David B. Kirk
Wen-mei W. Hwu

SECOND EDITION

Programming Massively Parallel Processors

A Hands-on Approach

MK
MORGAN KAUFMANN

nVIDIA.

Programming Massively Parallel Processors

This page intentionally left blank

Programming Massively Parallel Processors

A Hands-on Approach

Second Edition

David B. Kirk and Wen-mei W. Hwu

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER   BOOK AID International   Sabre Foundation

For information on all MK publications visit our website at www.mkp.com

Contents

This page intentionally left blank

Preface

We are proud to introduce the second edition of *Programming Massively Parallel Processors: A Hands-on Approach*. Mass-market computing systems that combine multicore computer processing units (CPUs) and many-thread GPUs have brought terascale computing to laptops and petascale computing to clusters. Armed with such computing power, we are at the dawn of pervasive use of computational experiments for science, engineering, health, and business disciplines. Many will be able to achieve breakthroughs in their disciplines using computational experiments that are of an unprecedented level of scale, accuracy, controllability, and observability. This book provides a critical ingredient for the vision: teaching parallel programming to millions of graduate and undergraduate students so that computational thinking and parallel programming skills will be as pervasive as calculus.

Since the first edition came out in 2010, we have received numerous comments from our readers and instructors. Many told us about the existing features they value. Others gave us ideas about how we should expand its contents to make the book even more valuable. Furthermore, the hardware and software technology for heterogeneous parallel computing has advanced tremendously. In the hardware arena, two more generations of graphics processing unit (GPU) computing architectures, Fermi and Kepler, have been introduced since the first edition. In the software domain, CUDA 4.0 and CUDA 5.0 have allowed programmers to access the new hardware features of Fermi and Kepler. Accordingly, we added eight new chapters and completely rewrote five existing chapters.

Broadly speaking, we aim for three major improvements in the second edition while preserving the most valued features of the first edition. The first improvement is to introduce parallel programming in a more systematic way. This is done by (1) adding new Chapters 8, 9, and 10 that introduce frequently used, basic parallel algorithm patterns; (2) adding more background material to Chapters 3, 4, 5, and 6; and (3) adding a treatment of numerical stability to Chapter 7. These additions are designed to remove the assumption that students are already familiar with basic parallel programming concepts. They also help to address the desire for more examples by our readers.

The second improvement is to cover practical techniques for using joint MPI-CUDA programming in a heterogeneous computing cluster. This has been a frequently requested addition by our readers. Due to the cost-effectiveness and high throughput per watt of GPUs, many high-performance computing systems now provision GPUs in each node. The new Chapter 19 explains the conceptual framework behind the programming interfaces of these systems.

The third improvement is an introduction of new parallel programming interfaces and tools that can significantly improve the productivity of data-parallel programming. The new Chapters 15, 16, 17, and 18 introduce OpenACC, Thrust,

CUDA FORTRAN, and C++AMP. Instead of replicating the detailed descriptions of these tools from their user guides, we focus on the conceptual understanding of the programming problems that these tools are designed to solve.

While we made all these improvements, we also preserved the first edition features that seem to contribute to its popularity. First, we kept the book as concise as possible. While it is very tempting to keep adding material, we want to minimize the number of pages readers need to go through to learn all the key concepts. Second, we kept our explanations as intuitive as possible. While it is extremely tempting to formalize some of the concepts, especially when we cover the basic parallel algorithms, we strive to keep all our explanations intuitive and practical.

Target Audience

The target audience of this book is graduate and undergraduate students from all science and engineering disciplines where computational thinking and parallel programming skills are needed to achieve breakthroughs. We assume that readers have at least some basic C programming experience. We especially target computational scientists in fields such as mechanical engineering, civil engineering, electrical engineering, bio-engineering, physics, chemistry, astronomy, and geography, who use computation to further their field of research. As such, these scientists are both experts in their domain as well as programmers. The book takes the approach of building on basic C programming skills, to teach parallel programming in C. We use CUDA C, a parallel programming environment that is supported on NVIDIA GPUs and emulated on CPUs. There are more than 375 million of these processors in the hands of consumers and professionals, and more than 120,000 programmers actively using CUDA. The applications that you develop as part of the learning experience will be able to run by a very large user community.

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 of the University of Illinois at Urbana-Champaign. We started to write up some 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 VSCSE and PUMPS summer schools.

A Three-Phased Approach

In ECE498AL the lectures and programming assignments are balanced with each other and organized into three phases:

Phase 1: One lecture based on Chapter 3 is dedicated to teaching the basic CUDA memory/threading model, the CUDA extensions to the C language, and the basic programming/debugging tools. After the lecture, students can write a simple vector addition code in a couple of hours. This is followed by a series of four lectures 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. These lectures are based on Chapters 4, 5, and 6.

Phase 2: A series of lectures covers floating-point considerations in parallel computing and common data-parallel programming patterns needed to develop a high-performance parallel application. These lectures are based on Chapters 7−10. The performance of their matrix multiplication codes increases by about 10 times through this period. The students also complete assignments on convolution, vector reduction, and prefix sum through this period.

Phase 3: Once the students have established solid CUDA programming skills, the remaining lectures cover application case studies, computational thinking, a broader range of parallel execution models, and parallel programming principles. These lectures are based on Chapters 11−20. (The voice and video recordings of these lectures are available online at the ECE408 web site: *http://courses.engr.illinois.edu/ece408/*.

Tying It All Together: The Final Project

While the lectures, labs, and chapters of this book help lay the intellectual foundation for the students, what brings the learning experience together is the final project. The final project is so important to the full-semester course that it is prominently positioned in the course and commands nearly two months' focus. It incorporates five innovative aspects: mentoring, workshop, clinic, final report, and symposium. (While much of the information about the final project is available at the ECE408 web site, we would like to offer the thinking that was behind the design of these aspects.)

Students are encouraged to base their final projects on problems that represent current challenges in the research community. To seed the process, the instructors should recruit several computational science research groups to propose problems and serve as mentors. The mentors are asked to contribute a one- to two-page project specification sheet that briefly describes the significance of the application, what the mentor would like to accomplish with the student teams on the application, the technical skills (particular type of math, physics, or chemistry courses) required to understand and work on the application, and a list of web

and traditional resources that students can draw upon for technical background, general information, and building blocks, along with specific URLs or FTP paths to particular implementations and coding examples. These project specification sheets also provide students with learning experiences in defining their own research projects later in their careers. (Several examples are available at the ECE408 course web site.)

Students are also encouraged to contact their potential mentors during their project selection process. Once the students and the mentors agree on a project, they enter into a close relationship, featuring frequent consultation and project reporting. The instructors should attempt to facilitate the collaborative relationship between students and their mentors, making it a very valuable experience for both mentors and students.

Project Workshop

The main vehicle for the whole class to contribute to each other's final project ideas is the project workshop. We usually dedicate six of the lecture slots to project workshops. The workshops are designed for students' benefit. For example, if a student has identified a project, the workshop serves as a venue to present preliminary thinking, get feedback, and recruit teammates. If a student has not identified a project, he or she can simply attend the presentations, participate in the discussions, and join one of the project teams. Students are not graded during the workshops, to keep the atmosphere nonthreatening and enable them to focus on a meaningful dialog with the instructors, teaching assistants, and the rest of the class.

The workshop schedule is designed so the instructors and teaching assistants can take some time to provide feedback to the project teams and so that students can ask questions. Presentations are limited to 10 minutes so there is time for feedback and questions during the class period. This limits the class size to about 36 presenters, assuming 90-minute lecture slots. All presentations are preloaded into a PC to control the schedule strictly and maximize feedback time. Since not all students present at the workshop, we have been able to accommodate up to 50 students in each class, with extra workshop time available as needed.

The instructors and teaching assistants must make a commitment to attend all the presentations and to give useful feedback. Students typically need the most help in answering the following questions: (1) Are the projects too big or too small for the amount of time available? (2) Is there existing work in the field that the project can benefit from? (3) Are the computations being targeted for parallel execution appropriate for the CUDA programming model?

Design Document

Once the students decide on a project and form a team, they are required to submit a design document for the project. This helps them 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, application-level objectives and potential impact, main features of the end application, an overview of their design, an implementation plan, their performance goals, a verification plan and acceptance test, and a project schedule.

The teaching assistants hold a project clinic for final project teams during the week before the class symposium. This clinic helps ensure that students are on track and that they have identified the potential roadblocks early in the process. Student teams are asked to come to the clinic with an initial draft of the following three versions of their application: (1) the best CPU sequential code in terms of performance, with SSE2 and other optimizations that establish a strong serial base of the code for their speedup comparisons and (2) the best CUDA parallel code in terms of performance—this version is the main output of the project. This version is used by the students to characterize the parallel algorithm overhead in terms of extra computations involved.

