerical methods for solving partial differential equations in application domains such as fluid dynamics, heat conductance, combustion, weather forecasting, climate simulation, and electromagnetics. The data that is processed by stencil-based algorithms consists of discretized quantities of physical significance, such as mass, velocity, force, acceleration, temperature, electrical field, and energy, whose relationships 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 both stencils and convolution 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, differentiable functions within the domain of interest. The data elements and the weight coefficients that are used for the elements in the stencil neighborhood are governed by the differential equations that are being solved. Some stencil patterns are amenable to optimizations that are not applicable to convolution. In the solvers in which 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 questions, the data that is processed by stencils tends to be high-precision floating data that consumes more on-chip memory for tiling techniques. Because of these differences, stencils tend to motivate different optimizations than convolution does.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00017-3

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. 8.1A shows the sine function $y = \sin(x)$ for $0 \leq x \leq \pi$. Fig. 8.1B shows the design of a one-dimensional (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 one dimension, rectangles in two dimension, bricks in three dimensions). As we will see later, with structured grids, derivatives of variables can be conveniently expressed as finite differences. Therefore structured grids are used mainly in finite-difference methods. Unstructured grids are more complex and are used in finite-element and finite-volume methods. For simplicity we will use only regular grids and thus finite-difference methods in this book.

Fig. 8.1C shows the resulting discrete representation in which 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 $i\frac{\pi}{6}$, 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 $2\frac{\pi}{6}$.

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 computation when we solve a partial differential equation.

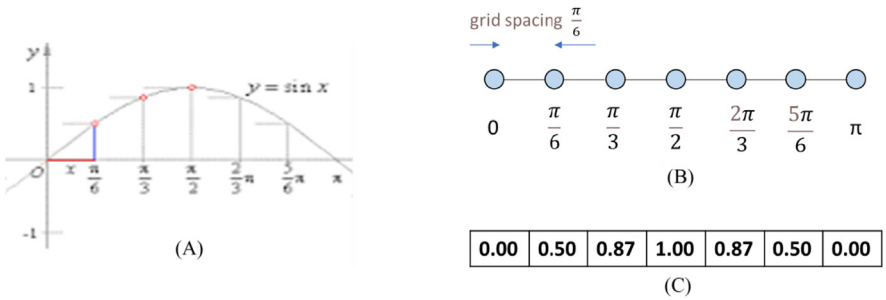


FIGURE 8.1

(A) Sine as a continuous, 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 $0 \leq x \leq \pi$.

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-precisions (32-bit), and half-precision (16-bit) representations today. Among the three, 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 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.

Let us discuss the definition of 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 by using finite differences between the values of the function at the point and its neighbors. Since partial differential equations express the relationships 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 $f(x)$ 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 by the difference of the function values at 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 be able to result in a reasonably close approximation.

Since the grid spacing is h, the current estimated $f(x-h)$, $f(x)$, and $f(x+h)$ values are in F[i − 1], F[i], and F[i + 1], respectively, where $x = i^{*}h$. Therefore we can calculate the derivative values of $f(x)$ at each grid point into an output array FD:

$$FD[i] = \frac{F[i+1] - F[i-1]}{2*h}$$

for all the grid points. This expression can be rewritten as

$$FD[i] = \frac{-1}{2*h} * F[i-1] + \frac{1}{2*h} * F[i+1]$$

That is, the calculation of the estimated function derivative value at grid point involves the current estimated function values at grid points [i − 1, i, i + 1] with coefficients $\left[\frac{-1}{2h}, 0, \frac{1}{2h}\right]$, which defines a 1D three-point stencil, as shown in Fig. 8.2A. 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 [i − 2, i − 1, i, i + 1, i + 2], which is a 1D five-point stencil, as shown in Fig. 8.2B. 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. 8.2C shows a 1D seven-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 are of order 1, 2, and 3, respectively.

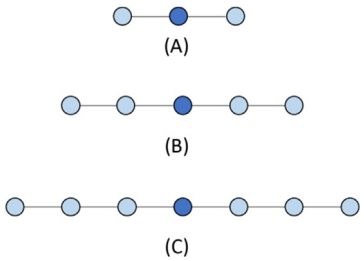


FIGURE 8.2

One-dimensional stencil examples. (A) Three-point (order 1) stencil. (B) Five-point (order 2) stencil. (C) Seven-point (order 3) stencil.

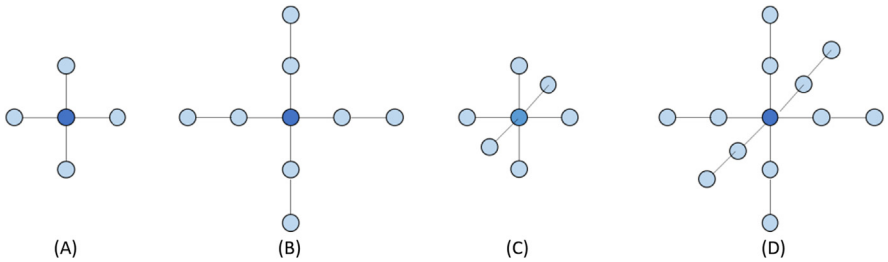


FIGURE 8.3

(A) Two-dimensional five-point stencil (order 1). (B) Two-dimensional nine-point stencil (order 2). (C) Three-dimensional seven-point stencil (order 1). (D) Three-dimensional 13-point stencil (order 2).

It should be obvious that solving a partial differential equation of two variables would require the function values to be discretized into a two-dimensional (2D) grid, and we will 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, for example, $\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y}$, but not $\frac{\partial f(x,y)}{\partial x \partial y}$, we can use 2D stencils whose selected grid points are all along the x axis and y axis. For example, for a partial differential equation involving only first derives 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 five-point stencil in Fig. 8.3A. If the equation involves up to second-order derivatives only by x or by y, we will use a 2D nine-point stencil, as shown in Fig. 8.3B.

Fig. 8.4 summarizes the concepts of discretization, 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 that is 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 will focus on the computation pattern in which a stencil is applied to all the relevant input grid points to generate the output values at all grid points, which will be referred to as a *stencil sweep*.

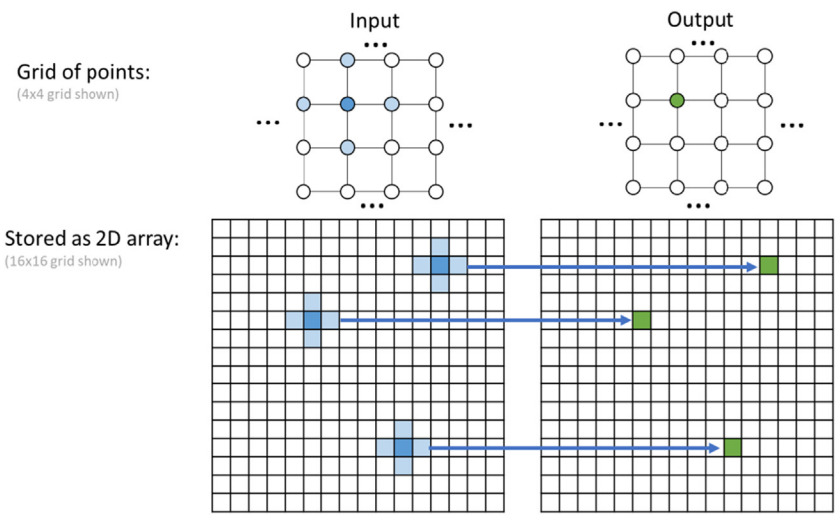


FIGURE 8.4

A 2D grid example and a five-point (order 1) stencil used to calculate the approximate derivative values at grid points.

8.2 Parallel stencil: a basic algorithm

We will 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 will not change from input to output, as illustrated in Fig. 8.5. That is, the shaded inside area in the output grid will be calculated, whereas the unshaded boundary cells will remain the same as the input values. This is a reasonable assumption, since stencils are used mainly 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 that each thread is assigned to one of the output grid points. A 2D tiling example of the output grid in which each thread block is responsible for a

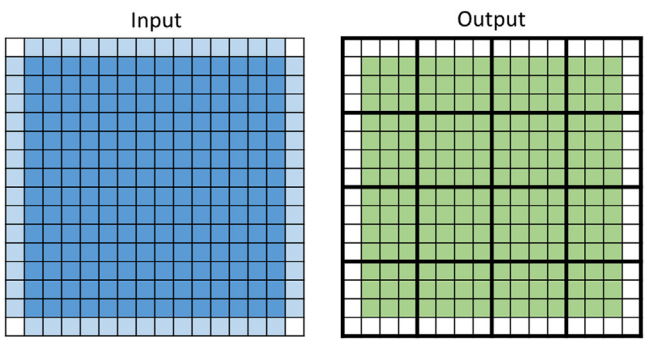


FIGURE 8.5

Simplifying boundary condition. The boundary cells contain boundary conditions that will not be 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.

4 × 4 output tile is shown in Fig. 8.5. However, since most real-world applications solve three-dimensional (3D) differential equations, the kernel in Fig. 8.6 assumes a 3D grid and a 3D seven-point stencil like the one in Fig. 8.3C. 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 depend on the differential equation that is being solved, as we explained in the background section.

Let us now calculate the floating-point to global memory access ratio for the kernel in Fig. 8.6. Each thread performs 13 floating-point operations (seven multiplications and six additions) and loads seven input values that are 4 bytes each. Therefore the floating-point to global memory access ratio for this kernel is 13/(7∗4) = 0.46 OP/B (operations per byte). As we discussed in Chapter 5, Memory Architecture and Data Locality, this ratio needs to be much larger for the performance of the kernel to be reasonably close to the level that is supported by the arithmetic compute resources. We will need to use a tiling technique like that discussed in Chapter 7, Convolution, to elevate the floating-point to global memory access ratio.

8.3 Shared memory tiling for stencil sweep

As we saw in Chapter 5, Memory Architecture and Data Locality, the ratio of floating-point operations to global memory accessing operations can be significantly elevated with shared memory tiling. As the reader probably suspected, the design of shared memory tiling for stencils is almost identical to that of convolution. However, there are a few subtle but important differences.

Fig. 8.7 shows the input and output tiles for a 2D five-point stencil applied to a small grid example. A quick comparison with Figure 7.11 shows a small difference between convolution and stencil sweep: The input tiles of the five-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 the input data reuse in a 2D five-point stencil to be significantly lower than that in a 3 × 3 convolution. As we discussed in Chapter 7, Convolution, the upper bound of the arithmetic to global memory access ratio of a 2D 3 × 3 convolution is 4.5 OP/B. However, for a 2D five-point stencil, the upper bound on the ratio is only 2.5 OP/B. This is because each output grid point value uses only five input grid values, compared to the nine input pixel values in 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

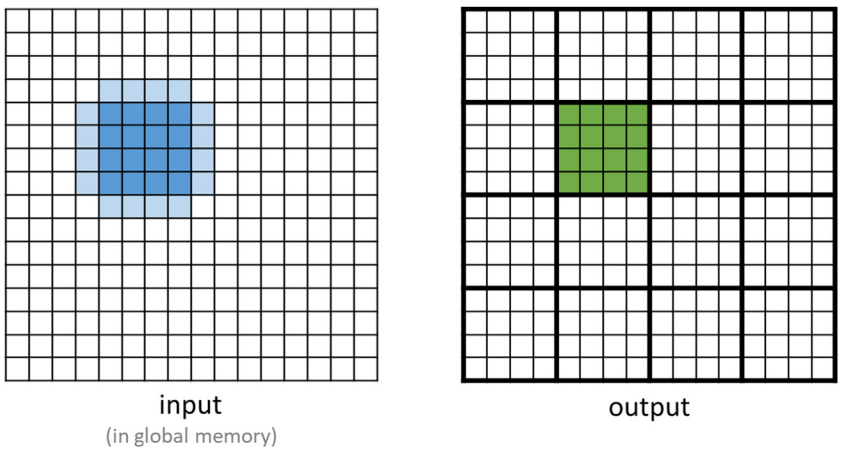


FIGURE 8.7

Input and output tiles for a 2D five-point stencil.

stencil from 1 (one grid point on each side, five-point stencil) to 2 (two grid points on each side, nine-point stencil), the upper bound on the ratio is 4.5 OP/B, compared to 12.5 OP/B for its counterpart 2D 5×5 convolution. This discrepancy is further pronounced when the order of the 2D stencil is increased to 3 (three grid points on each side, 13-point stencil). The upper bound on the ratio is 6.5 OP/B, compared to 24.5 OP/B for its counterpart 2D 7×7 convolution.

When we move into 3D, the discrepancy between the upper bound on the arithmetic to global memory access ratio of stencil sweep and convolution is even more pronounced. For example, a 3D third-order stencil (3 points on each side, 19-point stencil) has an upper bound of 9.5 OP/B, compared to 171.5 OP/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 a 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 in the chapter, this small but significant difference motivates the use of thread coarsening and register tiling in the third dimension.

Since all the strategies for loading input tiles for convolution apply directly to stencil sweep, we present in Fig. 8.8 a kernel like the convolution kernel in Figure 7.12 in which blocks are of the same size as input tiles and some of the threads are turned off in calculating output grid point values. The kernel is adapted from the basic stencil sweep kernel in Fig. 8.6, so we will focus only on the changes that are made during the adaptation. Like the tiled convolution kernel, the tiled stencil sweep kernel first calculates the beginning x, y, and z coordinates of the input patch that is used for each thread. The value 1 that is subtracted in each expression is because the kernel assumes a 3D seven-point stencil with

```
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 seven-point stencil sweep kernel with shared memory tiling.

one grid point on each side (lines 02−04). In general, the value that is 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 in lines 10−21. The conditions in 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 12−13 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 seven-point stencil.

We can evaluate the effectiveness of shared memory tiling by calculating the arithmetic to global memory access ratio that this kernel achieves. Recall that the original stencil sweep kernel achieved a ratio of 0.46 OP/B. With the shared memory tiling kernel, let us assume that each input tile is a cube with T grid points in each dimension and that 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 $13*(T-2)^3$ floating-point arithmetic operations. Moreover, each block loads an input tile by performing T^3 loads that are 4 bytes each. Therefore the floating point to global memory access ratio of the tiled kernel can be calculated as follows:

$$\frac{13*(T-2)^3}{4*T^3} = \frac{13}{4}*\left(1-\frac{2}{T}\right)^3 \quad OP/B$$

That is, the larger the T value, the more the input grid point values are reused. The upper bound on the ratio as T increases asymptotically is 13/4 = 3.25 OP/B.

Unfortunately, the 1024 limit of the block size on current hardware makes it difficult to have large T values. 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 that is 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. In contrast, convolution is often used to process 2D images in which larger tile dimensions (T value) such as 32×32 can be used.

There are two major disadvantages to the small limit on the T value that is imposed by the hardware. The first disadvantage is that it limits the reuse ratio and thus the compute to memory access ratio. For $T = 8$ the ratio for a seven-point stencil is only 1.37 OP/B, which is much less than the upper bound of 3.25 OP/B. The reuse ratio decreases as the T value decreases because of the halo overhead. As we discussed in Chapter 7, Convolution, halo elements are less reused than the nonhalo elements are. As the portion of halo elements in the input tile increases, both the data reuse ratio and the floating-point to global memory access ratio decrease. 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 that consists of 32 threads will be responsible for loading four different rows of the tile that have

eight input values each. Hence on the same load instruction the threads of the warp will access at least four distant locations in global memory. These accesses cannot be coalesced and will underutilize the DRAM bandwidth. Therefore T needs to be a much larger value for the reuse level to be closer to 3.25 OP/B and to enable full use of the DRAM bandwidth. The need for a larger T motivates the approach that we will cover in the following section.

8.4 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 that is 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 Section 6.3 that with thread coarsening, the programmer partially serializes parallel units of work into each thread and reduces the price paid for parallelism. In this case, the price that is paid for parallelism is the low data reuse due to the loading of halo elements by each block.

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

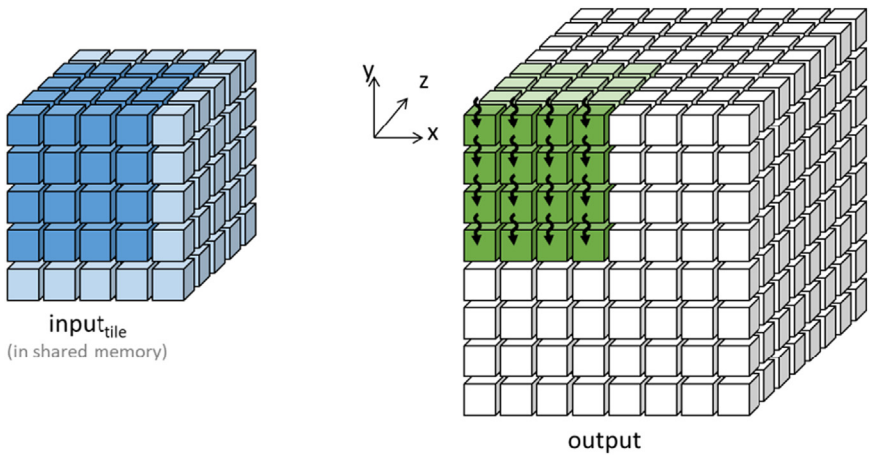


FIGURE 8.9

Thread coarsening in the z direction for a 3D seven-point stencil sweep.

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 that is 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 show only the internal threads of the thread block that are active in computing output tile values (i.e., 4×4).

Fig. 8.10 shows a kernel with thread coarsening in the z direction for a 3D seven-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 will be processing an x-y plane of an output tile; thus they will all be calculating output grid points whose z indices are identical.

```
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 seven-point stencil sweep.

Initially, each block needs to load into the shared memory the three input tile planes that contain all the points that are 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 load the first layer (lines 08–10) into the shared memory array `inPrev_s` and the second layer (lines 11–13) into the shared memory array `inCurr_s`. 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 all have 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 everyone 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 that are played by the input tile planes change when the threads move by one output plane in the z direction. Thus 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.

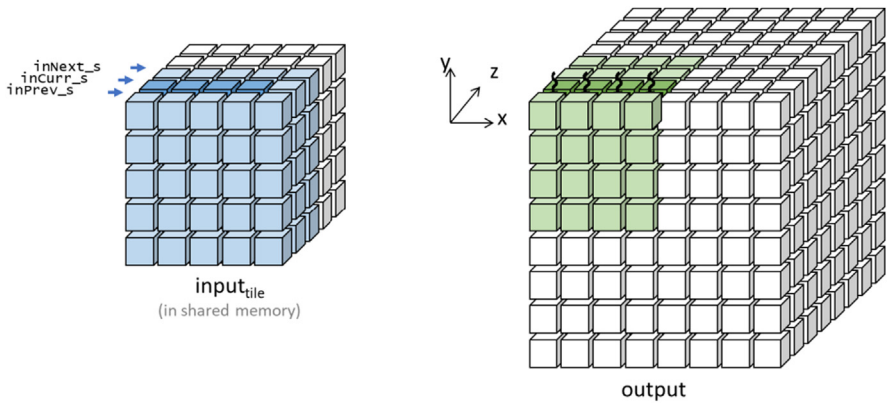


FIGURE 8.11

The mapping of the shared memory arrays to the input tile after the first iteration.

The advantages of the thread coarsening kernel are that it increases the tile size without increasing the number of threads and that it does not require all planes of the input tile to be present in the shared memory. The thread block size is now only T^2 instead of T^3, so we can use a much larger T value, such as 32, which would result in a block size of 1024 threads. With this T value we can expect that the floating-point arithmetic to global memory access ratio will be $\frac{13}{4} * \left(1 - \frac{2}{32}\right)^3 = 2.68\text{OP/B}$, which is a significant improvement over the 1.37 OP/B ratio of the original shared memory tiling kernel and closer to the 3.25 OP/B upper bound. 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 $3T^2$ elements instead of T^3 elements. For $T = 32$ the shared memory consumption is now at a reasonable level of $3*32^2*4\text{B} = 12\text{KB}$ per block.

8.5 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 fall into this description. The 3D seven-point stencil sweep kernel in Fig. 8.10 reflects this property. Each `inPrev_s` and `inNext_s` element is used by only 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.

We take advantage of this property with the register tiling kernel in Fig. 8.12. The kernel is built on the thread coarsening kernel in Fig. 8.10 with some simple but important modifications. We will focus on these modifications. First, we create the three register variables `inPrev`, `inCurr`, and `inNext` (lines 05, 07, 08). The register variables `inPrev` and `inNext` replace the shared memory arrays `inPrev_s` and `inNext`, 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 that is 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 destination. Thus the three planes of the "active part" of the input tile are held in the registers across the threads of the same block. In addition, the kernel always maintains a copy of the current plane of the input tile in 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.

```
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 seven-point stencil sweep.

The kernels in Figs. 8.10 and 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 seven-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 shifted into 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. 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 tradeoff 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.

Note that the idea of storing a tile of data collectively in the registers of a block's threads is one that we have seen before. In the matrix multiplication kernels in Chapters 3, Multidimensional Grids and Data, and 5, Memory Architecture and Data Locality, and in the convolution kernel in Chapter 7, Convolution, we stored the output value computed by each thread in a register of that thread. Hence the output tile that is computed by a block was stored collectively in the registers of that block's threads. Therefore register tiling is not a new optimization but one that we have applied before. It just became more apparent in this chapter because we are now using registers to store part of the input tile: the same tile was sometimes stored in registers and other times stored in shared memory throughout the course of the computation.

8.6 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 motivates thread coarsening for 3D stencils to enable larger input tiles and more data reuse. Second, the stencil patterns can sometimes enable register tiling of input data to further improve data access throughput and alleviate shared memory pressure.

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 that is computed during each stencil sweep?
   b. For the basic kernel in Fig. 8.6, what is the number of thread blocks that are 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 that are 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 that are needed, assuming a block size of 32×32?

2. Consider an implementation of a seven-point (3D) stencil with shared memory tiling and thread coarsening applied. The implementation is similar to those in Figs. 8.10 and 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 floating point to global memory access ratio (in OP/B) of the kernel?

d. How much shared memory (in bytes) is needed by each thread block if register tiling is not used, as in Fig. 8.10?

e. How much shared memory (in bytes) is needed by each thread block if register tiling is used, as in Fig. 8.12?

CHAPTER

Parallel histogram

An introduction to atomic operations and privatization

9

Chapter Outline

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, in which every thread can write into its designated output element(s) without any concern about interference from other threads. This chapter introduces the parallel histogram computation pattern, in which each output element can potentially be updated by any thread. Therefore one must take care to coordinate among threads as they update output elements and avoid any interference that could corrupt the final result. In practice, there are many other important parallel computation patterns in which output interference cannot be easily avoided. Therefore parallel histogram algorithms provide an example of output interference that occurs in these patterns. 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 to be able to reason about their applicability in different circumstances.

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00015-X

9.1 Background

A histogram is a display of the number count or percentage of occurrences of data values in a dataset. In the most common form of histogram, the value intervals are plotted along the horizontal axis, and the count of data values in each interval is represented as the height of a rectangle, or bar, rising from the horizontal axis. For example, a histogram can be used to show the frequency of letters of the alphabet in the phrase "programming massively parallel processors." For simplicity we assume that the input phrase is in all lowercase. By inspection we see that there are four instances of the letter "a," zero of the letter "b," one of the letter "c," and so on. We define each value interval as a continuous range of four letters of the alphabet. Thus the first value interval is "a" through "d," the second value interval is "e" through "h," and so on. Fig. 9.1 shows a histogram that displays the frequency of letters in the phrase "programming massively parallel processors" according to our definition of value intervals.

Histograms provide useful summaries of datasets. In our example we can see that the phrase being represented consists of letters that are heavily concentrated in the middle intervals of the alphabet and is noticeably sparse in the later intervals. The shape of the histogram is sometimes referred to as a *feature* of the dataset and provides a quick way to determine whether there are significant phenomena in the dataset. For 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 datasets for data analysis. One such area is computer vision. Histograms of different types of object images, such as faces versus cars, tend to exhibit different shapes. For example, one can plot the histogram of pixel luminous values in an image or an

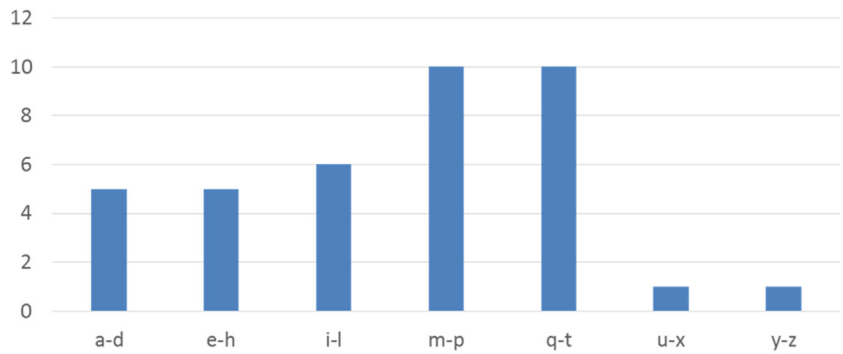


FIGURE 9.1

A histogram representation of the phrase "programming massively parallel processors."

area of an image. Such a histogram of the sky on a sunny day might have only a small number of very tall bars in the high-value intervals of the luminous spectrum. By dividing an image into 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, in which the features refer to patterns 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 the histogram defined in Fig. 9.1. For simplicity the histogram function is required to recognize only lowercase letters. The C code assumes that the input dataset comes in a `char` array `data` and the histogram will be generated into the `int` array `histo` (line 01). The number of input data items is specified in function parameter `length`. The for-loop (lines 02–07) sequentially traverses the array, identifies the alphabet index of the character in the visited position `data[i]`, saves the alphabet index into the alphabet_position variable, and increments the `histo[alphabet_position/4]` element associated with that interval. The calculation of the alphabet index relies on the fact that the input string is based on the standard ASCII code representation in which the alphabet letters "a" through "z" are encoded into consecutive values according to their order in the alphabet.

Although one might not know the exact encoded value of each letter, one can assume that the encoded value of a letter is the encoded value of "a" plus the alphabet position difference between that letter and "a." In the input, each character is stored in its encoded value. Thus the expression data[i] − "a" (line 03) derives the alphabet position of the letter with the alphabet position of "a" being 0. If the position value is greater than or equal to 0 and less than 26, the data character is indeed a lowercase alphabet letter (line 04). Keep in mind that we defined the intervals

```
01  void histogram_sequential(char *data, unsigned int length,
                              unsigned int *histo) {
02      for(unsigned int i = 0; i < length; ++i) {
03          int alphabet_position = data[i] - 'a';
04          if(alphabet_position >= 0 && alphabet_position < 26)
05              histo[alphabet_position/4]++;
06          }
07      }
08  }
```

FIGURE 9.2

A simple C function for calculating histogram for an input text string.

such that each interval contains four alphabet letters. Therefore the interval index for the letter is its alphabet position value divided by 4. We use the interval index to increment the appropriate `histo` array element (line 05).

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 data 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 `histo` array is so small that it fits well in the level-one (L1) data cache of the CPU, which ensures fast updates to the `histo` elements. For most modern CPUs, one can expect the execution speed of this code to be memory bound, that is, 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 histogram computation is to launch as many threads as there are data elements and have each thread process one input element. Each thread reads its assigned input element and increments the appropriate interval counter for the character. Fig. 9.3 illustrates an example of this parallelization strategy. Note that multiple threads need to update the same counter (m-p), which is a conflict that is 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 histo 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.

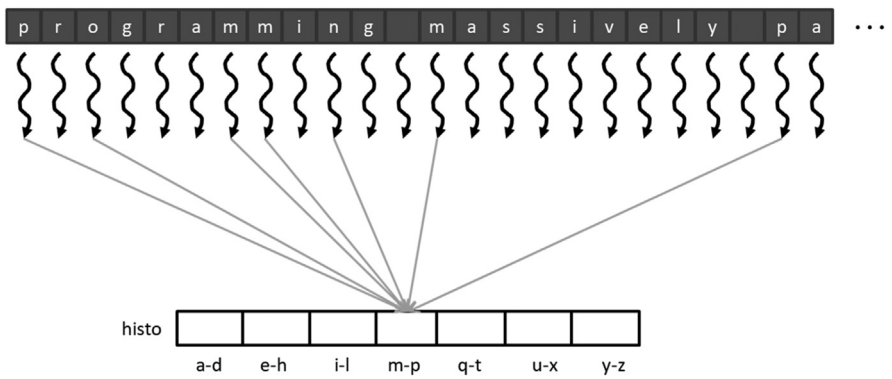


FIGURE 9.3

Basic parallelization of a histogram.

Time	Thread 1	Thread 2
1	(0) Old ← histo[x]	
2	(1) New ← Old + 1	
3	(1) histo[x] ← New	
4		(1) Old ← histo[x]
5		(2) New ← Old + 1
6		(2) histo[x] ← New

(A)

Time	Thread 1	Thread 2
1	(0) Old ← histo[x]	
2	(1) New ← Old + 1	
3		(0) Old ← histo[x]
4	(1) histo[x] ← New	
5		(1) New ← Old + 1
6		(1) histo[x] ← New

(B)

FIGURE 9.4

Race condition in updating a `histo` array element: (A) One possible interleaving of instructions; (B) Another possible interleaving of instructions.

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 that changes 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 think 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 situations happen in real life because of 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, verifies the ticket, and updates the display number to the next higher number. Ideally, all customers will be served in the order in which they enter the store. 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 to be the one who should receive service.

In both examples, undesirable outcomes are caused by a phenomenon called read-modify-write *race condition*, in which the outcome of two or more simultaneous update operations varies depending on the relative timing of the operations that are involved.[1] Some outcomes are correct, and some are incorrect. Fig. 9.4 illustrates a race condition when two threads attempt to update the same histo element in our text histogram example. Each row in Fig. 9.4 shows the activity during a time period, with time progressing from top to bottom.

[1]Note that this is similar but not the same as the write-after-read race condition in Chapter 10, Reduction and Minimizing Divergence, when we discussed the need for a barrier synchronization between the reads from and writes to the XY array in each iteration of the Kogge-Stone scan kernel.

Fig. 9.4A depicts a scenario in which 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 that the value of histo[x] was initially 0. In this scenario the value of histo[x] afterwards is 2, exactly what one would expect. That is, both threads successfully incremented histo[x]. The element value starts with 0 and becomes 2 after the operations complete.

In Fig. 9.4B the read-modify-write sequences of the two threads overlap. Note that thread 1 writes the new value into histo[x] at time period 4. When thread 2 reads histo [x] at time period 3, it still has the value 0. As a result, the new value that it calculates and eventually writes to histo[x] is 1 rather than 2. The problem is that thread 2 read histo[x] too early, before thread 1 had completed its update. The net outcome is that the value of histo[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. 9.5A, thread 2 completes its update before thread 1 starts its. In Fig. 9.5B, thread 1 starts its update before thread 2 has completed its. It should be obvious that the sequences in Fig. 9.5A result in a correct outcome for histo[x], but those in Fig. 9.5B produce an incorrect outcome.

The fact that the final value of histo[x] varies depending on the relative timing of the operations that are 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. 9.4A and 9.5A while eliminating the possibilities shown in Figs. 9.4B and 9.5B. This can be accomplished by 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, hence 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

Time	Thread 1	Thread 2
1		(0) Old ← histo[x]
2		(1) New ← Old + 1
3		(1) histo[x] ← New
4	(1) Old ← histo[x]	
5	(2) New ← Old + 1	
6	(2) histo[x] ← New	

(A)

Time	Thread 1	Thread 2
1		(0) Old ← histo[x]
2		(1) New ← Old + 1
3	(0) Old ← histo[x]	
4		(1) histo[x] ← New
5	(1) New ← Old + 1	
6	(1) histo[x] ← New	

(B)

FIGURE 9.5

Race condition scenarios in which thread 2 runs ahead of thread 1: (A) One possible interleaving of instructions; (B) Another possible interleaving of instructions.

operation is complete. In our example, such support eliminates the possibility depicted in Figs. 9.4B and 9.5B, since the trailing thread cannot start its update sequence until the leading thread has completed its.

It is important to remember that atomic operations do not enforce any particular execution order between threads. In our example, both orders shown in Figs. 9.4A and 9.5A are allowed by atomic operations. Thread 1 can run either ahead of or behind thread 2. The rule that is being enforced is that if both threads perform atomic operations on the same memory location, the atomic operation that is performed by the trailing thread cannot be started until the atomic operation of the leading thread completes. This effectively serializes the atomic operations that are being performed on a memory location.

Atomic operations are usually named according to the modification that is performed on the memory location. In our text histogram example we are adding a value to the memory location, so the atomic operation is called atomic add. Other types of atomic operations include subtraction, increment, decrement, minimum, maximum, logical and, and logical or. A CUDA kernel can perform an atomic add operation on a memory location through a function call:

```
int atomicAdd(int* address, int val);
```

The atomicAdd function is an intrinsic function (see the sidebar "Intrinsic Functions") that is compiled into a hardware atomic operation instruction. This instruction reads the 32-bit word pointed to by the address argument in global or shared memory, adds val to the old content, and stores the result back to memory at the same address. The function returns the old value at the address.

Intrinsic Functions

Modern processors often offer special instructions that either perform critical functionality (such as the atomic operations) or substantial performance enhancement (such as vector instructions). These instructions are typically exposed to the programmers as intrinsic functions, or simply instrinsics. 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.

Fig. 9.6 shows a CUDA kernel that performs parallel histogram computation. 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

```
01  __global__ void histo_kernel(char *data, unsigned int length,
                                 unsigned int *histo)  {
02      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03      if (i < length) {
04          int alphabet_position = data[i] - 'a';
05          if (alphabet_position >= 0 && alpha_position < 26) {
06              atomicAdd(&(histo[alphabet_position/4]), 1);
07          }
08      }
09  }
```

FIGURE 9.6

A CUDA kernel for calculation histogram.

calculation (line 02) and a boundary check (line 03) to assign a thread to each input element. The second distinction is that the increment expression in Fig. 9.2:

```
histo[alphabet_position/4]++
```

becomes an `atomicAdd()` function call in Fig. 9.6 (line 06). The address of the location to be updated, `&(histo[alphabet_position/4])`, is the first argument. The value to be added to the location, 1, is the second argument. This ensures that any simultaneous updates to any `histo` array element by different threads are properly serialized.

9.3 Latency and throughput of atomic operations

The atomic operation that is used in the kernel of Fig. 9.6 ensures the correctness of updates by serializing any simultaneous updates to a location. As we 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, Memory Architecture and Data Locality, the access latency to data in DRAMs can take hundreds of clock cycles. In Chapter 4, Compute Architecture and Scheduling, we learned that GPUs use zero-cycle context switching to tolerate such latency. In Chapter 6, Performance Considerations, 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. Therefore 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

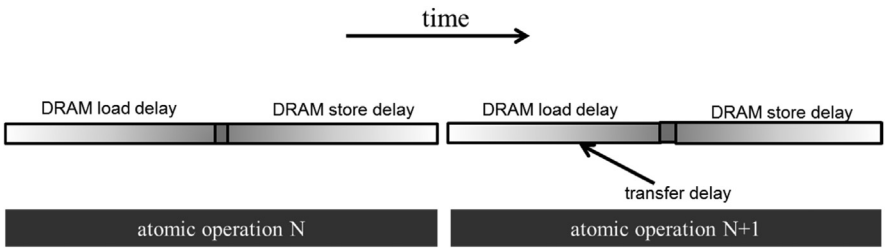


FIGURE 9.7

The throughput of an atomic operation is determined by the memory access latency.

complete. As is 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 minimum 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 a 64-bit (8-byte) double data rate DRAM interface per channel, eight channels, 1 GHz clock frequency, and typical access latency of 200 cycles. The peak access throughput of the memory system is 8 (bytes/transfer) * 2 (transfers per clock per channel) * 1 G (clocks per second) * 8 (channels) = 128 GB/s. Assuming that each data accessed is 4 bytes, the system has a peak access throughput of 32 G data elements per second.

However, in performing atomic operations on a particular memory location, the highest throughput that 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 time-based 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 text histogram example, the histogram has seven intervals. If the input characters were uniformly distributed in the alphabet, the atomic operations would be evenly distributed among the histo elements. This would boost the throughput to 7*2.5 M = 17.5 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 characters tend to have a biased distribution in the alphabet. For example, in Fig. 9.1 we see that the characters in the example phrase are heavily biased toward the m-p and q-t intervals. The heavy contention traffic to update these intervals will likely reduce the achievable throughput to only around (28/10)*2.5 M = 7 M.

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 to be performed in the last-level cache, which is shared among all streaming multiprocessors (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 that are updated by atomic operations tend to be heavily accessed by many threads, these variables tend to remain in the cache once they have been brought in from DRAM. 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 compared to early generations of GPU. This is an important reason why most modern GPUs support atomic operations in the last-level cache.

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 the heavy output interference problems in parallel computing. 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 original 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. For another example, we can create four private copies and have blocks whose indices are of the form $4n+i$ to update the ith private version for $I = 0, \ldots, 3$. A common approach is to create a private copy for each thread block. This approach has multiple advantages that we will see later.

Fig. 9.8 shows an example of how privatization is applied to the text histogram example from Fig. 9.3. In this example the threads are organized into thread blocks, each of which consists of eight threads (in practice, thread blocks are much larger). Each thread block receives a private copy of the histogram that it updates. As shown in Fig. 9.8, rather than having contention across all

the threads that update the same histogram bin, contention will be experienced only between threads in the same block and when the private copies are being merged at the end.

Fig. 9.9 presents a simple kernel that creates and associates a private copy of the histogram to every block. In this scheme, up to 1024 threads would work on a copy of the histogram. In this kernel the private histograms are in the global memory. These private copies will likely be cached in the L2 cache for reduced latency and improved throughput.

The first part of the kernel in Fig. 9.9 (lines 02−08) is similar to the kernel in Fig. 9.6 with one key distinction. The kernel in Fig. 9.9 assumes that the host code will allocate enough device memory for the `histo` array to hold all the

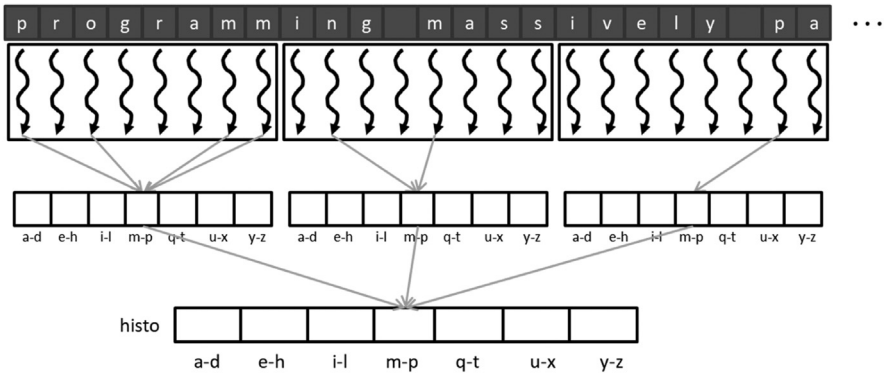


FIGURE 9.8

Private copies of histogram reduce contention of atomic operations.

```
01  __global__ void histo_private_kernel(char *data, unsigned int length,
                                          unsigned int *histo) {
02    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03    if(i < length) {
04      int alphabet_position = data[i] - 'a';
05      if (alphabet_position >= 0 && alphabet_position < 26) {
06         atomicAdd(&(histo[blockIdx.x*NUM_BINS + alphabet_position/4]), 1);
07      }
08    }
09    if(blockIdx.x > 0) {
10      __syncthreads();
11      for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x){
12         unsigned int binValue = histo[blockIdx.x*NUM_BINS + bin];
13         if(binValue > 0) {
14            atomicAdd(&(histo[bin]), binValue);
15         }
16      }
17    }
18  }
```

FIGURE 9.9

Histogram kernel with private versions in global memory for thread blocks.

private copies of the histogram, which amounts to gridDim.x*NUM_BINS*4 bytes. This is reflected in line 06, where each thread adds an offset of blockIdx.x*NUM_BINS to the index when performing atomic add on `histo` elements (bins of the histogram). This offset shifts the position to the private copy for the block to which the thread belongs. 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 orders of magnitude of improvement in the update throughput for the kernel.

At the end of the execution, each thread block will commit the values in private copy into the version produced by block 0 (lines 09–17). That is, we promote the private copy of block 0 into a public copy that will hold the total results from all block. The threads in the block first wait for each other to finish updating the private copy (line 10). Next, the threads iterate over the private histogram bins (line 11), each thread being responsible for committing one or more of the private bins. A loop is used to accommodate an arbitrary number of bins. Each thread reads the value of the private bin for which it is responsible (line 13) and checks whether the bin is nonzero (line 13). If it is, the thread commits the value by atomically adding it to the copy of block 0 (line 14). Note that the addition needs to be performed atomically because threads from multiple blocks can be simultaneously performing the addition on the same location. Therefore at the end of the kernel execution, the final histogram will be in the first NUM_BINS elements of the histo array. Since only one thread from each block will be updating any given histo array element during this phase of the kernel execution, the level of contention for each location is very 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. If the private copy were accessed by multiple blocks, we would have needed to call another kernel to merge the private copies or used other sophisticated techniques. Another benefit of creating a private copy of the histogram on a per-thread-block basis is that if the number of bins in the histogram is small enough, the private copy of the histogram can be declared in shared memory. Using shared memory would not be possible if the private copy were accessed by multiple blocks because blocks do not have visibility of each other's shared memory.

Recall that any reduction in latency directly translates into improved throughput of atomic operations on the same memory location. The latency for accessing memory can be dramatically reduced by placing data 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 increased throughput of atomic operations.

Fig. 9.10 shows a privatized histogram kernel that stores the private copies 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 `histo_s` array and is initialized to 0 in parallel by the threads of

```
01 __global__ void histo_private_kernel(char* data, unsigned int length,
                                        unsigned int* histo) {
02       // Initialize privatized bins
03     shared   unsigned int histo_s[NUM_BINS];
04   for(unsigned int bin = threadIdx.x; bin< NUM_BINS; bin += blockDim.x) {
05      histo_s[bin] = 0u;
06   }
07   __syncthreads();
08      // Histogram
09   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10    if(i < length) {
11      int alphabet_position = data[i] - 'a';
12      if(alphabet_position >= 0 && alphabet_position < 26) {
13         atomicAdd(&(histo_s[alphabet_position/4]), 1);
14      }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19          unsigned int binValue = histo_s[bin];
20          if(binValue > 0) {
21            atomicAdd(&(histo[bin]), binValue);
22          }
23      }
24  }
```

FIGURE 9.10

A privatized text histogram kernel using the shared memory.

the block (line 02−06). The barrier synchronization (line 07) 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 that in Fig. 9.9 except that the first atomic operation is performed on the elements of the shared memory array `histo_s` (line 13) and the private bin value is later read from there (line 19).

9.5 Coarsening

We have seen that privatization is effective in 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 commit the private copies to the public copy. This commit operation is 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 that are launched exceeds the number that can be executed simultaneously by the hardware, the hardware schedule 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 are 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 09–10. In Fig. 9.10 the input element index `i` corresponded to the global thread index, so

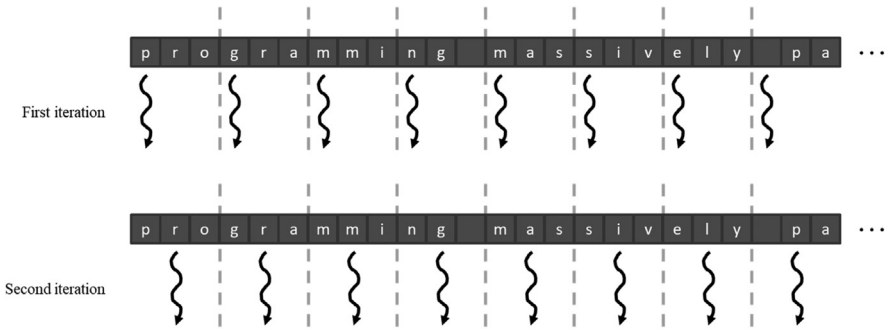


FIGURE 9.11

Contiguous partitioning of input elements.

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                          unsigned int* histo) {
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
05        histo_s[binIdx] = 0u;
06    }
07     __syncthreads();
08      // Histogram
09    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
10    for(unsigned int i=tid*CFACTOR; i<min((tid+1)*CFACTOR, length); ++i) {
11        int alphabet_position = data[i] - 'a';
12        if(alphabet_position >= 0 && alphabet_position < 26) {
13              atomicAdd(&(histo_s[alphabet_position/4]), 1);
14        }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19        unsigned int binValue = histo_s[binIdx];
20        if(binValue > 0) {
21          atomicAdd(&(histo[binIdx]), binValue);
22        }
23    }
24  }
```

FIGURE 9.12

Histogram kernel with coarsening using contiguous partitioning.

each thread received one input element. In Fig. 9.11 the input elements index `i` is the index of a loop that iterates from `tid*CFACTOR` to `(tid + 1)*CFACTOR` where `CFACTOR` is the coarsening factor. Therefore each thread takes a contiguous segment of `CFACTOR` elements. The `min` operation in the loop bound ensures that the threads at the end do not read out of bounds.

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 suboptimal memory access pattern. As we learned in Chapter 5, Memory Architecture and Data Locality, 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 characters 0 through 7 ("programm"). With memory coalescing, all the elements will be fetched with only one DRAM access. During the second iteration the four threads access the characters "ing mass" in one coalesced memory access. 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 characters (a 32-bit word) in each iteration to fully utilize the interconnect bandwidth between the caches and the SMs.

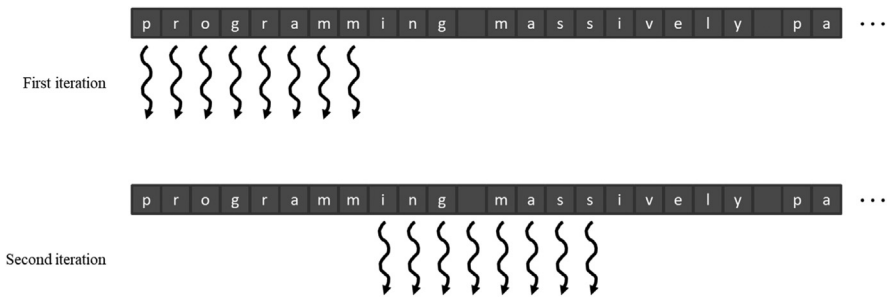


FIGURE 9.13

Interleaved partitioning of input elements.

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                          unsigned int* histo){
02      // Initialize privatized bins
03    __shared__ unsigned int histo_s[NUM_BINS];
04    for(unsigned int bin=threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
05          histo_s[binIdx] = 0u;
06    }
07    __syncthreads();
08      // Histogram
09    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
10    for(unsigned int i = tid; i < length; i += blockDim.x*gridDim.x) {
11       int alphabet_position = data[i] - 'a';
12       if(alphabet_position >= 0 && alphabet_position < 26) {
13           atomicAdd(&(histo_s[alphabet_position/4]), 1);
14       }
15    }
16    __syncthreads();
17      // Commit to global memory
18    for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
19       unsigned int binValue = histo_s[binIdx];
20       if(binValue > 0) {
21          atomicAdd(&(histo[binIdx]), binValue);
22       }
23    }
24  }
```

FIGURE 9.14

Histogram kernel with coarsening using interleaved partitioning.

Fig. 9.14 shows the histogram kernel with coarsening applied by using the interleaved partitioning strategy. The difference from Figs. 9.10 and 9.12 is again on lines 09−10. In the first iteration of the loop, each thread accesses the `data` array using its global thread index: Thread 0 accesses element 0, thread 1 accesses element 1, thread 2 accesses element 2, and so on. Thus all threads jointly process the first blockDim.x*gridDim.x elements of the input. In the second iteration, all threads add blockDim.x*gridDim.x to their indices and jointly process the next section of blockDim.x*gridDim.x elements.

When the index of a thread exceeds the valid range of the input buffer (its private `i` variable value is greater than or equal to `length`), the thread has completed processing its partition and will exit the loop. Since the size of the buffer might not be a multiple of the total number of threads, some of the threads might not participate in the processing of the last section. Therefore some threads will execute one fewer loop iteration than others.

9.6 Aggregation

Some datasets 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 a high concentration of identical values causes heavy contention and reduced throughput of parallel histogram computation.

For such datasets 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 (Merrill, 2015). Such aggregation reduces the number of atomic operations to the highly contended histogram elements, thus improving the effective throughput of the computation.

Fig. 9.15 shows an aggregated text histogram kernel. The key changes compared to the kernel in Fig. 9.14 are as follows: Each thread declares an additional `accumulator` variable (line 09) that keeps track of the number of updates aggregated thus far and a `prevBinIdx` variable (line 10) that tracks the index of the histogram bin that was last encountered and is being aggregated. Each thread initializes the `accumulator` variable to zero, indicating that no updates has been initially aggregated, and the `prevBinIdx` to -1 so that no alphabet input will match it.

```
01  __global__ void histo_private_kernel(char* data, unsigned int length,
                                          unsigned int* histo){
02      // Initialize privatized bins
03      __shared__ unsigned int histo_s[NUM_BINS];
04      for(unsigned int bin = threadIdx.x; bin < NUM_BINS; bin += blockDim.x){
05          histo_s[bin] = 0u;
06      }
07      __syncthreads();
08      // Histogram
09      unsigned int accumulator = 0;
10      int prevBinIdx = -1;
11      unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
12      for(unsigned int i = tid; i < length; i += blockDim.x*gridDim.x) {
13         int alphabet_position = data[i] - 'a';
14         if(alphabet_position >= 0 && alphabet_position < 26) {
15            int bin = alphabet_position/4;
16            if(bin == prevBinIdx) {
17                 ++accumulator;
18            } else {
19                 if(accumulator > 0) {
20                    atomicAdd(&(histo_s[prevBinIdx]), accumulator);
21                 }
22                 accumulator = 1;
23                 prevBinIdx = bin;
24            }
25         }
26      }
27      if(accumulator > 0) {
28           atomicAdd(&(histo_s[prevBinIdx]), accumulator);
29      }
30      __syncthreads();
31      // Commit to global memory
32      for(unsigned int bin = threadIdx.x; bin<NUM_BINS; bin += blockDim.x) {
33           unsigned int binValue = histo_s[bin];
34           if(binValue > 0) {
35             atomicAdd(&(histo[bin]), binValue);
36           }
37      }
38  }
```

FIGURE 9.15

An aggregated text histogram kernel.

When an alphabet data is found, the thread compares the index of the histogram element to be updated with that being aggregated (line 16). If the index is the same, the thread simply increments the accumulator (line 17), extending the streak of aggregated updates by one. If the index is different, the streak of aggregated updates to the histogram element has ended. The thread uses atomic operation to add the `accumulator` value to the histogram element whose index is tracked by `prevBinIdx` (lines 19–21). This effectively flushes out the total contribution of the previous streak of aggregated updates.

With this scheme, the update is always at least one element behind. In the extreme case in which there is no streak, all the updates will simply always be one element behind. This is the reason why after a thread has completed scanning all input elements and exited the loop, the threads need to check whether there is a need to flush out the `accumulator` value (line 27). If so, the accumulator value is flushed to the right `histo_s` element (line 28).

One important observation is that the aggregated kernel requires more statements and variables. Thus if the contention rate is low, an aggregated kernel may execute at lower speed than the simple kernel. However, if the data distribution leads to heavy contention in atomic operation execution, aggregation may result in significantly higher speed. The added if-statement can potentially exhibit control divergence. However, if there is either no contention or heavy contention, there will be little control divergence, since the threads would either all be flushing the accumulator value or all be in a streak. In the case in which some threads will be in a streak and some will be flushing out their accumulator values, the control divergence is likely to be compensated by the reduced contention.

9.7 Summary

Computing histograms is important for analyzing large datasets. It also represents an important class of parallel computation patterns in which the output location of each thread is data-dependent, which makes it infeasible to apply the 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. Coarsening was 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, for datasets that cause heavy contention, aggregation can also lead to significantly higher execution speed.

Exercises

1. Assume that each atomic operation in a DRAM system has a total latency of 100 ns. What is the maximum throughput that 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 Exercise 1, assume that a kernel performs five floating-point operations per atomic operation. What is the maximum floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
4. In Exercise 1, assume that we privatize the global memory variable into shared memory variables in the kernel and that 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 variable into the global variable adds 10% to the total execution time. Assume that a kernel performs five floating-point operations per atomic operation. What is the maximum floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
5. To perform an atomic add operation to add the value of an integer variable Partial to a global memory integer variable Total, which one of the following statements should be used?
   a. atomicAdd(Total, 1);
   b. atomicAdd(&Total, &Partial);
   c. atomicAdd(Total, &Partial);
   d. atomicAdd(&Total, Partial);
6. 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 that are 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

Merrill, D., 2015. Using compression to improve the performance response of parallel histogram computation, NVIDIA Research Technical Report.

CHAPTER

Reduction
And minimizing divergence

10

Chapter Outline

A reduction derives a single value from an array of values. The single value could be the sum, the maximum value, the minimal value, and so on among all elements. The value can also be of various types: integer, single-precision floating-point, double-precision floating-point, half-precision floating-point, characters, and so on. All these types of reductions have the same computation structure. Like a 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 get correct results. Such coordination must be done carefully to avoid performance bottlenecks, which are 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

Mathematically, a reduction can be defined for a set of items based on a binary operator if the operator has a well-defined identity value. For example, a floating-point addition operator has an identity value of 0.0; that is, an addition of any floating-point value v and value 0.0 results in the value v itself. Thus a reduction

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00023-9

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 the set. For example, 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.

A 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 identity value 0.0. It then uses a for-loop to iterate through the input array that holds the set 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 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. For real numbers, the identity value for the minimum operator is $+\infty$. A maximum (max) reduction can be defined for a maximum (max) comparison operator that returns the larger value of the two input. For real numbers, the identity value for the maximum operator is $-\infty$.

Fig. 10.2 shows a general form of 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 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

```
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.

smaller value is returned. The sequential algorithm ends when all the elements have been 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.