Student teams are asked to be prepared to discuss the key ideas used in each version of the code, any floating-point numerical issues, any comparison against previous results on the application, and the potential impact on the field if they achieve tremendous speedup. From our experience, the optimal schedule for the clinic is one week before the class symposium. An earlier time typically results in less mature projects and less meaningful sessions. A later time will not give students sufficient time to revise their projects according to the feedback.

Project Report

Students are required to submit a project report on their team's key findings. Six lecture slots are combined into 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 students' grades. Each student must answer questions directed to him or her as individuals, so that different grades can be assigned to individuals in the same team. We have recorded these presentations for viewing by future students at the ECE408 web site. The symposium is a major opportunity for students to learn to produce a concise presentation that motivates their peers to read a full paper. After their presentation, the students also submit a full report on their final project.

Online Supplements

The lab assignments, final project guidelines, and sample project specifications are available to instructors who use this book for their classes. While this book provides the intellectual contents for these classes, the additional material will be crucial in achieving the overall education goals. We would like to invite you to

take advantage of the online material that accompanies this book, which is available at

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.

Acknowledgements

To Caroline, Rose, and Leo
To Sabrina, Amanda, Bryan, and Carissa
for enduring our absence while working on the course and the book

This page intentionally left blank

Introduction

CHAPTER OUTLINE

Microprocessors based on a single central processing unit (CPU), such as those in the Intel Pentium family and the AMD Opteron family, drove rapid performance increases and cost reductions in computer applications for more than two decades. These microprocessors brought GFLOPS, or giga (10^{12}) floating-point operations per second, to the desktop and TFLOPS, or tera (10^{15}) floating-point operations per second, to cluster servers. 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.

This drive, however, has slowed since 2003 due to energy consumption and heat dissipation issues that limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Since then, virtually all microprocessor vendors have switched to models where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power. This switch has exerted a tremendous impact on the software developer community [Sutter2005].

Traditionally, the vast majority of software applications are written as sequential programs, as described by von Neumann in his seminal report in 1945 [vonNeumann1945]. The execution of these programs can be understood by a human sequentially stepping through the code. Historically, most software developers have relied on the advances in hardware to increase the speed of their sequential applications under the hood; the same software simply runs faster as each new generation of processors is introduced. Computer users have also become accustomed to the expectation that these programs run faster with each new generation of microprocessors. Such expectation is no longer valid from this day onward. A sequential program will only run on one of the processor cores, which will not become significantly faster than those in use today. Without performance improvement, application developers will no longer be able to introduce new features and capabilities into their software as new microprocessors are introduced, reducing the growth opportunities of the entire computer industry.

Rather, the applications software that will continue to enjoy 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 incentive for parallel program development has been referred to as the concurrency revolution [Sutter2005]. The practice of parallel programming is by no means new. The high-performance computing community has been developing parallel programs for decades. These programs run on large-scale, expensive computers. Only a few elite applications can justify the use of these expensive 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 [Hwu2008]. The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began with two core processors with the number of cores increasing with each semiconductor process generation. A current exemplar is the recent Intel Core i7™ microprocessor with four processor cores, each of which is an out-of-order, multiple instruction issue

processor implementing the full X86 instruction set, supporting hyper-threading with two hardware threads, designed to maximize the execution speed of sequential programs. In contrast, the *many-thread* trajectory focuses more on the execution throughput of parallel applications. The many-threads began with a large number of threads, and once again, the number of threads increases with each generation. A current exemplar is the NVIDIA GTX680 graphics processing unit (GPU) with 16,384 threads, executing in a large number of simple, in-order pipelines.

Many-threads processors, especially the GPUs, have led the race of floating-point performance since 2003. As of 2012, the ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs is about 10. These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips: 1.5 teraflops versus 150 gigaflops double precision in 2012.

Such a large performance gap between parallel and sequential execution has amounted to a significant "electrical potential" build-up, and at some point, something will have to give. We have reached that point now. To date, this large performance gap has already motivated many application developers to move the computationally intensive parts of their software to GPUs for execution. 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.

One might ask why there is such a large peak-performance gap between many-threads GPUs and general-purpose multicore CPUs. The answer lies in the differences in the fundamental design philosophies between the two types of processors, as illustrated in Figure 1.1. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed. As of 2012, the high-end general-purpose multicore microprocessors typically have six to eight large processor cores and multiple megabytes of on-chip cache memories designed to deliver strong sequential code performance.

Memory bandwidth is another important issue. The speed of many applications is limited by the rate at which data can be delivered from the

FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies.

memory system into the processors. Graphics chips have been operating at approximately six times the memory bandwidth of contemporaneously available CPU chips. In late 2006, GeForce 8800 GTX, or simply G80, was capable of moving data at about 85 gigabytes per second (GB/s) in and out of its main dynamic random-access memory (DRAM) because of graphics frame buffer requirements and the relaxed memory model (the way various system software, applications, and input/output (I/O) devices expect how their memory accesses work). The more recent GTX680 chip supports about 200 GB/s. In contrast, general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. As a result, CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

The design philosophy of GPUs is shaped by the fast-growing video game industry that exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations 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. The prevailing solution is to optimize for the execution throughput of massive numbers of threads. The design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency. The reduced area and power of the memory access hardware and arithmetic units allows the designers to have more of them on a chip and thus increase the total execution throughput.

The application software 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 are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM. This design style is commonly referred to as throughput-oriented design since 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.

The CPUs, on the other hand, are designed to minimize the execution latency of a single thread. 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. The arithmetic units and operand data delivery logic are also designed to minimize the effective latency of operation at the cost of increased use of chip area and power. By reducing the latency of operations within the same thread, the CPU hardware reduces the execution latency of each individual thread. However, the large cache memory, low-latency arithmetic units, and sophisticated operand delivery logic consume chip area and power that could be otherwise used to provide more arithmetic execution units and memory access channels. This design style is commonly referred to as latency-oriented design.

It should be clear now 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 numerically intensive parts on the GPUs. This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPU−GPU execution of an application.[1] The demand for supporting joint CPU−GPU execution is further reflected in more recent programming models such as OpenCL (see Chapter 14), OpenACC (see Chapter 15), and C++AMP (see Chapter 18).

It is also important to note that performance is not the only decision factor when application developers choose the processors for running their

---

[1]See Chapter 2 for more background on the evolution of GPU computing and the creation of CUDA.

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 micro-processors. Only a few elite applications funded by government and large corporations have been successfully developed on these traditional parallel computing systems. This has changed with many-core GPUs. Due to their popularity in the PC market, GPUs have been sold by the hundreds of millions. Virtually all PCs have GPUs in them. There are more than 400 million CUDA-enabled GPUs in use to date. This is the first time that massively parallel computing is feasible with a mass-market product. Such a large market presence has made these GPUs economically attrac-tive targets for application developers.

Another important decision factor is practical form factors and easy accessibility. Until 2006, parallel software applications usually ran on data center servers or departmental clusters. But such execution environ-ments 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 mag-netic resonance imaging (MRI) machines have been based on some com-bination of a PC and special hardware accelerators. The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks of compute server boxes into clinical settings, while this is common in academic departmental settings. In fact, National Institutes of Health (NIH) refused to fund parallel programming projects for some time: they felt that the impact of parallel software would be limited because huge cluster-based machines would not work in the clinical setting. Today, GE ships MRI products with GPUs and NIH funds research using GPU computing.

Yet another important consideration in selecting a processor for execut-ing numeric computing applications is the level of support for the Institute of Electrical and Electronic Engineers' (IEEE) floating-point standard. The standard makes it possible to have predictable results across proces-sors from different vendors. While the support for the IEEE floating-point standard was not strong in early GPUs, this has also changed for new generations of GPUs since the introduction of the G80. As we will discuss

in Chapter 7, GPU support for the IEEE floating-point standard has become comparable with that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable result values as the CPUs. Up to 2009, a major remaining issue was that the GPUs' floating-point arithmetic units were primarily single precision. Applications that truly require double-precision floating-point arithmetic units were not suitable for GPU execution. However, this has changed with the recent GPUs of which the double-precision execution speed approaches about half of that of single precision, a level that high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications.

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 processor cores, 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 to execute on these early GPUs. This technique was called GPGPU (general-purpose programming using a graphics processing unit). Even with a higher-level programming environment, the underlying code still needs to fit into the APIs that are designed to paint pixels. These APIs limit the kinds of applications that one can actually write for early GPGPUs. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent research results.

But everything changed in 2007 with the release of CUDA [NVIDIA2007]. NVIDIA stared to devote silicon areas on their GPU chips to facilitate the ease of parallel programming.This did not represent software changes alone; additional hardware was added to the chips. In the G80 and its successor chips for parallel computing, CUDA programs no longer go through the graphics interface at all. Instead, a new general-purpose parallel programming interface on the silicon chip serves the requests of CUDA programs. The general-purpose programming interface greatly expands the types of applications that one can easily develop for GPUs. Moreover, all the other software layers were redone as well, so that the programmers can use the familiar C/C++ programming tools. Some of our students tried to do their lab assignments using the old OpenGL-based programming interface, and their experience helped them to greatly appreciate the improvements that eliminated the need for using the graphics APIs for computing applications.

1.2 ARCHITECTURE OF A MODERN GPU

Figure 1.2 shows the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Figure 1.3, two SMs form a building block. However, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Also, in Figure 1.3, each SM has a number of streaming processors (SPs) that share control logic and an instruction cache. Each GPU currently comes with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory in Figure 1.3. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images and texture information for 3D rendering. But for computing, they function as very high bandwidth off-chip memory, though with somewhat longer latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

The G80 introduced the CUDA architecture and had 86.4 GB/s of memory bandwidth, plus a communication link to the CPU core logic over a PCI-Express Generation 2 (Gen2) interface. Over PCI-E Gen2, a CUDA application can transfer data from the system memory to the global memory at 4 GB/s, and at the same time upload data back to the system memory at 4 GB/s. Altogether, there is a combined total of 8 GB/s. More recent GPUs use PCI-E Gen3, which supports 8 GB/s in each direction. As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E to communicate with the CPU system memory if there is need for using a library that is only available on the CPUs. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

With 16,384 threads, the GTX680 exceeds 1.5 teraflops in double precision. A good application typically runs 5,000−12,000 threads simultaneously on this chip. For those who are used to multithreading in CPUs, note that Intel CPUs support two or four threads, depending on the machine model, per core. CPUs, however, are increasingly used with SIMD (single instruction, multiple data) instructions for high numerical performance. The level of parallelism supported by both GPU hardware and CPU hardware is increasing quickly. It is therefore very important to strive for high levels of parallelism when developing computing applications.
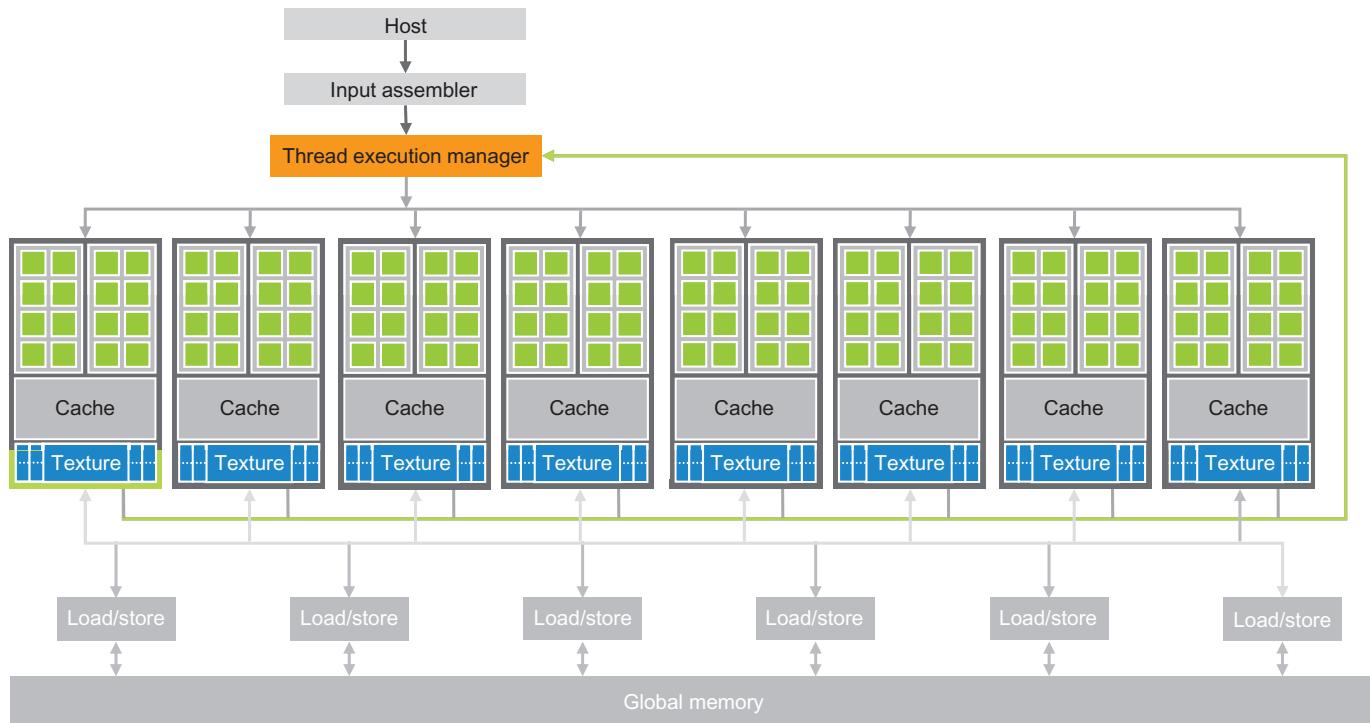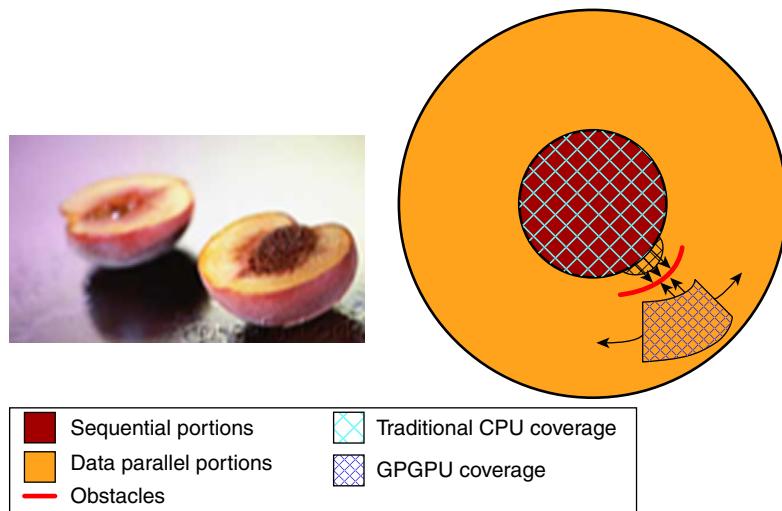
FIGURE 1.2

Architecture of a CUDA-capable GPU.

FIGURE 1.3

Coverage of sequential and parallel application portions.

1.3 WHY MORE SPEED OR PARALLELISM?

As we stated in Section 1.1, the main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations. One might ask why applications will continue to demand increased speed. Many applications that we have today seem to be running quite fast enough. As we will discuss in the case study chapters, when an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100 times ($100\times$) speedup over sequential execution on a single CPU core. If the application includes what we call *data parallelism*, it's often a simple task to achieve a $10\times$ speedup with just a few hours of work. For anything beyond that, we invite you to keep reading!

Despite the myriad of computing applications in today's world, many exciting mass-market applications of the future are what we currently consider "supercomputing applications," or super-applications. For example, the biology research community is moving more and more into the molecular level. Microscopes, arguably the most important instrument in molecular biology, used to rely on optics or electronic instrumentation. But there are limitations to the molecular-level observations that we can make with these instruments. These limitations can be effectively

addressed by incorporating a computational model to simulate the underlying molecular activities with boundary conditions set by traditional instrumentation. With simulation we can measure even more details and test more hypotheses than can ever be imagined with traditional instrumentation alone. These simulations will continue to benefit from the increasing computing speed in the foreseeable future in terms of the size of the biological system that can be modeled and the length of reaction time that can be simulated within a tolerable response time. These enhancements will have tremendous implications to science and medicine.

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

Among the benefits offered by more computing speed are much better user interfaces. Consider Apple's iPhone™ interfaces: the user enjoys a much more natural interface with the touchscreen than other cell phone devices even though the iPhone still has a limited-size window. Undoubtedly, future versions of these devices will incorporate higher-definition, 3D perspectives, applications that combine virtual and physical space information for enhanced usability, and voice-based and computer vision−based interfaces, requiring even more computing speed.