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.

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.

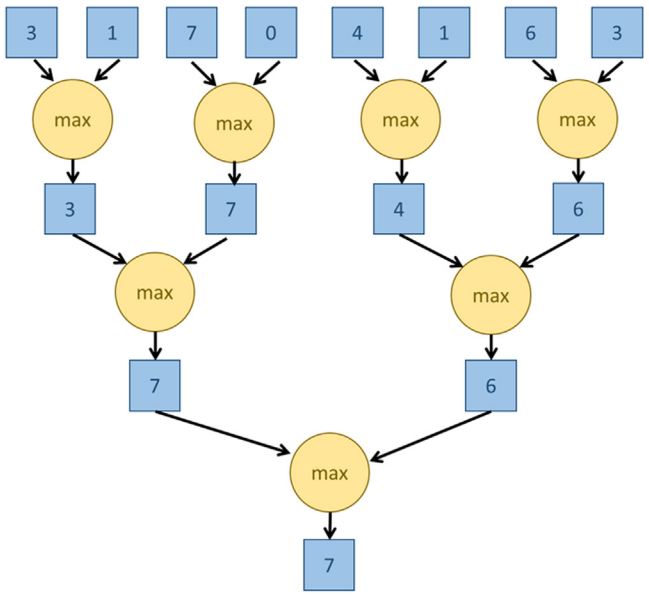


FIGURE 10.3

A parallel max reduction tree.

Note that the order of performing the operations will be changed from a sequential reduction algorithm to a parallel reduction algorithm. For example, for the input at the top of Fig. 10.3, 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 the input value 7 is first compared with the input value 0 before it is compared against the maximum of 3 and 1.

As we have seen, 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 $(a \Theta b) \Theta c = a \Theta (b \Theta c)$. For example, integer addition is associative ($(1+2)+3 = 1+(2+3) = 6$), whereas integer subtraction is not associative ($(1-2)-3 \neq 1-(2-3)$). That is, 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. Strictly speaking, floating-point additions are not associative, owing to potentially rounding results for different ways of introducing parentheses. However, most applications accept floating-point operation results to be the same if they are within a tolerable difference from each other. Such definition allows developers and compiler writers to treat floating-point addition as an associative operator. Interested readers are referred to Appendix A for a detailed treatment.

The conversion from the sequential reduction in Fig. 10.2 to the reduction tree in Fig. 10.3 requires that the operator be associative. We can think of a reduction as a list of operations. The difference of ordering between Fig. 10.2 and Fig. 10.3 is just inserting parenthesis 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 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, sum, and product. Obviously, not all operators are commutative. For example, addition is commutative ($1+2 = 2+1$), whereas integer subtraction is not ($1-2 \neq 2-1$).

The parallel reduction pattern in Fig. 10.3 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, in which 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.

The parallel operations result in significant improvement over the sequential code in the number of time steps needed to produce the final result. In the example in Fig. 10.3 the for-loop in the sequential code iterates eight times, or takes eight time steps, to visit all of the input elements and produce the final result. On the other hand, with the parallel operations in Fig. 10.3 the parallel reduction tree approach takes only three time steps: four max operations during the first time step, two during the second, and one during the third. This is a decrease of $5/8 = 62.5\%$ in terms of number of time steps, or a speedup of $8/3 = 2.67$. There is, of course, a cost to the parallel approach: One must have enough hardware comparators to perform up to four max operations in the same time step. For N input values, a reduction tree performs ½ N operations during the first round, ¼ N operations in the second round, and so on. Therefore the total number of operations that are performed is defined by the geometric series ½ N+¼ N+$^1/_8$ N+... $^1/_N$ N = N − 1 operations, which is similar to the sequential algorithm.

In terms of time steps, a reduction tree takes $\log_2 N$ steps to complete the reduction process for N input values. Thus it can reduce N = 1024 input values in just ten steps, assuming that there are enough execution resources. During the first step, we need ½ N = 512 execution resources! Note that the number of resources that are needed diminishes quickly as we progress in time steps. During the final time step, we need to have only one execution resource. The level of parallelism at each time step is the same as the number of execution units that are required. It is interesting to calculate the average level of parallelism across all time steps. The average parallelism is the total number of operations that are performed divided by the number of time steps, which is $(N - 1)/\log_2 N$. For N = 1024 the average parallelism across the ten time steps is 102.3, whereas the peak parallelism is 512 (during the first time step). Such variation in the level of parallelism and resource consumption across time steps makes reduction trees a challenging parallel pattern for parallel computing systems.

Fig. 10.5 shows a sum reduction tree example for eight input values. Everything we presented so far about the number of time steps and resources consumed 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. We will use this example to illustrate the design of sum reduction kernels.

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 quarterfinals, semifinals, 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 in which 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, and so on. With eight teams entering a tournament, four winners will emerge from the first round (quarterfinals in Fig. 10.4), two from the second round (semifinals in Fig. 10.4), and one final winner (champion) from the third round (final in Fig. 10.4). Each round is a time step of the reduction process.

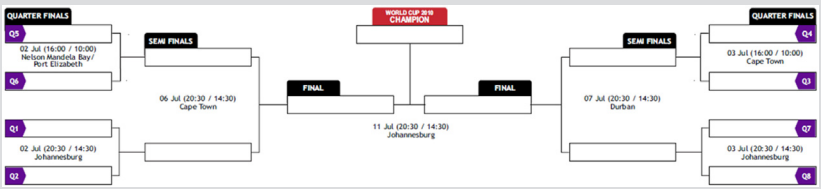


FIGURE 10.4

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 60,000 teams, we can determine the final winner in just 16 rounds. It is interesting to note that while reduction trees can greatly speed up 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 quarterfinals 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.

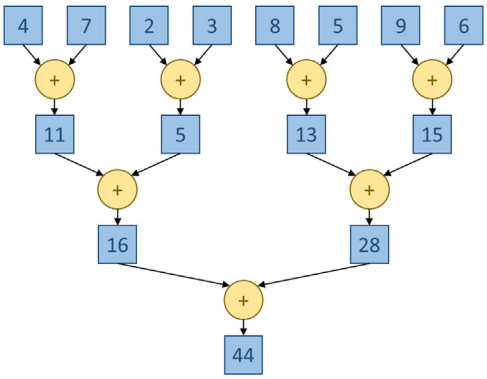


FIGURE 10.5

A parallel sum reduction tree.

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. 10.5. Since the reduction tree requires collaboration across all threads, which is not possible across an entire grid, we will 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 will call this simple kernel and launch a grid with one block of ½N 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 10.8. During the first time step, all ½N threads will participate, and each thread adds two elements to produce ½N partial sums. During the next time step, half of the threads will drop off, and only ¼N threads will continue to participate to produce ¼N partial sums. This process will continue until the last time step, in which only one thread will remain and produce the total sum.

Fig. 10.6 shows the code of the simple sum kernel function, and Fig. 10.7 illustrates the execution of the reduction tree that is implemented by this code. Note that in Fig. 10.7 the time progresses from top to bottom. We assume that the `input` array is in the global memory and that 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]`, and so on, as shown in the top row of Fig. 10.7. Each thread will be the "owner" of the location to which it is assigned and will be the only thread that writes into that location. The design of the kernel follows the "owner computes" approach, in which every data location is owned by a unique thread and can be updated only by that owner thread.

```
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 10.6

A simple sum reduction kernel.

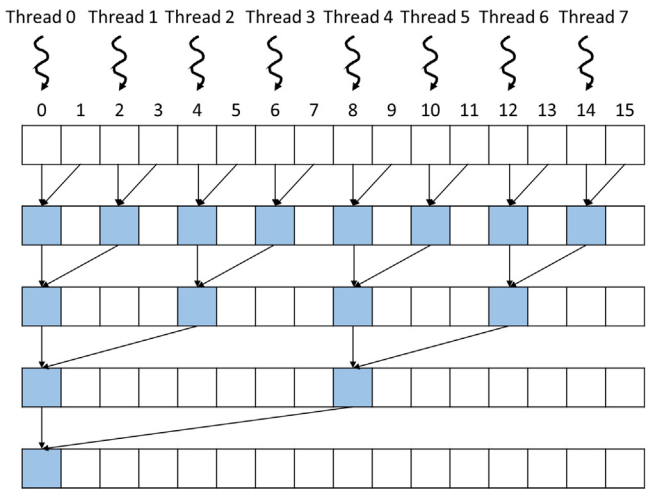


FIGURE 10.7

The assignment of threads ("owners") to the `input` array locations and progress of execution over time for the `SimpleSumReudctionKernel` in Fig. 10.6. The time progresses from top to bottom, and each level corresponds to one iteration of the for-loop.

The top row of Fig. 10.7 shows the assignment of threads to the locations of the input array, and each of the subsequent rows shows the writes to the input array locations at each time step, that is, iteration of the for-loop in Fig. 10.6 (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. 10.7. 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. 10.6 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 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. 10.7, 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 1, 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 the second iteration, 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 the second iteration, 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. 10.6 (line 04). The condition of the if-statement is set up to select the active threads in each iteration. As shown in Fig. 10.7, 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 whether 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, ... through the iterations, or 2^n for iteration n. Thus the condition indeed tests whether the thread index values are multiples of 2^n. Recall that all threads execute the same kernel code. The threads whose thread index value satisfies the if-condition are the active threads that perform the addition statement (line 05). The threads 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. 10.6) in the for-loop ensures that all partial sums that were 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. This way, all threads that enter an iteration will be able to correctly use the partial sums that were 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 have indeed been written to the even locations of the input array and are ready to be used by the active threads in the second iteration.

10.4 Minimizing control divergence

The kernel code in Fig. 10.6 implements the parallel reduction tree in Fig. 10.7 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. 10.7, only those threads whose `threadIdx.x` values are even will execute the addition statement during the second iteration. As we explained in Chapter 4, Compute Architecture and Scheduling, control divergence can significantly reduce the execution resource utilization efficiency, or the percentage of resources that are 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 5, only one out of the 32 threads in a warp are active, wasting $^{31}/_{32}$ of the execution resources.

If the size of the input array is greater than 32, entire warps will become inactive after the fifth iteration. For example, for an input size of 256, 128 threads or four warps would be launched. All four warps would have the same divergence pattern, as we explained in the previous paragraph for iterations 1 through 5. During the sixth 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 $^{31}/_{32}$ of the execution resource. During the seventh iteration, only warp 0 would be active, exhibiting control divergence and wasting $^{31}/_{32}$ of the execution resource.

In general, the execution resource utilization efficiency for an input array of size N can be calculated as the ratio between the total number of active threads to the total number of execution resources that are consumed. The total number of execution resources that are 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:

$$\left(N/64^{*}5 + N/64^{*}1/2 + N/64^{*}1/4 + \ldots + 1\right)^{*}32$$

Here, N/64 is the total number of warps that are launched, since N/2 threads will be launched and every 32 threads form a warp. The N/64 term is multiplied by 5 because all launched warps are active for five iterations. After the fifth iteration the number of warps is reduced by half in each successive iteration. The expression in parentheses 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 an input array size of 256, the consumed execution resource is (4*5+2+1)*32 = 736.

The number of execution results committed by the active threads is the total number of active threads across all iterations:

$$N/64^{*}(32 + 16 + 8 + 4 + 2 + 1) + N/64^{*}1/2^{*}1 + N/64^{*}1/4^{*}1 + \ldots + 1$$

The terms in the parenthesis give the active threads in the first five iterations for all N/64 warps. Starting at the sixth 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 an 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 that are needed to reduce 256 values is 255.

Putting the previous two results together, we find that the execution resource utilization efficiency for an input array size of 256 is 255/736 = 0.35. This ratio states that the parallel execution resources did not achieve their full potential in speeding up this computation. On average, only about 35% of the resources consumed contributed to the sum reduction result. That is, we used only about 35% of the hardware's potential to speed up the computation.

Based on this analysis, we see that there is widespread control divergence across warps and over time. As the reader might have wondered, there may be 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. 10.7 is that the partial sum locations become increasingly distant from each other, 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 control divergence.

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 decrease, rather than increase, over time. The revised assignment strategy is shown in Fig. 10.8 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

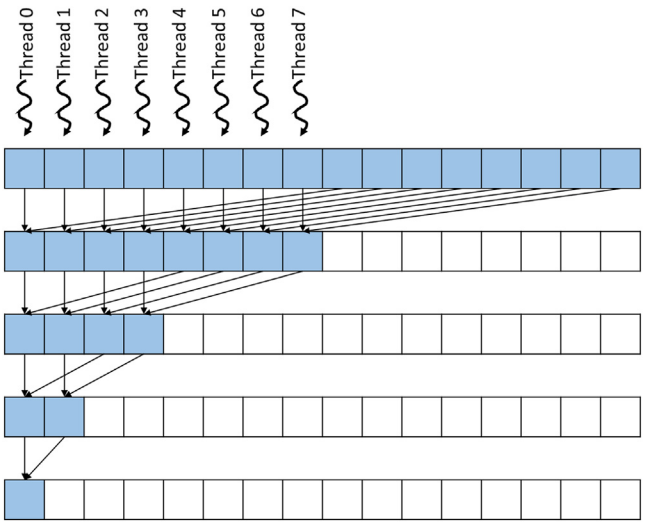


FIGURE 10.8

A better assignment of threads to input array locations for reduced control divergence.

to its owner location. In our example, thread 0 adds input[8] to its owned position input[0], thread 1 adds input[9] to its owned position input[1], and so on. 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 the third iteration 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. 10.8 to Fig. 10.7, there is effectively a reordering of the operands in the list rather than just inserting parentheses in different ways. For the result to always remain the same with such reordering, the operation must be commutative as well as being associative.

Fig. 10.9 shows a kernel with some subtle but critical changes to the simple kernel in Fig. 10.6. The owner position variable i is set to `threadIdx.x` rather than `2*threadIdx.x` (line 02). Thus the owner positions of all threads are now adjacent to each other, as illustrated in Fig. 10.8. The stride value is initialized as `blockDim.x` and is reduced by half until it reaches 1 (line 03). In each iteration, only the threads whose indices are smaller than the stride value remain active (line 04). Thus all active threads are of consecutive thread indices, as shown in Fig. 10.8. 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 that are added during the first round are `blockDim.x` away from each other. After the first iteration, all the pairwise sums are stored in the first half of the input array, as shown in Fig. 10.8. 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 a quarter of a section away from each other during the second iteration.

The kernel in Fig. 10.9 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. 10.6. Then 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

```
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 10.9

A kernel with less control divergence and improved execution resource utilization efficiency.

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 pairwise 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 warp 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. 10.9 does not completely eliminate the divergence caused by the if-statement. The reader should verify that for the 256-element example, starting with the fourth 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 follows:

$$\left(N/64^{*}1 + N/64^{*}1/_2 + \ldots + 1 + 5^{*}1\right)^{*}32$$

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 that are consumed.

For our 256-element example the execution resources that are consumed are (4+2+1+5*1)*32 = 384, which is almost half of 736, the resources that were consumed by the kernel in Fig. 10.6. Since the number of active threads in each iteration did not change from Fig. 10.7 to Fig. 10.8, the efficiency of the new kernel in Fig. 10.9 is 255/384 = 66%, which is almost double the efficiency of the kernel in Fig. 10.6. Note also that since the warps are scheduled to take turns executing in a streaming multiprocessor of limited execution resources, the total execution time will also improve with the reduced resource consumption.

The difference between the kernels in Fig. 10.6 and Fig. 10.9 is small but can have a significant performance impact. It requires someone with clear understanding of the execution of threads on the single-instruction, multiple-data hardware of the device to be able to confidently make such adjustments.

10.5 Minimizing memory divergence

The simple kernel in Fig. 10.6 has another performance issue: memory divergence. As we explained in Chapter 5, Memory Architecture and Data Locality, 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, in Fig. 10.7, 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 thread are not adjacent locations, the accesses that are made by adjacent threads will not be fully coalesced. During each iteration the memory locations that are collectively accessed by a warp are of stride distance away from each other.

For example, as shown in Fig. 10.7, 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, two global memory requests are triggered, and half the data returned will not be used by the threads. The same behavior occurs for the second read and the write. During the second iteration, every other thread drops out, and the locations that are collectively accessed by the warp are four elements away from each other. Two global memory requests are again performed, and only one-fourth of the data returned will be used by the threads. This will continue until there is only one active thread for each warp that remains active. Only when there is one active thread in the warp will the warp perform one global memory request. Thus the total number of global memory requests is as follows:

$$\left(N/64^{*}5^{*}2 + N/64^{*}1 + N/64^{*}1/2 + N/64^{*}1/4 + \ldots + 1\right)^{*}3$$

The first term (N/64*5*2) corresponds to the first five iterations, in which all N/64 warps have two or more active threads, so each warp performs two global memory requests. The remaining terms account for the final iterations, in which each warp has only one active thread and performs 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. 10.9 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 triggers only one global memory request on any read or write. As the iterations progress, entire warps drop out, so no global memory access will be performed by any thread in these inactive warps. Half of the warps drop out in each iteration until there is only one warp for the final five iterations. Therefore the total number of global memory requests performed by the kernel is as follows:

$$\left(\left(N/64 + N/64^{*}1/2 + N/64^{*}1/4 + \ldots + 1\right) + 5\right)^{*}3$$

For the 256-element example the total number of global memory requests performed is ((4+2+1)+5)*3 = 36. The improved kernel results in 141/36 = 3.9× fewer global memory requests. Since DRAM bandwidth is a limited resource, the execution time is likely to be significantly longer for the simple kernel in Fig. 10.6.

For a 2048-element example the total number of global memory requests that are performed by the kernel in Fig. 10.6 is (32*5*2+32+16+8+4+2+1)*3 = 1149, whereas the number of global memory requests that are performed by the kernel in Fig. 10.9 is (32+16+8+4+2+1+5)*3 = 204. The ratio is 5.6, even more than in the 256-element example. This is because of the inefficient execution pattern of the kernel in Fig. 10.6, in which there are more active warps during the initial five iterations of the execution and each active warp triggers twice the number of global memory requests as the convergent kernel in Fig. 10.9.

In conclusion, the convergent kernel offers more efficiency in using both execution resources and DRAM bandwidth. The advantage comes from both reduced control divergence and improved memory coalescing.

10.6 Minimizing global memory accesses

The convergent kernel in Fig. 10.9 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. Since the shared memory has much shorter latency and higher bandwidth than 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. 10.10.

The strategy for using the shared memory is implemented in the kernel shown in Fig. 10.11. The idea is to use each thread to load and add two of the original

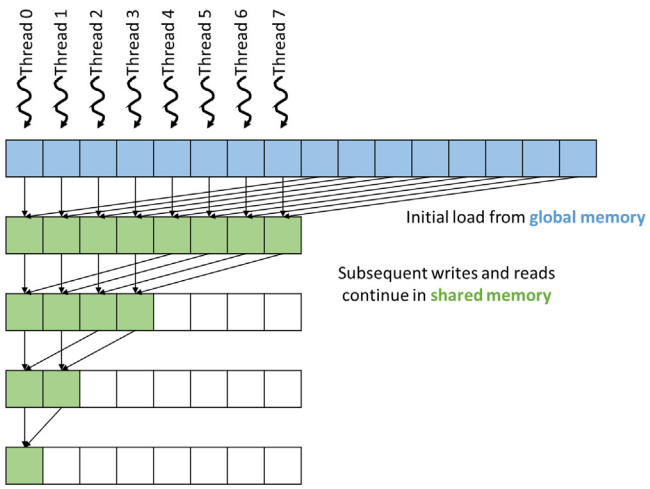


FIGURE 10.10

Using shared memory to reduce accesses to the global memory.

```
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 10.11

A kernel that uses shared memory to reduce global memory accesses.

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 in the previous kernels (lines 11−13).

Using the kernel in Fig. 10.11, 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 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. 10.11 (line 04) are coalesced. So with coalescing, there will be only (N/32) +1 global memory requests. For the 256-element example the total number of global memory requests that are triggered will be reduced from 36 for the kernel in Fig. 10.9 to 8+1 = 9 for the shared memory kernel in Fig. 10.10, a 4× improvement. Another 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 Hierarchical reduction for arbitrary input length

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. 10.12 illustrates the concept of hierarchical, segmented multiblock reduction using atomic operations, and Fig. 10.13 shows the corresponding kernel implementation. 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.

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

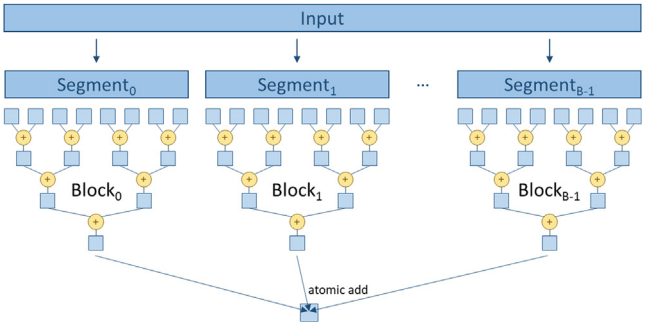


FIGURE 10.12

Segmented multiblock reduction using atomic operations.

```
01    __global__ SegmentedSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int segment = 2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        unsigned int t = threadIdx.x;
06        input_s[t] = input[i] + input[i + BLOCK_DIM];
07        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
08            __syncthreads();
09            if (t < stride) {
10                input_s[t] += input_s[t + stride];
11            }
12        }
13        if (t == 0) {
14              atomicAdd(output, input_s[0]);
15          }
16    }
```

FIGURE 10.13

A segmented multiblock sum reduction kernel using atomic operations.

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, and so on.

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 to which the thread belongs (line 04). The local variable i holds the owned location of the thread in the global input array, whereas t holds the owned location of the thread in the shared input_s array. Line 06 is adapted to use i instead of t when accessing the global input array. The for-loop from Fig. 10.11 is used without change. This is because each block has its own private input_s in the shared memory, so it can be accessed with t = threadIdx.x as if the segment is the entire input.

Once the reduction tree for-loop is complete, the partial sum for the segment is in input_s[0]. The if-statement in line 16 of Fig. 10.13 selects thread 0 to contribute the value in input_s[0] to output, as illustrated in the bottom part of Fig. 10.12. This is done with an atomic add, as shown in line 14 of Fig. 10.13. Once all blocks of the grid have completed execution, the kernel will return, and the total sum is in the memory location pointed to by output.

10.8 Thread coarsening for reduced 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 thread blocks. As we saw in earlier sections, hardware underutilization increases with each successive stage of the reduction tree because of more warps becoming idle and the final warp experiencing more control divergence. The phase in which the hardware is underutilized occurs for every thread block that we launch. It is an inevitable price to pay if the thread blocks are to actually run in parallel. However, if the hardware is to serialize these thread blocks, we are better off serializing them ourselves in a more efficient manner. As we discussed in Chapter 6, Performance Considerations, thread granularity coarsening, or

thread coarsening for brevity, is a category of optimizations that serialize 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. 10.14 illustrates how thread coarsening can be applied to the example in Fig. 10.10. In Fig. 10.10, each thread block received 16 elements, which is two elements per thread. Each thread independently adds the two elements for which it is responsible; then the threads collaborate to execute a reduction tree. In Fig. 10.14 we coarsen the thread block by a factor of 2. Hence each thread block receives twice the number of elements, that is, 32 elements, which is four elements per thread. In this case, each thread independently adds four elements 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. 10.14. Note that all threads are active during these three steps. Moreover, since the threads independently add the four elements for which they are responsible, they do not need to synchronize, and they do not need to store their partial sums to shared memory until after all four elements have been added. The remaining steps in performing the reduction tree are the same as those in Fig. 10.10.

Fig. 10.15 shows the kernel code for implementing reduction with thread coarsening for the multiblock segmented kernel. Compared to Fig. 10.13, the kernel has two main 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 for which the thread is responsible, rather than just adding two elements (line 06 in Fig. 10.13), we use a coarsening loop to iterate over the elements and add them based on

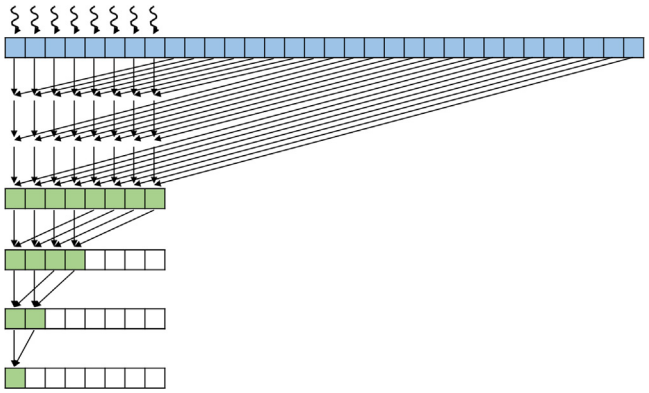


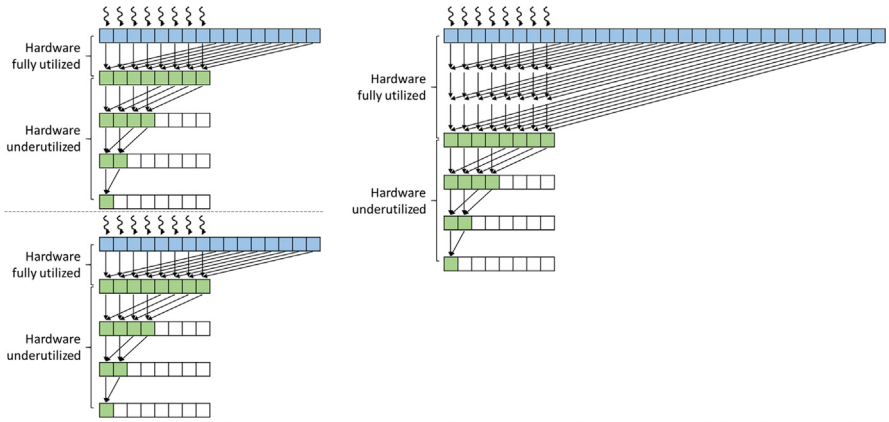
FIGURE 10.14

Thread coarsening in reduction.

```
01    __global__ CoarsenedSumReductionKernel(float* input, float* output) {
02        __shared__ float input_s[BLOCK_DIM];
03        unsigned int segment = COARSE_FACTOR*2*blockDim.x*blockIdx.x;
04        unsigned int i = segment + threadIdx.x;
05        unsigned int t = threadIdx.x;
06        float sum = input[i];
07        for(unsigned int tile = 1; tile < COARSE_FACTOR*2; ++tile) {
08            sum += input[i + tile*BLOCK_DIM];
09        }
10        input_s[t] = sum;
11        for (unsigned int stride = blockDim.x/2; stride >= 1; stride /= 2){
12            __syncthreads();
13            if (t < stride) {
14                input_s[t] += input_s[t + stride];
15            }
16        }
17        if (t == 0) {
18              atomicAdd(output, input_s[0]);
19          }
20    }
```

FIGURE 10.15

Sum reduction kernel with thread coarsening.



(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 10.16

Comparing parallel reduction with and without thread coarsening.

`COARSE_FACTOR` (lines 06–09 in Fig. 10.15). Note that all threads are active throughout this coarsening loop, the partial sum is accumulated to the local variable `sum`, and no calls to `__syncthreads()` are made in the loop because the threads act independently.

Fig. 10.16 compares the execution of two original thread blocks without coarsening serialized by the hardware, shown in Fig. 10.16A with one coarsened thread block performing the work of two thread blocks, shown in Fig. 10.16B. In

Fig. 10.16A the first thread block performs one step in which each thread adds the two elements for which it is responsible. All threads are active during this step, so the hardware is fully utilized. The remaining three steps execute the reduction tree in which 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, of which two steps fully utilize the hardware and six steps underutilize the hardware and require barrier synchronization and shared memory access.

By contrast, in Fig. 10.16B the same amount of data is processed by only a single thread block that is coarsened by a factor of 2. This thread block initially takes three steps in which each thread adds the four elements for which it is responsible. All threads are active during all three steps, so the hardware is fully utilized, and no barrier synchronizations or accesses to shared memory are performed. The remaining three steps execute the reduction tree in which half the threads drop out each step, underutilizing the hardware, and barrier synchronization and accesses to shared memory are needed. Overall, only six steps are needed (instead of eight), of which three steps (instead of two) fully utilize the hardware and 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 that is 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 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.

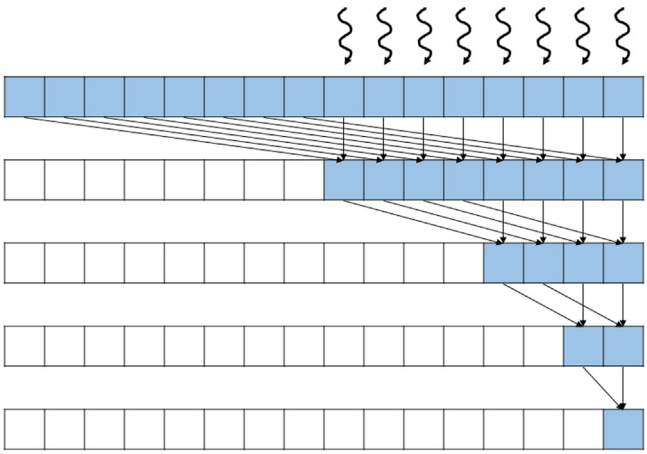
10.9 Summary

The parallel reduction pattern is important, as it plays a key row in many data-processing applications. Although the sequential code is simple, it should be clear to the reader that several techniques, such as thread index assignment for reduced divergence, using shared memory for reduced global memory accesses, segmented reduction with atomic operations, and thread coarsening, are needed to achieve high performance for large inputs. The reduction computation 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 of Chapter 11, Prefix Sum (Scan).

Exercises

1. For the simple reduction kernel in Fig. 10.6, if the number of elements is 1024 and the warp size is 32, how many warps in the block will have divergence during the fifth iteration?
2. For the improved reduction kernel in Fig. 10.9, if the number of elements is 1024 and the warp size is 32, how many warps will have divergence during the fifth iteration?
3. Modify the kernel in Fig. 10.9 to use the access pattern illustrated below.



4. Modify the kernel in Fig. 10.15 to perform a max reduction instead of a sum reduction.
5. Modify the kernel in Fig. 10.15 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 that represents the length of the input.
6. Assume that 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. 10.6 is used.

Initial array:

6	2	7	4	5	8	3	1

b. The kernel optimized for coalescing and divergence in Fig. 10.9 is used.

Initial array:

6	2	7	4	5	8	3	1

CHAPTER

11 Prefix sum (scan)

An introduction to work efficiency in parallel algorithms

With special contributions from Li-Wen Chang, Juan Gómez-Luna and John Owens

Chapter Outline

Our next parallel pattern is prefix sum, which is also commonly known as scan. 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 in which each item in a series is defined in terms of the previous item, it can likely be parallelized 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 radix sort, quick sort, string comparison, polynomial evaluation, solving recurrences, tree operations, and stream compaction. The radix sort example will be presented in Chapter 13, Sorting.

Another reason why parallel scan is an important parallel pattern is that it is a typical example of where the work performed by some parallel algorithms can have higher complexity than the work performed by a sequential algorithm, leading to a tradeoff that needs to be carefully made between algorithm complexity and parallelization. As we will show, a slight increase in algorithm complexity can make parallel

Programming Massively Parallel Processors. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00022-7

scan run more slowly than sequential scan for large datasets. Such consideration is becoming even more important in the age of "big data," in which massive datasets challenge tradition algorithms that have high computational complexity.

11.1 Background

Mathematically, an *inclusive scan* operation takes a binary associative operator $\oplus$ and an input array of n elements $[x_0, x_1, \ldots, x_{n-1}]$, and returns the following output array:

$$[x_0, \quad (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})]$$

For example, if $\oplus$ is addition, an inclusive scan operation on the input array [3 1 7 0 4 1 6 3] would return [2, 3+1, 3+1+7, 3+1+7+0, ..., 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.

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 eight people. Each person has ordered a different amount in terms of inches: 3, 1, 7, 0, 4, 1, 6, and 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 one-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 the 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:

$$[i, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})]$$

That is, each output element *excludes* the effect of the corresponding input element. The first output element is i, 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, 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, an 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. For another example, the section for Person 7 starts at the 22-inch point. The beginning point information is important in applications such as memory allocation, in which 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 elements and fill in an element. When converting from inclusive to exclusive, one can simply shift all elements to the right and fill in identity value for the 0th element. When converting from exclusive to inclusive, one needs to 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 present parallel algorithms and implementations only for inclusive scan.

Before we present parallel scan algorithms and their implementations, we would like to 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 `x` array and the output elements are to be written into the `y` array.

The code initializes the output element y[0] with the value of input element x[0] (line 02). 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 algorithm is $O(N)$.

In Sections 11.2–11.5, we will present alternative algorithms for performing parallel *segmented scan*, in which every thread block will perform a scan on a segment, that is, a section, of elements in the input array in parallel. We will then present in Sections 11.6 and 11.7 methods that combine the segmented scan results into the scan output for the entire input array.