Similar developments are underway in consumer electronic gaming. In the past, driving a car in a game was in fact simply a prearranged set of scenes. If your car bumped into an obstacle, the course of your vehicle did not change, only the game score changed. Your wheels were not bent or damaged, and it was no more difficult to drive, regardless of whether you bumped your wheels or even lost a wheel. With increased computing speed, the games can be based on dynamic simulation rather than prearranged scenes. We can expect to see more of these realistic effects in the future: accidents will damage your wheels and your online driving experience will be much more realistic. Realistic modeling and simulation of physics effects are known to demand very large amounts of computing power.

All the new applications that we mention here involve simulating a physical, concurrent world in different ways and at different levels, with tremendous amounts of data being processed. In fact, the problem of handling massive amounts of data is so prevalent that the term *big data* has become a household word. And with this huge quantity of data, much of the computation can be done on different parts of the data in parallel, although they will have to be reconciled at some point. In most cases, effective management of data delivery can have a major impact on the achievable speed of a parallel application. While techniques for doing so are often well known to a few experts who work with such applications on a daily basis, the vast majority of application developers can benefit from more intuitive understanding and practical working knowledge of these techniques.

We aim to present the data management techniques in an intuitive way to application developers whose formal education may not be in computer science or computer engineering. We also aim to provide many practical code examples and hands-on exercises that help readers 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.

1.4 SPEEDING UP REAL APPLICATIONS

How many times speedup can be expected from parallelizing an application? It depends on the portion of the application that can be parallelized. If the percentage of time spent in the part that can be parallelized is 30%, a $100\times$ speedup of the parallel portion will reduce the execution time by no more than 29.7%. The speedup for the entire application will be only about $1.4\times$. In fact, even an infinite amount of speedup in the parallel portion can only slash 30% off execution time, achieving no more than $1.43\times$ speedup. On the other hand, if 99% of the execution time is in the parallel portion, a $100\times$ speedup will reduce the application execution to 1.99% of the original time. This gives the entire application a $50\times$ speedup. Therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speed up its execution.

Researchers have achieved speedups of more than $100\times$ 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 execution time is in parallel execution. In practice, straightforward parallelization of applications often saturates the memory (DRAM) bandwidth, resulting in only about a $10\times$ speedup. The trick is to figure out how to get around memory bandwidth limitations, which involves doing one of many transformations to utilize specialized GPU on-chip memories to drastically reduce the number of accesses to the DRAM. One must, however, further optimize the code to get around limitations such as limited on-chip memory capacity. An important goal of this book is to help readers fully understand these optimizations and become skilled in them.

Keep in mind that the level of speedup achieved over single-core CPU execution can also reflect the suitability of the CPU to the application: in some applications, CPUs perform very well, making it harder to speed up performance using a GPU. Most applications have portions that can be much better executed by the CPU. Thus, one must give the CPU a fair chance to perform and make sure that code is written so that GPUs *complement* CPU execution, thus properly exploiting the heterogeneous parallel computing capabilities of the combined CPU−GPU system. This is precisely what the CUDA programming model promotes, as we will further explain in the book.

Figure 1.3 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 these portions, although they can take up a large portion of the code, tend to account for only a small portion of the execution time of super-applications.

Then come what we call the "peach meat" portions. These portions are easy to parallelize, as are some early graphics applications. Parallel programming in heterogeneous computing systems can drastically improve the quality of these applications. As illustrated in Figure 1.3, early GPGPUs cover only a small portion of the meat section, which is analogous to a small portion of the most exciting applications. As we will see, the CUDA programming models are designed to cover a much larger section of the peach meat portions of exciting applications. In fact, as we will discuss in Chapter 20, these programming models and their underlying hardware are still evolving at a fast pace to enable efficient parallelization of even larger sections of applications.

1.5 **PARALLEL PROGRAMMING LANGUAGES AND MODELS**

Many parallel programming languages and models have been proposed in the past several decades Mattson2004]. The ones that are the most widely used are Message Passing Interface (MPI) [MPI2009] for scalable cluster computing, and OpenMP [Open2005] for shared-memory multiprocessor systems. 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. More recently, a variation called OpenACC (see Chapter 15) has been proposed and supported by multiple computer vendors for programming heterogeneous computing systems.

The major advantage of OpenACC is that it provides compiler automation and runtime support for abstracting away many parallel programming details from programmers. Such automation and abstraction can help make the application code more portable across systems produced by different vendors, as well as different generations of systems from the same vendor. This is why we teach OpenACC programming in Chapter 15. However, effective programming in OpenACC still requires the programmers to understand all the detailed parallel programming concepts involved. Because CUDA gives programmers explicit control of these parallel programming details, it is an excellent learning vehicle even for someone who would like to use OpenMP and OpenACC as their primary programming interface. Furthermore, from our experience, OpenACC compilers are still evolving and improving. Many programmers will likely need to use CDUA-style interfaces for parts where OpenACC compilers fall short.

MPI is a model where computing nodes in a cluster do not share memory [MPI2009]. All data sharing and interaction must be done through explicit message passing. MPI has been successful in high-performance computing (HPC). Applications written in MPI have run successfully on cluster computing systems with more than 100,000 nodes. Today, many HPC clusters employ heterogeneous CPU−GPU nodes. While CUDA is an effective interface with each node, most application developers need to use MPI to program at the cluster level. Therefore, it is important that a parallel

programmer in HPC understands how to do joint MPI/CUDA programming, which is presented in Chapter 19.

The amount of effort needed to port an application into MPI, however, can be quite high due to the lack of shared memory across computing nodes. The programmer needs to perform domain decomposition to partition the input and output data into cluster nodes. Based on the domain decomposition, the programmer also needs to call message sending and receiving functions to manage the data exchange between nodes. CUDA, on the other hand, provides shared memory for parallel execution in the GPU to address this difficulty. As for CPU and GPU communication, CUDA previously provided very limited shared memory capability between the CPU and the GPU. The programmers needed to manage the data transfer between the CPU and the GPU in a manner similar to the "one-sided" message passing. New runtime support for global address space and automated data transfer in heterogeneous computing systems, such as GMAC [GCN2010] and CUDA 4.0, are now available. With GMAC, a CUDA or OpenCL programmer can declare C variables and data structures as shared between CPU and GPU. The GMAC runtime maintains coherence and automatically performs optimized data transfer operations on behalf of the programmer on an as-needed basis. Such support significantly reduces the CUDA and OpenCL programming complexity involved in overlapping data transfer with computation and I/O activities.

In 2009, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, jointly developed a standardized programming model called Open Compute Language (OpenCL) [Khronos2009]. 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 than CUDA. This allows vendors to quickly adapt their existing compilers and tools to handle OpenCL programs. OpenCL is a standardized programming model in that applications developed in OpenCL can run correctly without modification on all processors that support the OpenCL language extensions and API. However, one will likely need to modify the applications to achieve high performance for a new processor.

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

learned using CUDA can be easily applied to OpenCL programming. Therefore, we introduce OpenCL in Chapter 14 and explain how one can apply the key concepts in this book to OpenCL programming.

1.6 OVERARCHING GOALS

Our primary goal is to teach the readers how to program massively parallel processors to achieve high performance, and our approach will not require a great deal of hardware expertise. Someone once said that if you don't care about performance, parallel programming is very easy. You can literally write a parallel program in an hour. But we're going to dedicate many pages to techniques for developing *high-performance* parallel programs. And, we believe that it will become easy once you develop the right insight and go about it the right way. In particular, we will focus on *computational thinking* techniques that will enable you to think about problems in ways that are amenable to high-performance parallel computing.

Note that hardware architecture features have constraints. High-performance parallel programming on most processors will require some knowledge of how the hardware works. It will probably take 10 or more years before we can build tools and machines so that most programmers can work without this knowledge. Even if we have such tools, we suspect that programmers with more knowledge of the hardware will be able to use the tools in a much more effective way than those who do not. However, we will not be teaching computer architecture as a separate topic. Instead, we will teach the essential computer architecture knowledge as part our discussions on high-performance parallel programming techniques.

Our second goal is to teach parallel programming for correct functionality and reliability, which constitute a subtle issue in parallel computing. Those who have worked on parallel systems in the past know that achieving initial performance is not enough. The challenge is to achieve it in such a way that you can debug the code and support users. The CUDA programming model encourages the use of a simple form of barrier synchronization and memory consistency for managing parallelism. We will show that by focusing on data parallelism, one can achieve both high performance and high reliability in their 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 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 accessing and updating data structures.

Much technical knowledge will be required to achieve these goals, so we will cover quite a few principles and patterns of parallel programming in this book. We cannot guarantee that we will cover all of them, however, so we have selected the most useful and well-proven techniques to cover in detail. To complement your knowledge and expertise, we include a list of recommended literature. We are now ready to give you a quick overview of the rest of the book.

1.7 ORGANIZATION OF THE BOOK

Chapter 2 reviews the history of GPU computing. It starts with a brief summary of the evolution of graphics hardware toward more programmability and then discusses the historical GPGPU movement. Many of the current features and limitations of the CUDA programming model find their root in these historic developments. A good understanding of these historic developments will help readers better understand the current state and the future trends of hardware evolution that will continue to impact the types of applications that will benefit from CUDA.

Chapter 3 introduces data parallelism and the CUDA C programming. This chapter relies on the fact that students have had previous experience with C programming. It first introduces CUDA C as a simple, small extension to C that supports heterogeneous CPU−GPU joint computing and the widely used SPMD (single program, multiple data) parallel programming model. It then covers the thought process involved in (1) identifying the part of application programs to be parallelized; (2) isolating the data to be used by the parallelized code, using an API (Application Programming Interface) function to allocate memory on the parallel computing device; (3) using an API function to transfer data to the parallel computing device; (4) developing a kernel function that will be executed by threads in the parallelized part; (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.

While the objective of Chapter 3 is to teach enough concepts of the CUDA C programming model so that the students can write a simple parallel CUDA C program, it actually covers several basic skills needed to

develop a parallel application based on any parallel programming model. We use a running example of vector addition to make this chapter concrete. We also compare CUDA with other parallel programming models including OpenMP and OpenCL.

Chapter 4 presents more details of the parallel execution model of CUDA. It gives enough insight into the creation, organization, resource binding, data binding, and scheduling of threads to enable readers to implement sophisticated computation using CUDA C and reason about the performance behavior of their CUDA code. Chapter 5 is dedicated to the special memories that can be used to hold CUDA variables for managing data delivery and improving program execution speed.

Chapter 6 presents several important performance considerations in current CUDA hardware. In particular, it gives more details in thread execution, memory data accesses, and resource allocation. These details form the conceptual basis for programmers to reason about the consequence of their decisions on organizing their computation and data.

Chapter 7 introduces the concepts of floating-point number format, precision, and accuracy. It shows why different parallel execution arrangements can result in different output values. It also teaches the concept of numerical stability and practical techniques for maintaining numerical stability in parallel algorithms.

Chapters 8-10 present three important parallel computation patterns that give readers more insight into parallel programming techniques and parallel execution mechanisms. Chapter 8 presents convolution, a frequently used parallel computing pattern that requires careful management of data access locality. We also use this pattern to introduce constant memory and caching in modern GPUs. Chapter 9 presents prefix sum, or scan, an important parallel computing pattern that coverts sequential computation into parallel computation. We also use this pattern to introduce the concept of work efficiency in parallel algorithms. Chapter 10 presents sparse matrix computation, a pattern used for processing very large data sets. This chapter introduces readers to the concepts of rearranging data for more efficient parallel access: padding, sorting, transposition, and regularization.

While these chapters are based on CUDA, they help readers build up the foundation for parallel programming in general. We believe that humans understand best when we learn from the bottom up. 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 model. An in-depth experience

with the CUDA model also enables us to gain maturity, which will help us learn concepts that may not even be pertinent to the CUDA model.

Chapters 11 and 12 are case studies of two real applications, which take readers through the thought process of parallelizing and optimizing their applications for significant speedups. For each application, we start by identifying alternative ways of formulating the basic structure of the parallel execution and follow up with reasoning about the advantages and disadvantages of each alternative. We then go through the steps of code transformation needed to achieve high performance. These two chapters help readers put all the materials from the previous chapters together and prepare for their own application development projects.

Chapter 13 generalizes the parallel programming techniques into problem decomposition principles, algorithm strategies, and computational thinking. 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 concepts into computational tasks, which is an important first step in producing quality application software, serial or parallel. It then discusses parallel algorithm structures and their effects on application performance, which is grounded in the performance tuning experience with CUDA. The chapter concludes with a treatment of parallel programming styles and models, enabling readers to place their knowledge in a wider context. With this chapter, readers can begin to generalize from the SPMD programming style to other styles of parallel programming, such as loop parallelism in OpenMP and fork-join in p-thread programming. Although we do not go into these alternative parallel programming styles, we expect that readers will be able to learn to program in any of them with the foundation they gain in this book.

Chapter 14 introduces the OpenCL programming model from a CUDA programmer's perspective. Readers will find OpenCL to be extremely similar to CUDA. The most important difference arises from OpenCL's use of API functions to implement functionalities such as kernel launching and thread identification. The use of API functions makes OpenCL more tedious to use. Nevertheless, a CUDA programmer has all the knowledge and skills needed to understand and write OpenCL programs. In fact, we believe that the best way to teach OpenCL programming is to teach CUDA first. We demonstrate this with a chapter that relates all major OpenCL features to their corresponding CUDA features. We also illustrate the use of these features by adapting our simple CUDA examples into OpenCL.

Chapter 15 presents the OpenACC programming interface. It shows how to use directives and pragmas to tell the compiler that a loop can be parallelized, and if desirable, instruct the compiler how to parallelize the loop. It also uses concrete examples to illustrate how one can take advantage of the interface and make their code more portable across vendor systems. With the foundational concepts in this book, readers will find the OpenACC programming directives and pragmas easy to learn and master.

Chapter 16 covers Thrust, a productivity-oriented C++ library for building CUDA applications. This is a chapter that shows how modern object-oriented programming interfaces and techniques can be used to increase productivity in a parallel programming environment. In particular, it shows how generic programming and abstractions can significantly reduce the efforts and code complexity of applications.

Chapter 17 presents CUDA FORTRAN, an interface that supports FORTRAN-style programming based on the CUDA model. All concepts and techniques learned using CUDA C can be applied when programming in CUDA. In addition, the CUDA FORTRAN interface has strong support for multidimensional arrays that make programming of 3D models much more readable. It also assumes the FORTRAN array data layout convention and works better with an existing application written in FORTRAN.

Chapter 18 is an overview of the C++AMP programming interface from Microsoft. This programming interface uses a combination language extension and API support to support data-parallel computation patterns. It allows programmers to use C++ features to increase their productivity. Like OpenACC, C++AMP abstracts away some of the parallel programming details that are specific to the hardware so the code is potentially more portable across vendor systems.

Chapter 19 presents an introduction to joint MPI/CUDA programming. We cover the key MPI concepts that a programmer needs to understand to scale their heterogeneous applications to multiple nodes in a cluster environment. In particular, we will focus on domain partitioning, point-to-point communication, collective communication in the context of scaling a CUDA kernel into multiple nodes.

Chapter 20 introduces the dynamic parallelism capability available in the Kepler GPUs and their successors. Dynamic parallelism can potentially help the implementations of sophisticated algorithms to reduce CPU-GPU interaction overhead, free up CPU for other tasks, and improve the utilization of GPU execution resources. We describe the basic concepts of dynamic parallelism and why some algorithms can benefit from dynamic parallelism. We then illustrate the usage of dynamic parallelism with a

small contrived code example as well as a more complex realistic code example.

Chapter 21 offers some 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 present a brief survey of the major trends in the architecture of massively parallel processors and how these trends will likely impact parallel programming in the future. We 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

Gelado, I., Cabezas, J., Navarro, N., Stone, J. E., Patel, S. J., & Hwu, W. W. An Asynchronous Distributed Shared Memory Model for Heterogeneous Parallel Systems, International Conference on Architectural Support for Programming Languages and Operating Systems, March 2010. Technical Report, IMPACT Group, University of Illinois, Urbana-Champaign.

Hwu, W. W., Keutzer, K., & Mattson, T. (2008). The Concurrency Challenge. *IEEE Design and Test of Computers,* July/August 312−320.

The Khronos Group, The OpenCL Specification Version 1.0, Available at: (*http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf*).

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). Patterns of Parallel Programmin*g* Boston: Addison-Wesley Professional.

Message Passing Interface Forum, "MPI—A Message Passing Interface Standard Version 2.2," Available at: <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>., Sept. 4, 2009.

NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.0, June 2007, Available at: <http://www.cs.berkeley.edu/∼yelick/cs194f07/handouts/NVIDIA_CUDA_Programming_Guide.pdf>

OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1." July 2011, Available at: <http://www.openmp.org/mp-documents/OpenMP3.1.pdf>

Sutter, H., & Larus,, J. (2005). Software and the Concurrency Revolution. *ACM Queue*, *3*(7), 54−62.

von Neumann, J. (1945). First Draft of a Report on the EDVAC. In H. H. Goldstine (Ed.), *The Computer: From Pascal to von Neumann*. Princeton, NJ: Princeton University Press.