```
01    void sequential_scan(float *x, float *y, unsigned int N) {
02        y[0] = x[0];
03        for(unsigned int i = 1; i < N; ++i) {
04            y[i] = y[i - 1] + x[i];
05        }
06    }
```

FIGURE 11.1

A simple sequential implementation of inclusive scan based on addition.

11.2 Parallel scan with the Kogge-Stone algorithm

We start with a simple parallel inclusive scan algorithm by performing a reduction operation for each output element. One might be tempted to use each thread to perform sequential reduction as shown in Fig. 10.2 for one output element. After all, this allows the calculations for all output elements to be performed in parallel. Unfortunately, this approach will be unlikely to improve the execution time over the sequential scan code in Fig. 11.1. This is because the calculation of y_{n-1} will take n steps, the same number of steps taken by the sequential scan code, and each step (iteration) in the reduction involves the same amount of work as each iteration of the sequential scan. Since the completion time of a parallel program is limited by the thread that takes the longest time, this approach is unlikely be any faster than sequential scan. In fact, with limited computing resources, the execution time of this naïve parallel scan algorithm can be much longer than that of the sequential algorithm. The computation cost, or the total number of operations performed, would unfortunately be much higher for the proposed approach. Since the number of reduction steps for output element i would be i, the total number of steps performed by all threads would be

$$\sum_{i=0}^{n-1} i \quad = \frac{n \cdot (n-1)}{2}$$

That is, the proposed approach has a computation 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 computational complexity means that considerably more execution resources need to be provisioned. This is obviously a bad idea.

A better approach is to adapt the parallel reduction tree in Chapter 10, Reduction and Minimizing Divergence, to calculate each output element with a reduction tree of the relevant input elements. 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 computational 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 (Kogge & Stone, 1973). This algorithm is still being used in the design of high-speed computer arithmetic hardware.

The algorithm, illustrated in Fig. 11.2, is an in-place scan algorithm that operates on an array XY that originally contains input elements. It iteratively evolves the contents of the array into output elements. Before the algorithm begins, we assume that XY[i] contains input element x_i. After k iterations, XY[i] will contain the sum of up to 2^k input elements at and before the location. For example, after one iteration, XY[i] will contain $x_{i-1}+x_i$ and at the end of iteration 2, XY[i] will contain $x_{i-3}+x_{i-2}+x_{i-1}+x_i$, and so on.

Fig. 11.2 illustrates the algorithm with a 16-element input example. Each vertical line represents an element of the XY array, with XY[0] in the 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 XY[0] contains its final answer. In the first iteration, each position other than XY[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, XY[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, XY[3] contains x_2+x_3, shown as

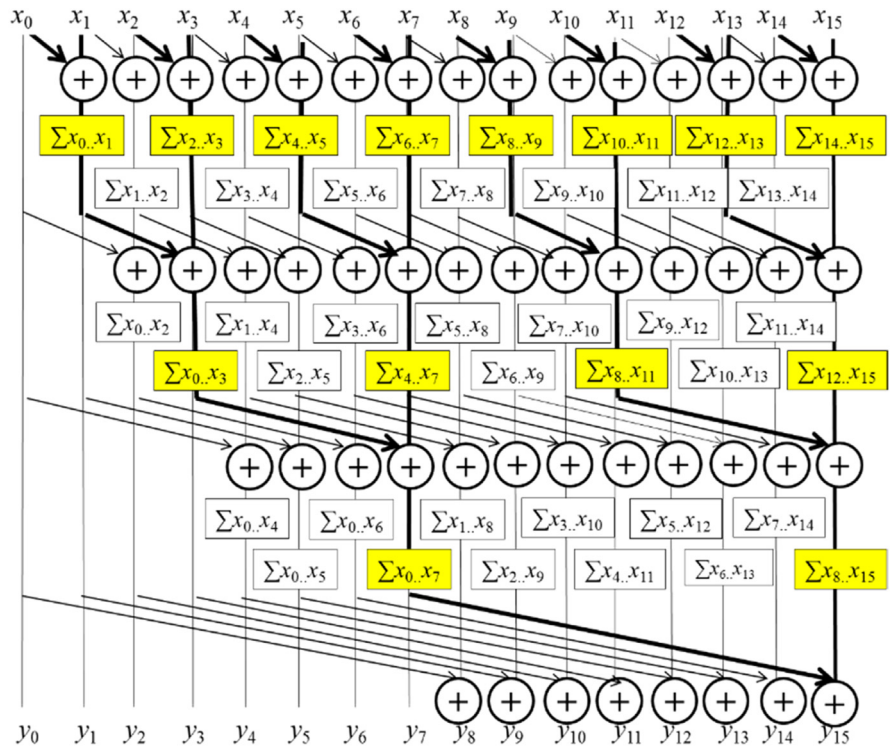


FIGURE 11.2

A parallel inclusive scan algorithm based on Kogge-Stone adder design.

$\sum x_2 \ldots x_3$. Note that after the first iteration, XY[1] is equal to x_0+x_1, which is the final answer for this position. So, there should be no further changes to XY[1] in subsequent iterations.

In the second iteration, each position other than XY[0] and XY[1] receives the sum of its current content and that of the position that is two elements away. This is illustrated in the labeled boxes below the second row of addition operators. As a result, XY[i] becomes $x_{i-3}+x_{i-2}+x_{i-1}+x_i$. For example, after the second iteration, XY[3] becomes $x_0+x_1+x_2+x_3$, shown as $\sum x_0 \ldots x_3$. Note that after the second iteration, XY[2] and XY[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 the parallel implementation of the algorithm illustrated in Fig. 11.2. We implement a kernel that performs local scans on different segments (sections) of the input that are each small enough for a single block to handle. Later, we will make final adjustments to consolidate these sectional scan results for large input arrays. The size of a section is defined as a compile-time constant `SECTION_SIZE`. We assume that the kernel function will be called using `SECTION_SIZE` as the block size, so there will be the same number of threads and section elements. We assign each thread to evolve the contents of one XY element.

The implementation shown in Fig. 11.3 assumes that input values were originally in a global memory array X, whose address is passed into the kernel as an argument (line 01). We will have all the threads in the block collaboratively load the X array elements into a shared memory array XY (line 02). This is done by having each thread calculate its global data index `i=blockIdx.x*blockDim.x+threadIdx.x` (line 03) for the output vector element position it is responsible for. Each thread loads the input element at

```
01    __global__ void Kogge_Stone_scan_kernel(float *X, float *Y, unsigned int N){
02        __shared__ float XY[SECTION_SIZE];
03      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) {
05          XY[threadIdx.x] = X[i];
06      } else {
07          XY[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 = XY[threadIdx.x] + XY[threadIdx.x-stride];
14            __syncthreads();
15          if(threadIdx.x >= stride)
16              XY[threadIdx.x] = temp;
17      }
18      if(i < N) {
19          Y[i] = XY[threadIdx.x];
20      }
21  }
```

FIGURE 11.3

A Kogge-Stone kernel for inclusive (segmented) scan.

that position into the shared memory at the beginning of the kernel (lines 04–08). At the end of the kernel, each thread will write its result into the assigned output array Y (lines 18–20).

We now focus on the implementation of the iterative calculations for each XY element in Fig. 11.3 as a for-loop (lines 09–17). The loop iterates through the reduction tree for the XY array position that is assigned to a thread. When the stride value becomes greater than a thread's `threadIdx.x` value, it means that the thread's assigned XY position has already accumulated all the required input values and the thread no longer needs to be active (lines 12 and 15). Note that we use a barrier synchronization (line 10) to make sure that all threads have finished their previous iteration before any of them starts the next iteration. This is the same use of `__syncthreads()` as in the reduction discussion in Chapter 10, Reduction and Minimizing Divergence.

There is, however, a very important difference compared to reduction in updating of the XY elements (lines 12–16) in each iteration of the for-loop. Note that each active thread first stores the partial sum for its position into a temp variable (in a register). After all threads have completed a second barrier synchronization (line 14), all of them store their partial sum values to their XY positions (line 16). The need for the extra temp and `__syncthreads()` has to do with a write-after-read data dependence hazard in these updates. Each active thread adds the XY value at its own position `(XY[threadIdx.x])` and that at a position of another thread `(XY[threadIdx.x-stride])`. If a thread *i* writes to its output position before another thread *i+stride* has had the chance to read the old value at that position, the new value can corrupt the addition performed by the other thread. The corruption may or may not occur, depending on the execution timing of the threads involved, which is referred to as a race condition. Note that this race condition is different from the one we saw in Chapter 9, Parallel Histogram, with the histogram pattern. The race condition in Chapter 9, Parallel Histogram, was a read-modify-write race condition that can be solved with atomic operations. For the write-after-read race condition that we see here, a different solution is required.

The race condition can be easily observed in Fig. 11.2. Let us examine the activities of thread 4 (x_4) and thread 6 (x_6) during iteration 2, which is represented as the addition operations in the second row from the top. Note that thread 6 needs to add the old value of XY[4] (x_3+x_4) to the old value XY[6] (x_5+x_6) to generate the new value of XY[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 XY[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 XY[6]. Since x_1+x_2 will be added again to XY[6] by thread 6 in the third iteration, the final answer in XY[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 XY[4] before thread 4 overwrites it during iteration 2, the results will 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 with the temporary variable 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 XY locations will be overwritten. The barrier `__syncthread( )` in line 14 ensures that all active threads have completed their read of the old XY 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 XY locations.

The reason why an updated XY position may be used by another active thread is that the Kogge-Stone approach reuses the partial sums across reduction trees to reduce the computational complexity. We will study this point further in Section 11.3. The reader might wonder why the reduction tree kernels in Chapter 10, Reduction and Minimizing Divergence, did not need to use temporary variables and an extra `__syncthreads()`. The answer is that there is no race condition caused by a write-after-read hazard in these reduction kernels. This is because 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 Fig. 10.7 and 10.8. For example, in Fig. 10.8, each active thread 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]`. Since the execution within a thread is always sequential, each thread will always be able to read the old value in `input[threadIdx.x]` before writing the new value into the position. The reader should verify that the same property holds in Fig. 10.7.

If we want to avoid having a second barrier synchronization on every iteration, another way to overcome the race condition is to use separate arrays for input and output. If separate arrays are used, the location that is being written to is different from the location that is being read from, so there is no longer any potential write-after-read race condition. This approach would require having two shared memory buffers instead of one. In the beginning, we load from the global memory to the first buffer. In the first iteration we read from the first buffer and write to the second buffer. After the iteration is over, the second buffer has the most up-to-date results, and the results in the first buffer are no longer needed. Hence in the second iteration we read from the second buffer and write to the first buffer. Following the same reasoning, in the third iteration we read from the first buffer and write to the second buffer. We continue alternating input/output buffers until the iterations complete. This optimization is called *double-buffering*. Double-buffering is commonly used in parallel programming as a way to overcome write-after-read race conditions. We leave the implementation of this optimization as an exercise for the reader.

Furthermore, as is shown in Fig. 11.2, the actions on the smaller positions of XY end earlier than those on the larger positions (see the if-statement condition). This will cause some level of control divergence in the first warp when stride values are small. Note that adjacent threads will tend to execute the same number of iterations. The effect of divergence should be quite modest for large block sizes, since divergence will arise only in the first warp. The detailed analysis is left as an exercise for the reader.

Although we have shown only an inclusive scan kernel, we can easily convert an inclusive scan kernel to an exclusive scan kernel. Recall that an exclusive scan is equivalent to an inclusive scan with all elements shifted to the right by one position and element 0 filled with the identity value. This is illustrated in Fig. 11.4. Note that the only real difference is the alignment of elements on top of the picture. All labeling boxes are updated to reflect the new alignment. All iterative operations remain the same.

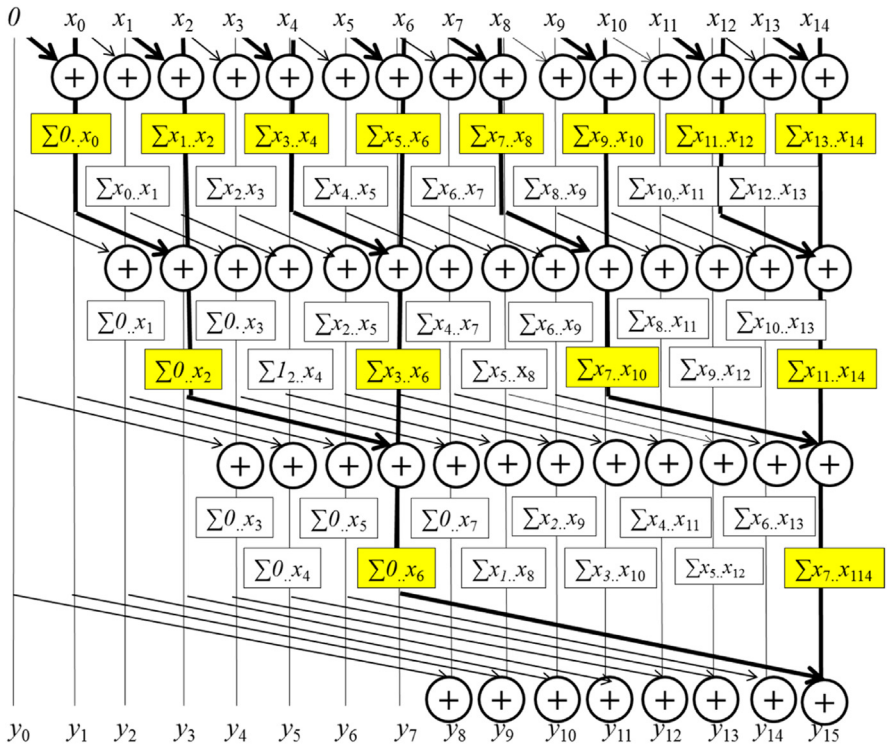


FIGURE 11.4

A parallel exclusive scan algorithm based on Kogge-Stone adder design.

We can now easily convert the kernel in Fig. 11.3 into an exclusive scan kernel. The only modification we need to make is to load 0 into XY[0] and `X[i-1]` into `XY[threadIdx.x]`, as shown in the following code:

```
if (i < N && threadIdx.x != 0) {
    XY[threadIdx.x] = X[i-1];
} else {
    XY[threadIdx.x] = 0.0f;
}
```

By substituting these four lines of code for lines 04–08 of Fig. 11.3, we convert the inclusive scan kernel into an exclusive scan kernel. We leave the work to finish the exclusive scan kernel as an exercise for the reader.

11.3 Speed and work efficiency consideration

One important consideration in analyzing parallel algorithms is *work efficiency*. The work efficiency of an algorithm refers to the extent to which the work that is performed by the algorithm is close to the minimum amount of work needed for the computation. 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 $N^*(N-1)/2$ additions, or $O(N^2)$, which is substantially larger than the sequential algorithm. For this reason, the naïve parallel algorithm is not work efficient.

We now analyze the work efficiency of the Kogge-Stone kernel in Fig. 11.3, focusing on the work of a single thread block. All threads iterate up to $\log_2 N$ steps, where N is the `SECTION_SIZE`. In each iteration the number of inactive threads is equal to the stride size. Therefore we can calculate the amount of work done (one iteration of the for-loop, represented by the add operation in Fig. 8.1) for the algorithm as

$$\sum_{stride}(N-\text{stride}),\ \text{for strides}\ 1,\ 2,\ 4,\ \ldots N/2\left(\log_2 N\ \text{terms}\right)$$

The first part of each term is independent of stride, and its summation adds up to $N^*\log_2(N)$. The second part is a familiar geometric series and sums up to $(N-1)$. So, the total amount of work done is

$$N^*\log_2(N)-(N-1)$$

The good news is that the computational complexity of the Kogge-Stone approach is $O(N^*\log_2(N))$, better than the $O(N^2)$ complexity of a naïve approach that performs complete reduction trees for all output elements. The bad news is that the Kogge-Stone algorithm is still not as work efficient as the sequential

algorithm. Even for modest-sized sections, the kernel in Fig. 11.3 does much more work than the sequential algorithm. In the case of 512 elements, the kernel does approximately eight times more work than the sequential code. The ratio will increase as N becomes larger.

Although the Kogge-Stone algorithm performs more computations than the sequential algorithm, it does so in fewer steps because of parallel execution. The for-loop of the sequential code executes N iterations. As for the kernel code, the for-loop of each thread executes up to $\log_2$N iterations, which defines the minimal number of steps needed for executing the kernel. 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×.

In a real CUDA GPU device, the amount of work done by the Kogge-Stone kernel is more than the theoretical $N^*\log2(N) - (N - 1)$. This is because we are using N threads. While many of the threads stop participating in the execution of the for-loop, some of them still consume execution resources until the entire warp completes execution. Realistically, the amount of execution resources consumed by the Kogge-Stone is closer to $N^*\log2(N)$.

We will use the concept of computation 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 $(N^*\log2(N))/P$ steps. If P is equal to N, that is, if we have enough execution units to process all input element in parallel, then we need $\log2(N)$ 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 × 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. Hence the parallel algorithm will be slower. Second, all the extra work consumes additional energy. This makes the kernel less appropriate for power-constrained environments such as mobile applications.

The strength of the Kogge-Stone kernel is that it can achieve very good execution speed when there is enough hardware resources. It is typically used to calculate the scan result for a section with a modest number of elements, such as 512 or 1024. This, of course, assumes that GPUs can provide sufficient hardware resources and use the additional parallelism to tolerate latencies. As we have seen, its execution has a very limited amount of control divergence. In newer GPU architecture generations its computation can be efficiently performed with shuffle instructions within warps. We will see later in this chapter that it is an important component of the modern high-speed parallel scan algorithms.

11.4 Parallel scan with the Brent-Kung algorithm

While the Kogge-Stone kernel in Fig. 11.3 is conceptually simple, its work efficiency is quite low for some practical applications. Just by inspecting Figs. 11.2 and 11.4, 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 is a reduction tree. With sufficient execution units, a reduction tree can generate the sum for N values in $\log_2(N)$ time units. 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 (Brent & Kung, 1979). The Brent-Kung adder design can also be used to implement a parallel scan algorithm with better work efficiency.

Fig. 11.5 illustrates the steps for a parallel inclusive scan algorithm based on the Brent-Kung adder design. In the top half of Fig. 11.5, we produce the sum of all 16 elements in four steps. We use the minimal number of operations needed to generate the sum. During the first step, only the odd element of XY[i] will be updated to XY[i-1] +XY[i]. During the second step, only the XY elements whose indices are of the form of $4^*n - 1$, which are 3, 7, 11, and 15 in Fig. 11.5, will be updated. During the third step, only the XY elements whose indices are of the form $8^*n - 1$, which are 7 and 15, will be updated. Finally, during the fourth step, only XY[15] is updated. The total number of operations performed is 8+4+2+1=15. In general, for a scan section of N elements, we would do $(N/2)+(N/4)+\ldots+2+1=N-1$ 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

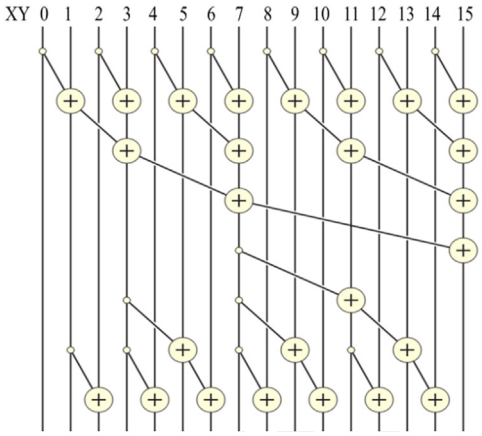


FIGURE 11.5

A parallel inclusive scan algorithm based on Brent-Kung adder design.

distribution of partial sums is illustrated in the bottom half of Fig. 11.5. 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 XY. It should be apparent from an inspection of Fig. 11.5 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 XY 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. 11.6 shows the state of each position (column), including both the values already accumulated into the position and 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 $x_i...x_j$, 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 XY[11] before the reverse tree begins (right after the reduction phase shown in the bottom portion of Fig. 11.5). 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, XY[0], XY[1], XY[3], XY[7], and XY[15] are all complete with their final answers.

The need for additional input element values is indicated by the shade of each cell in Fig. 11.6: White means that the position needs to accumulate partial sums from three other positions, light gray means 2, dark gray means 1, and black means 0. For example, initially, XY[14] is marked white 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 XY[7] ($x_0...x_7$), XY[11] ($x_8...x_{11}$), and XY[13] ($x_{12}...x_{13}$) to complete its final scan value ($x_0...x_{14}$). The reader should verify that because of the structure of the reduction tree, the XY positions for an input of size N elements will never need the accumulation from more than $\log_2(N) - 1$ partial sums from other XY positions. Furthermore, these partial sum positions will always be 1, 2, 4, ... (powers of 2) away from each other. In our example, XY[14] needs $\log_2(16) - 1=3$ partial sums from positions that are 1 (between XY[14] and XY[13]), 2 (between XY[13] and XY[11]), and 4 (between XY[11] and XY[7]).

To organize our second half of the addition operations, we will first show all the operations that need partial sums from four positions away, then two positions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Initial	x_0	$x_0..x_1$	x_2	$x_0..x_3$	x_4	$x_4..x_5$	x_6	$x_0..x_7$	x_8	$x_8..x_9$	x_{10}	$x_8..x_{11}$	x_{12}	$x_{12}..x_{13}$	x_{14}	$x_0..x_{15}$
Level 1												$x_0..x_{11}$				
Level 2						$x_0..x_5$				$x_0..x_9$				$x_0..x_{13}$		
Level 3			$x_0..x_2$		$x_0..x_4$		$x_0..x_6$		$x_0..x_8$		$x_0..x_{10}$		$x_0..x_{12}$		$x_0..x_{14}$	

FIGURE 11.6

Progression of values in XY after each level of additions in the reverse tree.

away, then one position away. During the first level of the reverse tree, we add XY[7] to XY[11], which brings XY[11] to the final answer. In Fig. 11.6, position 11 is the only one that advances to its final answer. During the second level, we complete XY[5], XY[9], and XY[13], which can be completed with the partial sums that are two positions away: XY[3], XY[7], and XY[11], respectively. Finally, during the third level, we complete all even positions XY[2], XY[4], XY[6], XY[8], XY[10], and XY[12] by accumulating the partial sums that are one position away (immediate left neighbor of each position).

We are now ready to implement the Brent-Kung approach to scan. We could implement the reduction tree phase of the parallel scan using the following loop:

```
for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __synchthreads();
    if ((threadIdx.x + 1)%(2*stride) == 0) {
        XY[threadIdx.x] += XY[threadIdx.x - stride];
    }
}
```

Note that this loop is similar to the reduction in Fig. 10.6. There are only two differences. The first difference is that we accumulate the sum value toward the highest position, which is `XY[blockDim.x-1]`, rather than XY[0]. This is because the final result of the highest position is the total sum. For this reason, each active thread reaches for a partial sum to its left by subtracting the stride value from its index. The second difference is that we want the active threads to have a thread index of the form $2^n - 1$ rather than 2^n. This is why we add 1 to the threadIdx.x before the modulo (%) operation when we select the threads for performing addition in each iteration.

One drawback of this style of reduction is that it has significant control divergence problems. As we saw in Chapter 10, Reduction and Minimizing Divergence, a better way is to use a decreasing number of contiguous threads to perform the additions as the loop advances. Unfortunately, the technique we used to reduce divergence in Fig. 10.8 cannot be used in the scan reduction tree phase, since it does not generate the needed partial sum values in the intermediate XY positions. Therefore we resort to a more sophisticated thread index to data index mapping that maps a continuous section of threads to a series of data positions that are of stride distance apart. The following code does so by mapping a continuous section of threads to the XY positions whose indices are of the form $k*2^n - 1$:

```
for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1)*2*stride - 1;
    if(index < SECTION_SIZE) {
        XY[index] += XY[index - stride];
    }
}
```

By using such a complex index calculation in each iteration of the for-loop, a contiguous set of threads starting from thread 0 will be used in every iteration to avoid control divergence within warps. In our small example in Fig. 11.5, there are 16 threads in the block. In the first iteration, stride is equal to 1. The first eight consecutive threads in the block will satisfy the if-condition. The XY index values calculated for these threads will be 1, 3, 5, 7, 9, 11, 13, and 15. These threads will perform the first row of additions in Fig. 11.5. In the second iteration, stride is equal to 2. Only the first four threads in the block will satisfy the if-condition. The index values calculated for these threads will be 3, 7, 11, and 15. These threads will perform the second row of additions in Fig. 11.5. Note that since each iteration will always be using consecutive threads, the control divergence problem does not arise until the number of active threads drops below the warp size.

The reverse tree is a little more complex to implement. We see that the stride value decreases from `SECTION_SIZE`/4 to 1. In each iteration we need to "push" the value of XY elements from positions that are multiples of twice the stride value minus 1 to the right by stride positions. For example, in Fig. 11.5 the stride value decreases from 4 (2^2) down to 1. In the first iteration we would like to push (add) the value of XY[7] into XY[11], where 7 is $2*2^2 - 1$ and distance (stride) is 2^2. Note that only thread 0 will be used for this iteration, since the index calculated for other threads will be too large to satisfy the if-condition. In the second iteration we push the values of XY[3], XY[7], and XY[11] to XY[5], XY[9], and XY[13], respectively. The indices 3, 7, and 11 are $1*2*2^1 - 1$, $2*2*2^1 - 1$, and $3*2*2^1 - 1$, respectively. The destination positions are 2^1 positions away from the source positions. Finally, in the third iteration we push the values at all the odd positions to their even position right-hand neighbor (stride=2°).

On the basis of the discussions above, the reverse tree can be implemented with the following loop:

```
for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
    __syncthreads();
    int index = (threadIdx.x + 1)*stride*2 - 1;
    if(index + stride < SECTION_SIZE) {
        XY[index + stride] += XY[index];
    }
}
```

The calculation of index is similar to that in the reduction tree phase. The `XY[index+stride] +=XY[index]` statement reflects the push from the threads' mapped location into a higher position that is stride away.

The final kernel code for a Brent-Kung parallel scan is shown in Fig. 11.7. The reader should notice that we never need to have more than `SECTION_SIZE`/2 threads for either the reduction phase or the distribution phase. Therefore we could simply launch a kernel with `SECTION_SIZE`/2 threads in a block. Since we

```
01    __global__ void Brent_Kung_scan_kernel(float *X, float *Y, unsigned int N) {
02        __shared__ float XY[SECTION_SIZE];
03      unsigned int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
04      if(i < N) XY[threadIdx.x] = X[i];
05      if(i + blockDim.x < N) XY[threadIdx.x + blockDim.x] = X[i + blockDim.x];
06      for(unsigned int stride = 1; stride <= blockDim.x; stride *= 2) {
07            __syncthreads();
08          unsigned int index = (threadIdx.x + 1)*2*stride - 1;
09          if(index < SECTION_SIZE) {
10              XY[index] += XY[index - stride];
11          }
12      }
13      for (int stride = SECTION_SIZE/4; stride > 0; stride /= 2) {
14            __syncthreads();
15          unsigned int index = (threadIdx.x + 1)*stride*2 - 1;
16          if(index + stride < SECTION_SIZE) {
17              XY[index + stride] += XY[index];
18          }
19      }
20      __syncthreads();
21      if (i < N) Y[i] = XY[threadIdx.x];
22      if (i + blockDim.x < N) Y[i + blockDim.x] = XY[threadIdx.x + blockDim.x];
23  }
```

FIGURE 11.7

A Brent-Kung kernel for inclusive (segmented) scan.

can have up to 1024 threads in a block, each scan section can have up to 2048 elements. However, we will need to have each thread load two X elements at the beginning and store two Y elements at the end.

As was in the case of the Kogge-Stone scan kernel, one can easily adapt the Brent-Kung inclusive parallel scan kernel into an exclusive scan kernel with a minor adjustment to the statement that loads X elements into XY. Interested readers should also read Harris et al., 2007, for an interesting natively exclusive scan kernel that is based on a different way of designing the reverse tree phase of the scan kernel.

We now turn our attention to the analysis of the number of operations in the reverse tree stage. The number of operations is $(16/8) - 1 + (16/4) - 1 + (16/2) - 1$. In general, for N input elements the total number of operations would be $(2 - 1) + (4 - 1) + \ldots + (N/4 - 1) + (N/2 - 1)$, which is $N - 1 - \log_2(N)$. Therefore the total number of operations in the parallel scan, including both the reduction tree ($N - 1$ operations) and the reverse tree ($N - 1 - \log_2(N)$ operations) phases, is $2^*N - 2 - \log_2(N)$. Note that the total number of operations is now $O(N)$, as opposed to $O(N^*\log_2 N)$ for the Kogge-Stone algorithm.

The advantage of the Brent-Kung algorithm over the Kogge-Stone algorithm is quite clear. As the input section becomes bigger, the Brent-Kung algorithm never performs more than 2 times the number of operations performed by the sequential algorithm. In an energy-constrained execution environment the Brent-Kung algorithm strikes a good balance between parallelism and efficiency.

While the Brent-Kung algorithm has a much higher level of theoretical work efficiency than the Kogge-Stone algorithm, its advantage in a CUDA kernel implementation is more limited. Recall that the Brent-Kung algorithm is using $N/2$ threads. The major difference is that the number of active threads drops much

faster through the reduction tree than the Kogge-Stone algorithm. However, some of the inactive threads may still consume execution resources in a CUDA device because they are bound to other active threads by SIMD. This makes the advantage in work efficiency of Brent-Kung over Kogge-Stone less drastic in a CUDA device.

The main disadvantage of Brent-Kung over Kogge-Stone is its potentially longer execution time despite its higher work efficiency. With infinite execution resources, Brent-Kung should take about twice as long as Kogge-Stone, owing to the need for additional steps to perform the reverse tree phase. However, the speed comparison can be quite different when we have limited execution resources. Using the example in Section 11.3, if we process 1024 input elements with 32 execution units, the Brent-Kung kernel is expected to take approximately (2*1024 − 2 − 10)/32=63.6 steps. The reader should verify that with control divergence there will be about five more steps when the total number of active threads drops below 32 in each phase. This results in a speedup of 1024/73.6=14 over sequential execution. This is in comparison with the 320 time units and speedup of 3.2 for Kogge-Stone. The comparison would be to Kogge-Stone's advantage, of course, when there are many more execution resources and/or when the latencies are much longer.

11.5 Coarsening for even more work efficiency

The overhead of parallelizing scan across multiple threads is like reduction in that it includes the hardware underutilization and synchronization overhead of the tree execution pattern. However, scan has an additional parallelization overhead, and that is reduced work efficiency. As we have seen, parallel scan is less work efficient than sequential scan. This lower work efficiency is an acceptable price to pay to parallelize if the threads were actually to run in parallel. However, if the hardware were to serialize them, we would be better off serializing them ourselves via thread coarsening to improve work efficiency.

We can design a parallel segmented scan algorithm that achieves even better work efficiency by adding a phase of fully independent sequential scans on subsections of the input. Each thread block receives a section of the input that is larger than the original section by the coarsening factor. At the beginning of the algorithm we partition the block's section of the input into multiple contiguous subsections: one for each thread. The number of subsections is the same as the number of threads in the thread block.

The coarsened scan is divided into three phases as shown in Fig. 11.8. During the first phase, we have each thread perform a sequential scan on its contiguous subsection. For example, in Fig. 11.8 we assume that there are four threads in a block. We partition the 16-element input section into four subsections with four elements each. Thread 0 will perform scan on its section (2, 1, 3, 1) and generate (2, 3, 6, 7). Thread 1 will perform scan on its section (0, 4, 1, 2) and generate (0, 4, 5, 7), and so on.

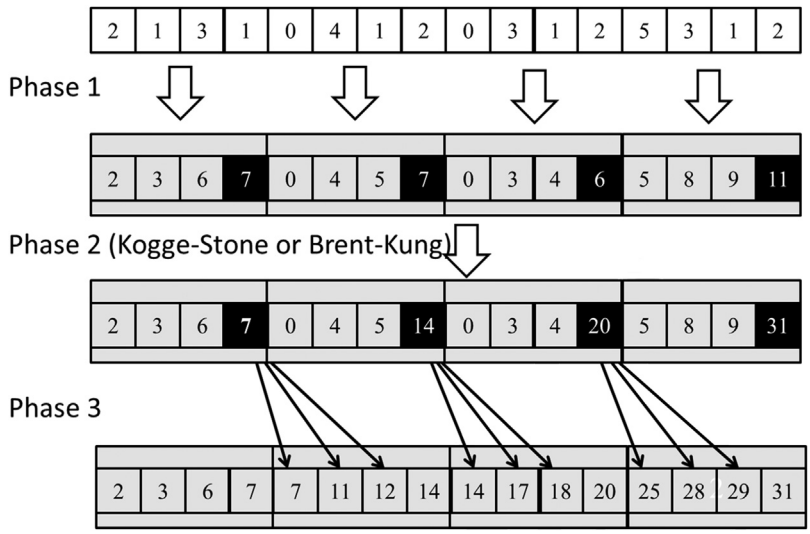


FIGURE 11.8

A three-phase parallel scan for higher work efficiency.

Note that if each thread directly performs scan by accessing the input from global memory, their accesses would not be coalesced. For example, during the first iteration, thread 0 would be accessing input element 0, thread 1 would be accessing input element 4, and so on. Therefore we improve memory coalescing by using shared memory to absorb the memory accesses that cannot be coalesced, as was mentioned in Chapter 6, Performance Considerations. That is, we transfer data between shared memory and global memory in a coalesced manner and perform the memory accesses with the unfavorable access pattern in shared memory. At the beginning of the first phase, all threads collaborate to load the input into the shared memory in an iterative manner. In each iteration, adjacent threads load adjacent elements to enable memory coalescing. For example, in Fig. 11.8 we have all threads collaborate and load four elements in a coalesced manner: Thread 0 loads element 0, thread 1 loads element 1, and so on. All threads then move to load the next four elements: Thread 0 loads element 4, thread 1 loads element 5, and so on.

Once all input elements are in the shared memory, the threads access their own subsection from the shared memory and perform a sequential scan on it. This is shown as Phase 1 in Fig. 11.8. Note that at the end of Phase 1 the last element of each section (highlighted as black in the second row) contains the sum of all input elements in the section. For example, the last element of section 0 contains value 7, the sum of the input elements (2, 1, 3, 1) in the section.

During the second phase, all threads in each block collaborate and perform a scan operation on a logical array that consists of the last element of each section.

This can be done with a Kogge-Stone or Brent-Kung algorithm, since there are only a modest number of elements (number of threads in a block). Note that the thread-to-element mappings need to be slightly modified from those in Figs. 11.3 and 11.7, since the elements that need to be scanned are of stride (four elements in Fig. 11.8) distance away from each other.

In the third phase, each thread adds the new value of the last element of its predecessor's section to its elements. The last elements of each subsection do not need to be updated during this phase. For example, in Fig. 11.8, thread 1 adds the value 7 to elements (0, 4, 5) in its section to produce (7, 11, 12). The last element of the section is already the correct value 14 and does not need to be updated.

With this three-phase approach, we can use a much smaller number of threads than the number of elements in a section. The maximal size of the section is no longer limited by the number of threads one can have in a block but rather by the size of the shared memory; all elements of the section need to fit into the shared memory.

The major advantage of thread coarsening for scan is its efficient use of execution resources. Assume that we use Kogge-Stone for phase 2. For an input list of N elements, if we use T threads, the amount of work done by each phase is $N - T$ for phase 1, $T^*\log_2 T$ for phase 2, and $N - T$ for phase 3. If we use P execution units, we can expect that the execution will take $(N - T + T^*\log_2 T + N - T)/P$ steps. For example, if we use 64 threads and 32 execution units to process 1024 elements, the algorithm should take approximately $(1024 - 64 + 64^*6 + 1024 - 64)/32 = 72$ steps. We leave the implementation of the coarsened scan kernel as an exercise for the reader.

11.6 Segmented parallel scan for arbitrary-length inputs

For many applications the number of elements to be processed by a scan operation can be in the millions or even billions. The kernels that we have presented so far perform local block-wide scans on sections of the input, but we still need a way to consolidate the results from different sections. To do so, we can use a hierarchical scan approach, as illustrated in Fig. 11.9.

For a large dataset we first partition the input into sections so that each of them can fit into the shared memory of a streaming multiprocessor and be processed by a single block. Assume that we call one of the kernels in Figs. 11.3 and 11.7 on a large input dataset. At the end of the grid execution, the Y array will contain the scan results for individual sections, called *scan blocks* in Fig. 11.9. Each element in a scan block only contains the accumulated values of all preceding elements in the same scan block. These scan blocks need to be combined into the final result; that is, we need to call another kernel that adds the sum of all elements in preceding scan blocks to each element of a scan block.

Fig. 11.10 shows a small example of the hierarchical scan approach of Fig. 11.9. In this example there are 16 input elements that are divided into four

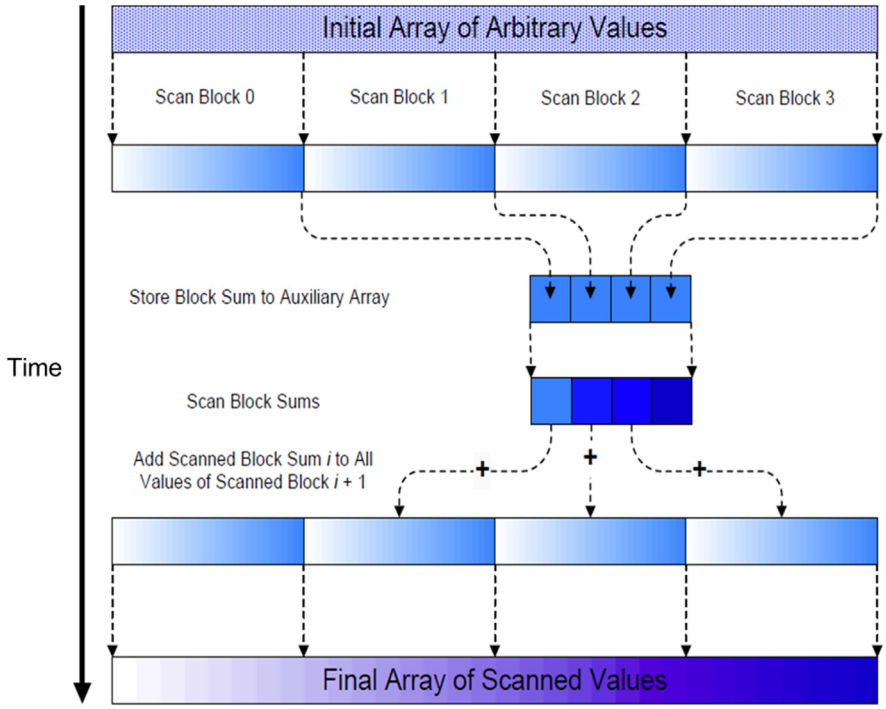


FIGURE 11.9

A hierarchical scan for arbitrary length inputs.

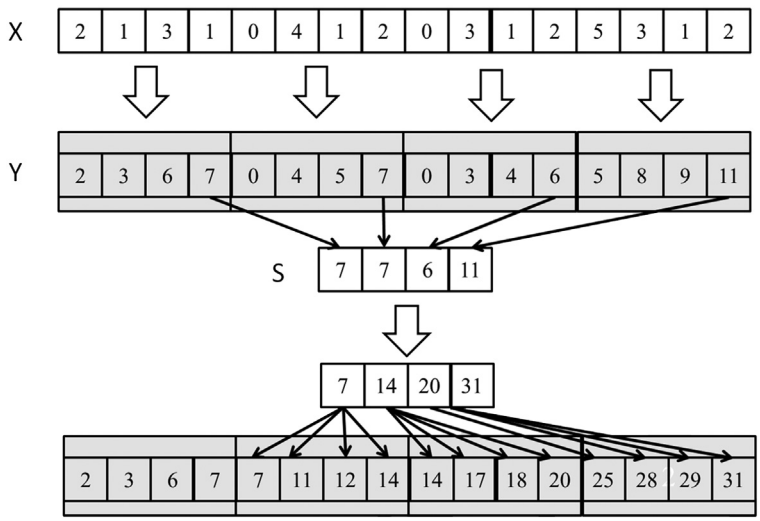


FIGURE 11.10

An example of hierarchical scan.

scan blocks. We can use the Kogge-Stone kernel, the Brent-Kung kernel, or a coarsened kernel to process the individual scan blocks. The kernel treats the four scan blocks as independent input datasets. After the scan kernel terminates, each Y element contains the scan result within its scan block. For example, scan block 1 has inputs 0, 4, 1, 2. The scan kernel produces the scan result for this section, which is 0, 4, 5, 7. Note that these results do not contain the contributions from any of the elements in scan block 0. To produce the final result for this scan block, the sum of all elements in scan block 0, that is, 2+1+3+1=7, should be added to every result element of scan block 1. For another example the inputs in scan block 2 are 0, 3, 1, 2. The kernel produces the scan result for this scan block, which is 0, 3, 4, 6. To produce the final results for this scan block, the sum of all elements in both scan block 0 and scan block 1, that is, 2+1+3+1+0+4+1+2=14, should be added to every result element of scan block 2.

It is important to note that the last output element of each scan block gives the sum of all input elements of the scan block. These values are 7, 7, 6, and 11 in Fig. 11.10. This brings us to the second step of the segmented scan algorithm in Fig. 11.9, which gathers the last result elements from each scan block into an array and performs a scan on these output elements. This step is also illustrated in Fig. 11.10, where the last scan output elements of all scan blocks are collected into a new array S. While the second step of Fig. 11.10 is logically the same as the second step of Fig. 11.8, the main difference is that Fig. 11.10 involves threads from different thread blocks. As a result, the last element of each section needs to be collected (written) into a global memory array so that they can be visible across thread blocks.

Gathering the last result of each scan block can be done by changing the code at the end of the scan kernel so that the last thread of each block writes its result into an S array using its blockIdx.x as the array index. A scan operation is then performed on S to produce output values 7, 14, 20, 31. Note that each of these second-level scan output values are the accumulated sum from the beginning location X[0] to the end location of each scan block. That is, the value in S[0]=7 is the accumulated sum from X[0] to the end of scan block 0, which is X[3]. The value in S[1]=14 is the accumulated sum from X[0] to the end of scan block 1, which is X[7]. Therefore the output values in the S array give the scan results at "strategic" locations of the original scan problem. In other words, in Fig. 11.10 the output values in S[0], S[1], S[2], and S[3] give the final scan results for the original problem at positions X[3], X[7], X[11], and X[15], respectively. These results can be used to bring the partial results in each scan block to their final values.

This brings us to the last step of the segmented scan algorithm in Fig. 11.10. The second-level scan output values are added to the values of their corresponding scan blocks. For example, in Fig. 11.10, the value of S[0] (value 7) will be added to Y[0], Y[1], Y[2], Y[3] of thread block 1, which completes the results in these positions. The final results in these positions are 7, 11, 12, 14. This is because S[0] contains the sum of the values of the original input X[0] through

X[3]. These final results are 14, 17, 18, and 20. The value of S[1] (14) will be added to Y[8], Y[9], Y[10], Y[11], which completes the results in these positions. The value of S[2] (20) will be added to Y[12], Y[13], Y[14], Y[15]. Finally, the value of S[3] is the sum of all elements of the original input, which is also the final result in Y[15].

Readers who are familiar with computer arithmetic algorithms should recognize that the principle behind the segmented scan algorithm is quite similar to the principle behind carry look-ahead in hardware adders of modern processors. This should be no surprise, considering that the two parallel scan algorithms that we have studied so far are also based on innovative hardware adder designs.

We can implement the segmented scan with three kernels. The first kernel is largely the same as the three-phase kernel. (We could just as easily use the Kogge-Stone kernel or the Brent-Kung kernel.) We need to add one more parameter S, which has the dimension of N/`SECTION_SIZE`. At the end of the kernel, we add a conditional statement for the last thread in the block to write the output value of the last XY element in the scan block to the blockIdx.x position of S:

```
__syncthreads();
if (threadIdx.x == blockDim.x - 1) {
    S[blockIdx.x] = XY[SECTION_SIZE - 1];
}
```

The second kernel is simply one of the three parallel scan kernels configured with a single thread block, which takes S as input and writes S as output without producing any partial sums.

The third kernel takes the S array and Y array as inputs and writes its output back into Y. Assuming that we launch the kernel with `SECTION_SIZE` threads in each block, each thread adds one of the S elements (selected by the `blockIdx.x-1`) to one Y element:

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
Y[i] += S[blockIdx.x - 1];
```

In other words, the threads in a block add the sum of all previous scan blocks to the elements of their scan block. We leave it as an exercise for the reader to complete the details of each kernel and complete the host code.

11.7 Single-pass scan for memory access efficiency

In the segmented scan mentioned in Section 11.6, the partially scanned results (scan blocks) are stored into global memory before launching the global scan

kernel, and then reloaded back from the global memory by the third kernel. The time for performing these extra memory stores and loads is not overlapped with the computation in the subsequent kernels and can significantly affect the speed of the segmented scan algorithms. To avoid such a negative impact, multiple techniques have been proposed (Dotsenko et al., 2008; Merrill & Garland, 2016; Yan et al., 2013). In this chapter a stream-based scan algorithm is discussed. The reader is encouraged to read the references to understand the other techniques.

In the context of CUDA C programming, a stream-based scan algorithm (not to be confused with CUDA streams to be introduced in Chapter 20, Programming a Heterogeneous Computing Cluster), or domino-style scan algorithm, refers to a segmented scan algorithm in which partial sum data is passed in one direction through the global memory between adjacent thread blocks in the same grid. Stream-based scan builds on a key observation that the global scan step (the middle part of Fig. 11.9) can be performed in a domino fashion and does not truly require a grid-wide synchronization. For example, in Fig. 11.10, scan block 0 can pass its partial sum value 7 to scan block 1 and complete its job. scan block 1 receives the partial sum value 7 from scan block 0, sums up with its local partial sum value 7 to get 14, passes its partial sum value 14 to scan block 2, and then completes its final step by adding 7 to all partial scan values in its scan block. This process continues for all thread blocks.

To implement a domino-style scan algorithm, one can write a single kernel to perform all three steps of the segmented scan algorithm in Fig. 11.9. Thread block i first performs a scan on its scan block, using one of the three parallel algorithms we presented in Sections 11.2 through 11.5. It then waits for its left neighbor block, $i-1$, to pass the sum value. Once it receives the sum from block $i-1$, it adds the value to its local sum and passes the cumulative sum value to its right neighbor block, i+1. It then moves on to add the sum value received from block $i-1$ to all partial scan values to produce all the output values of the scan block.

During the first phase of the kernel, all blocks can execute in parallel. They will be serialized during the data-passing phase. However, as soon as each block receives the sum value from its predecessor, it can perform its final phase in parallel with all other blocks that have received the sum values from their predecessors. As long as the sum values can be passed through the blocks quickly, there can be ample parallelism among blocks during the third phase.

To make this domino-style scan work, adjacent (block) synchronization is needed (Yan et al., 2013). Adjacent synchronization is a customized synchronization to allow the adjacent thread blocks to synchronize and/or exchange data. Particularly, in scan, the data are passed from scan block $i-1$ to scan block i, as in a producer-consumer chain. On the producer side (scan block $i-1$), a flag is set to a particular value after the partial sum has been stored to memory, while on the consumer side (scan block i), the flag is checked to see whether it is that particular value before the passed partial sum is loaded. As has been mentioned, the loaded value is further added with the local sum and then is passed to the next block (scan block i+1). Adjacent synchronization can be implemented by using

atomic operations. The following code segment illustrates the use of atomic operations to implement adjacent synchronization:

```
__shared__ float previous_sum;
if (threadIdx.x == 0){
    // Wait for previous flag
    while(atomicAdd(&flags[bid], 0) == 0) { }
    // Read previous partial sum
    previous_sum = scan_value[bid];
    // Propagate partial sum
    scan_value[bid + 1] = previous_sum + local_sum;
    // Memory fence
    __threadfence();
    // Set flag
    atomicAdd(&flags[bid + 1], 1);
}
__syncthreads();
```

This code section is executed by only one leader thread in each block (e.g., thread with index 0). The rest of threads will wait at `__syncthreads()` in the last line. In block bid, the leader thread checks flags[bid], a global memory array, repeatedly until it is set. Then it loads the partial sum from its predecessor by accessing the global memory array `scan_value[bid]` and stores the value into its local shared memory variable `previous_sum`. It adds the `previous_sum` with its local partial sum `local_sum` and stores the result into the global memory array `scan_value[bid+1]`. The memory fence function `__threadfence()` is required to ensure that the `scan_value[bid+1]` value arrives to the global memory before the flag is set with `atomicAdd()`.

Although it may appear that the atomic operations on the flags array and the accesses to the `scan_value` array incur global memory traffic, these operations are mostly performed in the second-level caches of recent GPU architectures (Chapter 9, Parallel Histogram). Any such stores and loads to the global memory will likely be overlapped with the phase 1 and phase 3 computational activities of other blocks. On the other hand, when executing the three-kernel segmented scan algorithm in Section 11.5, the stores to and loads from the S array in the global memory are in a separate kernel and cannot be overlapped with phase 1 or phase 3.

There is one subtle issue with domino-style algorithms. In GPUs, thread blocks may not always be scheduled linearly according to their blockIdx values, which means that scan block i may be scheduled and performed after scan block $i+1$. In this situation the execution order arranged by the scheduler might contradict the execution order assumed by the adjacent synchronization code and cause performance degradation or even a deadlock. For example, the scheduler may schedule scan block i through scan block $i+N$ before it schedules scan block $i-1$. If scan block i through scan block $i+N$ occupy all the streaming multiprocessors, scan block $i-1$ would not be able to start execution until at least one of

them finishes execution. However, all of them are waiting for the sum value from scan block $i - 1$. This causes the system to deadlock.

There are multiple techniques to resolve this issue (Gupta et al., 2012; Yan et al., 2013). Here, we discuss only one particular method, dynamic block index assignment, and leave the rest as a reference for readers. Dynamic block index assignment decouples the assignment of the thread block index from the built-in `blockIdx.x`. In the single-pass scan, the value of the bid variable of each block is no longer tied to the value of `blockIdx.x`. Instead, it is determined by using the following code at the beginning of the kernel:

```
__shared__ unsigned int bid_s;
if (threadIdx.x == 0) {
    bid_s = atomicAdd(blockCounter, 1);
}
__syncthreads();
unsigned int bid = bid_s;
```

The leader thread atomically increments a global counter variable pointed to by `blockCounter`. The global counter stores the dynamic block index of the next block that is scheduled. The leader thread then stores the acquired dynamic block index value into a shared memory variable `bid_s` so that it is accessible by all threads of the block after `__syncthreads()`. This guarantees that all scan blocks are scheduled linearly and prevents a potential deadlock. In other words, if a block obtains a bid value of i, then it is guaranteed that a block with value $i - 1$ has been scheduled because it has executed the atomic operation.

11.8 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 segmented scan algorithm that is fast and conceptually simple but not work-efficient. The algorithm performs $O(N^*\log_2 N)$ operations, which is more than its sequential counterpart. As the size of the dataset increases, the number of execution units that are 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 then presented a parallel Brent-Kung segmented scan algorithm that is conceptually more complicated. Using a reduction tree phase and a reverse tree phase, the algorithm performs only $2^*N - 3$, or $O(N)$, additions no matter how large the input datasets are. Such a work-efficient algorithm whose number of operations grows linearly with the size of the input set is often also referred to as a data-scalable algorithm. Although the Brent-Kung algorithm has better work efficiency than the Kogge-Stone algorithm, it requires more steps to complete. Therefore in a system with enough execution resources, the Kogge-Stone algorithm is expected to have better performance despite being less work efficient.

We also applied thread coarsening to mitigate the hardware underutilization and synchronization overhead of parallel scan and 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 subsection of input elements before the threads collaborate to perform the less work-efficient block-wide parallel scan to produce the entire block's section.

We presented a hierarchical scan approach to extend the parallel scan algorithms to handle input sets of arbitrary sizes. Unfortunately, a straightforward, three-kernel implementation of the segmented scan algorithm incurs redundant global memory accesses whose latencies are not overlapped with computation. For this reason we also presented a domino-style hierarchical scan algorithm to enable a single-pass, single-kernel implementation and improve the global memory access efficiency of the hierarchical scan algorithm. However, this approach requires a carefully designed adjacent block synchronization mechanism using atomic operations, thread memory fence, and barrier synchronization. Special care also must be taken to prevent deadlocks by using dynamic block index assignment.

There are further optimization opportunities for even higher performance implementations using, for example, warp-level shuffle operations. In general, implementing and optimizing parallel scan algorithms on GPUs are complex processes, and the average user is more likely to use a parallel scan library for GPUs such as Thrust (Bell and Hoberock, 2012) 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 tradeoffs that go into optimizing parallel patterns.

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. Modify the Kogge-Stone parallel scan kernel in Fig. 11.3 to use double-buffering instead of a second call to `__syncthreads()` to overcome the write-after-read race condition.

3. Analyze the Kogge-Stone parallel scan kernel in Fig. 11.3. Show that control divergence occurs only 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.
4. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 2048 elements. Which of the following gives the closest approximation of how many add operations will be performed?
5. 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.
6. For the Brent-Kung scan kernel, assume that we have 2048 elements. How many add operations will be performed in both the reduction tree phase and the inverse reduction tree phase?
7. Use the algorithm in Fig. 11.4 to complete an exclusive scan kernel.
8. Complete the host code and all three kernels for the segmented parallel scan algorithm in Fig. 11.9.

References

Bell, N., Hoberock, J., 2012. "Thrust: A productivity-oriented library for CUDA." GPU computing gems Jade edition. Morgan Kaufmann, 359–371.

Brent, R. P., & Kung, H. T., 1979. A regular layout for parallel adders, Technical Report, Computer Science Department, Carnegie-Mellon University.

Dotsenko, Y., Govindaraju, N.K., Sloan, P.P., Boyd, C., Manferdelli, J., 2008. Fast scan algorithms on graphics processors. Proc. 22nd Annu. Int. Conf. Supercomputing, 205–213.

Gupta, K., Stuart, J.A., Owens, J.D., 2012. A study of persistent threads style GPU programming for GPGPU workloads. Innovative Parallel Comput. (InPar) 1–14. IEEE.

Harris, M., Sengupta, S., & Owens, J. D. (2007). Parallel prefix sum with CUDA. GPU Gems 3. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf.

Kogge, P., Stone, H., 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Computers C-22, 783–791.

Merrill, D., & Garland, M. (2016, March). Single-pass parallel prefix scan with decoupled look-back. Technical Report NVR2016-001, NVIDIA Research.

Yan, S., Long, G., Zhang, Y., 2013. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization, PPoPP. ACM SIGPLAN Not. 48 (8), 229–238.

CHAPTER

Merge
An introduction to dynamic input data identification

12

With special contributions from Li-Wen Chang and Jie Lv

Chapter Outline

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 13, Sorting. Ordered merge operations also form the basis of modern map-reduce frameworks. This chapter presents a parallel ordered merge algorithm in which 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.

12.1 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. DOI: https://doi.org/10.1016/B978-0-323-91231-0.00025-2