Wing, J. (2006). Computational Thinking. *Communications of the ACM*, *49*(3), 33−35.

This page intentionally left blank

History of GPU Computing

2

CHAPTER OUTLINE

To CUDA C and OpenCL programmers, GPUs are massively parallel numeric computing processors programmed in C with extensions. One does not need to understand graphics algorithms or terminology to be able to program these processors. However, understanding the graphics heritage of these processors illuminates the strengths and weaknesses of them with respect to major computational patterns. In particular, the history helps to clarify the rationale behind major architectural design decisions of modern programmable GPUs: massive multithreading, relatively small cache memories compared to CPUs, and bandwidth-centric memory interface design. Insights into the historical developments will also likely give readers the context needed to project the future evolution of GPUs as computing devices.

2.1 EVOLUTION OF GRAPHICS PIPELINES

Three-dimensional (3D) graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid- to late 1990s. During this period, the performance-leading graphics subsystems declined in price from $50,000 to $500. During the same period, the performance increased from 50 million pixels per second to 1 billion pixels per second, and from 100,000 vertices per second to 10 million vertices per second. While these

advancements have much to do with the relentlessly shrinking feature sizes of semiconductor devices, they also come from the new innovations of graphics algorithms and hardware design innovations. These innovations have shaped the native hardware capabilities of modern GPUs.

The remarkable advancement of graphics hardware performance has been driven by the market demand for high-quality real-time graphics in computer applications. For example, in an electronic gaming application, one needs to render evermore complex scenes at ever-increasing resolution at a rate of 60 frames per second. The net result is that over the last 30 years, graphics architecture has evolved from a simple pipeline for drawing wireframe diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering complex interactive imagery of 3D scenes. Concurrently, many of the hardware functionalities involved became far more sophisticated and user programmable.

The Era of Fixed-Function Graphics Pipelines

From the early 1980s to the late 1990s, the leading performance graphics hardware was fixed-function pipelines that were configurable, but not programmable. In that same era, major graphics Application Programming Interface (API) libraries became popular. An API is a standardized layer of software, that is, a collection of library functions that allows applications (e.g., games) to use software or hardware services and functionality. For example, an API can allow a game to send commands to a graphics processing unit to draw objects on a display. One such API is DirectX, Microsoft's proprietary API for media functionality. The Direct3D component of DirectX provides interface functions to graphics processors. The other major API is OpenGL, an open-standard API supported by multiple vendors and popular in professional workstation applications. This era of fixed-function graphics pipeline roughly corresponds to the first seven generations of DirectX.

DIRECT MEMORY ACCESS

Modern computer systems use a specialized hardware mechanism called direct memory access (DMA) to transfer data between an I/O device and the system DRAM. When a program requests an I/O operation, say reading from a disk drive, the operating system makes an arrangement by setting a DMA operation defined by the starting address of the data in the I/O device buffer memory, the starting address of the DRAM memory, the number of bytes to be copied, and the direction of the copy.

Using a specialized hardware mechanism to copy data between I/O devices and system DRAM has two major advantages. First, the CPU is not burdened with the chore of copying

data. So, while the DMA hardware is copying data, the CPU can execute programs that do not depend on the I/O data.

The second advantage of using a specialized hardware mechanism to copy data is that the hardware is designed to perform copy. The hardware is very simple and efficient. There is no overhead of fetching and decoding instructions while performing the copy. As a result, the copy can be done at a higher speed than most processors can.

As we will learn later, DMA is used in data copy operations between a CPU and a GPU. It requires pinned memory in DRAM and has subtle implications on how applications should allocate memory.

Figure 2.1 shows an example fixed-function graphics pipeline in early NVIDIA GeForce GPUs. The host interface receives graphics commands and data from the CPU. The commands are typically given by application programs by calling an API function. The host interface typically contains a specialized DMA hardware to efficiently transfer bulk data to and from the host system memory to the graphics pipeline. The host interface also communicates back the status and result data of executing the commands.

Before we describe the other stages of the pipeline, we should clarify that the term *vertex* usually means the "corners" of a polygon. The
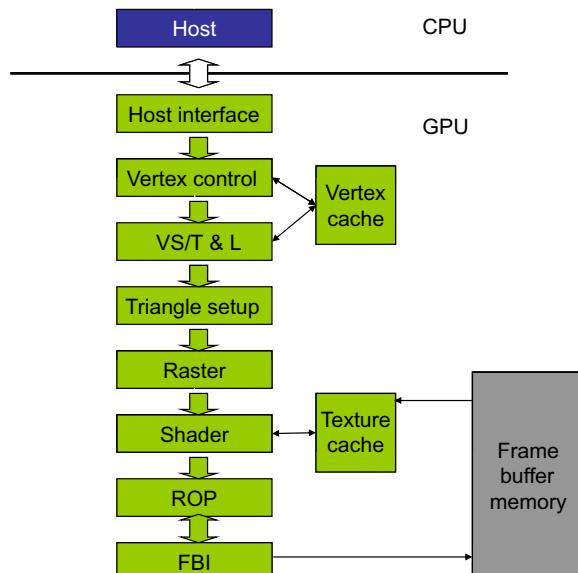
FIGURE 2.1

A fixed-function NVIDIA GeForce graphics pipeline.
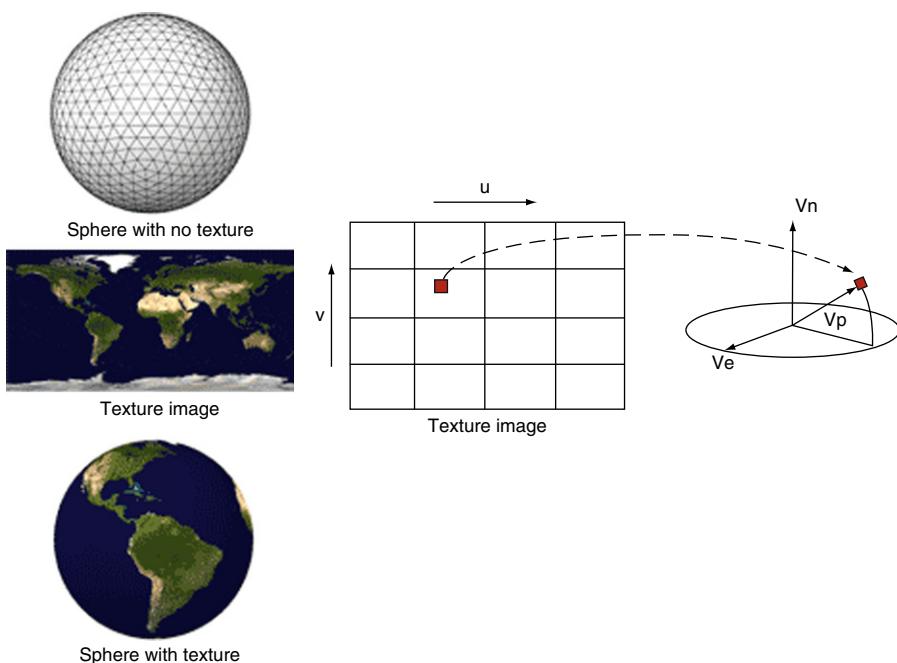
GeForce graphics pipeline is designed to render triangles, so vertex is typically used to refer to the corners of a triangle. The surface of an object is drawn as a collection of triangles. The finer the sizes of the triangles are, the better the quality of the picture typically becomes. The vertex control stage in Figure 2.1 receives parameterized triangle data from the CPU. The vertex control stage converts the triangle data into a form that the hardware understands and places the prepared data into the vertex cache.

The vertex shading, transform, and lighting (VS/T&L) stage in Figure 2.1 transforms vertices and assigns per-vertex values (colors, normals, texture coordinates, tangents, etc.). The shading is done by the pixel shader hardware. The vertex shader can assign a color to each vertex but it is not applied to triangle pixels until later. The triangle setup stage further creates edge equations that are used to interpolate colors and other per-vertex data (e.g., texture coordinates) across the pixels touched by the triangle. The raster stage determines which pixels are contained in each triangle. For each of these pixels, the raster stage interpolates per-vertex values necessary for shading the pixel, which includes color, position, and texture position that will be shaded (painted) on the pixel.
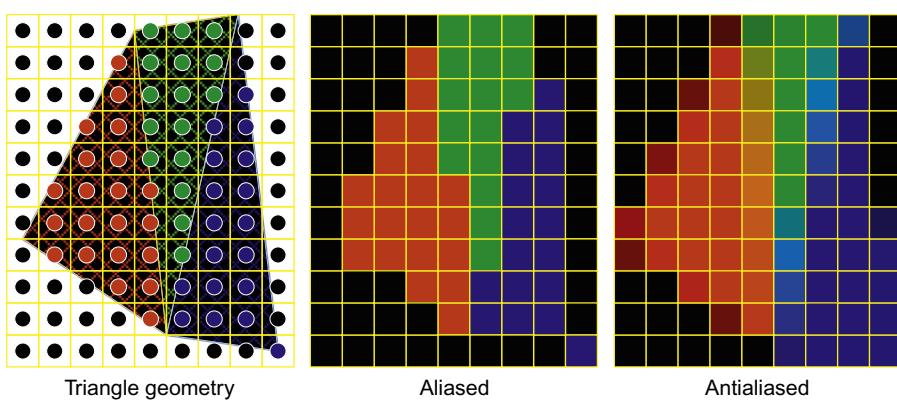
The shader stage in Figure 2.1 determines the final color of each pixel. This can be generated as a combined effect of many techniques: interpolation of vertex colors, texture mapping, per-pixel lighting mathematics, reflections, and more. Many effects that make the rendered images more realistic are incorporated in the shader stage. Figure 2.2 illustrates texture mapping, one of the shader stage functionalities. It shows an example in which a world map texture is mapped onto a sphere object. Note that the sphere object is described as a large collection of triangles. Although the shader stage needs to perform only a small number of coordinate transform calculations to identify the exact coordinates of the texture point that will be painted on a point in one of the triangles that describes the sphere object, the sheer number of pixels covered by the image requires the shader stage to perform a very large number of coordinate transforms for each frame.

The ROP (raster operation) stage in Figure 2.2 performs the final raster operations on the pixels. It performs color raster operations that blend the color of overlapping/adjacent objects for transparency and anti-aliasing effects. It also determines the visible objects for a given viewpoint and discards the occluded pixels. A pixel becomes occluded when it is blocked by pixels from other objects according to the given viewpoint.

Figure 2.3 illustrates anti-aliasing, one of the ROP stage operations. There are three adjacent triangles with a black background. In the aliased

FIGURE 2.2

Texture mapping example: painting a world map texture image.

FIGURE 2.3

Examples of anti-aliasing operations: (a) triangle geometry, (b) aliased, and (c) anti-aliased.

output, each pixel assumes the color of one of the objects or the background. The limited resolution makes the edges look crooked and the shapes of the objects distorted. The problem is that many pixels are partly in one object and partly in another object or the background. Forcing these pixels to assume the color of one of the objects introduces distortion into the edges of the objects. The anti-aliasing operation gives each pixel a color that is blended, or linearly combined, from the colors of all the objects and background that partly overlap the pixel. The contribution of each object to the color of the pixel is to the amount of the pixel that the object overlaps.

Finally, the frame buffer interface (FBI) stage in Figure 2.1 manages memory reads from and writes to the display frame buffer memory. For high-resolution displays, there is a very high bandwidth requirement in accessing the frame buffer. Such bandwidth is achieved by two strategies. One is that graphics pipelines typically use special memory designs that provide higher bandwidth than the system memories. Second, the FBI simultaneously manages multiple memory channels that connect to multiple memory banks. The combined bandwidth improvement of multiple channels and special memory structures gives the frame buffers much higher bandwidth than their contemporaneous system memories. Such high memory bandwidth has continued to this day and has become a distinguishing feature of modern GPU design.

During the past two decades, each generation of hardware and its corresponding generation of API brought incremental improvements to the various stages of the graphics pipeline. Each generation introduced hardware resources and configurability to the pipeline stages. However, developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The obvious next step was to make some of these graphics pipeline stages into programmable processors.

Evolution of Programmable Real-Time Graphics

In 2001, the NVIDIA GeForce 3 took the first step toward true general shader programmability. It exposed the application developer to what had been the private internal instruction set of the floating-point vertex engine (VS/T&L stage). This coincided with the release of Microsoft DirectX 8 and OpenGL vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating-point capability to the pixel shader stage, and made texture accessible from the vertex shader

stage. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel shader processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. These programmable pixel shader processors were part of a general trend toward unifying the functionality of the different stages as seen by the application programmer. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs dedicated to vertex and pixel processing. The XBox 360 introduced an early unified-processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

In graphics pipelines, certain stages do a great deal of floating-point arithmetic on completely independent data, such as transforming the positions of triangle vertices or generating pixel colors. This *data independence* as the dominating application characteristic is a key difference between the design assumptions for GPUs and CPUs. A single frame, rendered in 1/60 of a second, might have 1 million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

The specific functions executed at a few graphics pipeline stages vary with rendering algorithms. Such variation has motivated the hardware designers to make those pipeline stages programmable. Two particular programmable stages stand out: the vertex shader and the pixel shader. Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation. Typically a vertex shader thread reads a floating-point (x, y, z, w) vertex position and computes a floating-point (x, y, z) screen position. Geometry shader programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Vertex shader programs and geometry shader programs execute on the VS/T&L stage of the graphics pipeline.

Pixel shader programs each "shade" one pixel, computing a floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample (x, y) image position. These programs execute on the shader stage of the graphics pipeline. For all three types of graphics shader programs, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects. This property has motivated the design of the programmable pipeline stages into massively parallel processors.

Figure 2.4 shows an example of a programmable pipeline that employs a vertex processor and a fragment (pixel) processor. The programmable vertex processor executes the programs designated to the VS/T&L stage

3D application
or game

3D API
Commands

3D API:
OpenGL or
Direct3D

CPU

CPU – GPU boundary

GPU
command and
data stream

GPU

GPU
front
end

Vertex index
stream

Assembled
polygons,
lines, and
points

Pixel
location
stream

Pixel
updates

Primitive
assembly

Rasterization and
interpolation

Raster
operations

Framebuffer

Pretransformed
Vertices

Transformed
vertices

Rasterized
pretransformed
fragments

Transformed
fragments

Programmable
vertex
processor

Programmable
fragment
processor

FIGURE 2.4

An example of a separate vertex processor and fragment processor in a programmable graphics pipeline.

and the programmable fragment processor executes the programs designated to the (pixel) shader stage. Between these programmable graphics pipeline stages are dozens of fixed-function stages that perform well-defined tasks far more efficiently than a programmable processor could, and that would benefit far less from programmability. For example, between the geometry processing stage and the pixel processing stage is a "rasterizer," a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries. Together, the mix of programmable and fixed-function stages is engineered to balance extreme performance with user control over the rendering algorithms.

Common rendering algorithms perform a single pass over input primitives and access other memory resources in a highly coherent manner. That is, these algorithms tend to simultaneously access contiguous memory locations, such as all triangles or all pixels in a neighborhood. As a result, these algorithms exhibit excellent efficiency in memory bandwidth utilization and are largely insensitive to memory latency. Combined with a pixel shader workload that is usually compute-limited, these characteristics have guided GPUs along a different evolutionary path than CPUs. In particular, whereas the CPU die area is dominated by cache memories, GPUs are dominated by floating-point data path and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (since latency can be readily hidden by massively parallel execution); indeed, bandwidth is typically many times higher than a CPU, exceeding 190 GB/s in more recent designs.

Unified Graphics and Computing Processors

Introduced in 2006, NVIDIA's GeForce 8800 GPU mapped the separate programmable graphics stages to an array of unified processors; the logical graphics pipeline is physically a recirculating path that visits these processors three times, with much fixed-function graphics logic between visits. This is illustrated in Figure 2.5. The unified processor array allows dynamic partitioning of the array to vertex shading, geometry processing, and pixel processing. Since different rendering algorithms present wildly different loads among the three programmable stages, this unification allows the same pool of execution resources to be dynamically allocated to different pipeline stages and achieve better load balance.

The GeForce 8800 hardware corresponds to the DirectX 10 API generation. By the DirectX 10 generation, the functionality of vertex and pixel
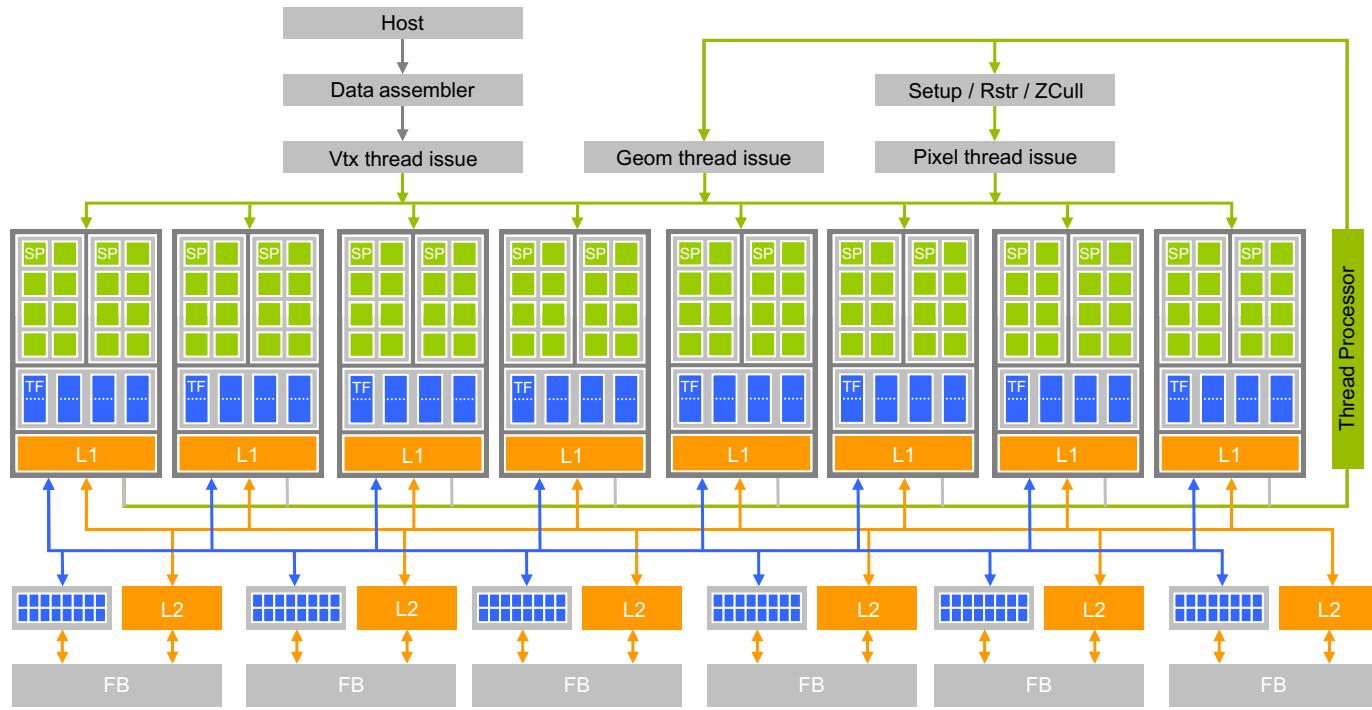
FIGURE 2.5

Unified programmable processor array of the GeForce 8800 GT graphics pipeline.

shaders was to be made identical to the programmer, and a new logical stage was introduced, the geometry shader, to process all the vertices of a primitive rather than vertices in isolation. The GeForce 8800 was designed with DirectX 10 in mind. Developers were coming up with more sophisticated shading algorithms and this motivated a sharp increase in the available shader operation rate, particularly floating-point operations. NVIDIA pursued a processor design with higher operating clock frequency than what was allowed by standard-cell methodologies to deliver the desired operation throughput as area-efficiently as possible. High−clock speed design requires substantially more engineering effort, and this favored designing one processor array, rather than two (or three, given the new geometry stage). It became worthwhile to take on the engineering challenges of a unified processor (load balancing and recirculation of a logical pipeline onto threads of the processor array) while seeking the benefits of one processor design. Such design paved the way for using the programmable GPU processor array for general numeric computing.

2.2 GPGPU: AN INTERMEDIATE STEP

While the GPU hardware design evolved toward more unified processors, it increasingly resembled high-performance parallel computers. As DirectX 9−capable GPUs became available, some researchers took notice of the raw performance growth path of GPUs and they started to explore the use of GPUs to solve compute-intensive science and engineering problems. However, DirectX 9 GPUs had been designed only to match the features required by the graphics APIs. To access the computational resources, a programmer had to cast his or her problem into graphics operations so that the computation could be launched through OpenGL or DirectX API calls. For example, to run many simultaneous instances of a compute function, it had to be written as a pixel shader. The collection of input data had to be stored in texture images and issued to the GPU by submitting triangles (with clipping to a rectangle shape if that's what was desired). The output had to be cast as a set of pixels generated from the raster operations.

The fact that the GPU processor array and frame buffer memory interface were designed to process graphics data proved too restrictive for general numeric applications. In particular, the output data of the shader programs are single pixels of which the memory location has been predetermined. Thus, the graphics processor array is designed with very restricted

memory reading and writing capability. Figure 2.6 illustrates the limited memory access capability of early programmable shader processor arrays; shader programmers needed to use texture to access arbitrary memory locations for their input data. More importantly, shaders did not have the means to perform writes with calculated memory addresses, referred to as *scatter operations*, to memory. The only way to write a result to memory was to emit it as a pixel color value, and configure the frame buffer operation stage to write (or blend, if desired) the result to a 2D frame buffer.

Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel frame buffer, then use that frame buffer as a texture map as input to the pixel fragment shader of the next stage of the computation. There was also no support for general user-defined data types—most data had to be stored in one-, two-, or four-component vector arrays. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called GPGPU, for general-purpose computing on GPUs.

2.3 GPU COMPUTING

While developing the Tesla GPU architecture, NVIDIA realized its potential usefulness would be much greater if programmers could think of the
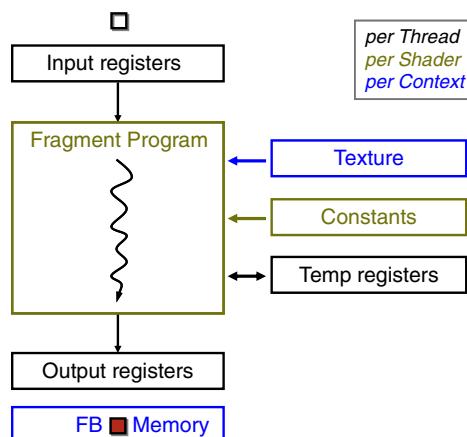
FIGURE 2.6

The restricted input and output capabilities of a shader programming model.

GPU like a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

For the DirectX 10−generation graphics, NVIDIA had already begun work on a high-efficiency floating-point and integer processor that could run a variety of simultaneous workloads to support the logical graphics pipeline. The designers of the Tesla architecture GPUs took another step. The shader processors became fully programmable processors with instruction memory, instruction cache, and instruction sequencing control logic. The cost of these additional hardware resources was reduced by having multiple shader processors to share their instruction cache and instruction sequencing control logic. This design style works well with graphics applications because the same shader program needs to be applied to a massive number of vertices or pixels. NVIDIA added memory load and store instructions with random byte addressing capability to support the requirements of compiled C programs. To nongraphics application programmers, the Tesla architecture GPUs introduced a more generic parallel programming model with a hierarchy of parallel threads, barrier synchronization, and atomic operations to dispatch and manage highly parallel computing work. NVIDIA also developed the CUDA C/C++ compiler, libraries, and runtime software to enable programmers to readily access the new data-parallel computation model and develop applications. Programmers no longer need to use the graphics API to access the GPU parallel computing capabilities. The G80 chip was based on the Tesla architecture and was used in NVIDIA's GeForce 8800 GTX. G80 was followed later by G92, GT200, Fermi, and Kepler.

Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. In the early days, workstation graphics systems gave customers a choice in pixel horsepower by varying the number of pixel processor circuit boards installed. Prior to the mid-1990s, PC graphics scaling was almost nonexistent. There was one option—the VGA controller. As 3D-capable accelerators appeared, there was room in the market for a range of offerings; for instance, 3dfx introduced multiboard scaling with the original SLI (scan line interleave) on their Voodoo2, which held the performance crown for its time (1998). Also in 1998, NVIDIA introduced distinct products as variants on a single architecture with Riva TNT Ultra (high performance) and Vanta (low cost), first by speed binning and

packaging, then with separate chip designs (GeForce 2 GTS and GeForce 2 MX). At present, for a given architecture generation, four or five separate chip designs are needed to cover the range of desktop PC performance and price points. In addition, there are separate segments in notebook and workstation systems. After acquiring 3dfx, NVIDIA continued the multi-GPU SLI concept in 2004 starting with GeForce 6800, providing multi-GPU scalability transparently to both the programmer and to the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

By switching to the multicore trajectory, CPUs are scaling to higher transistor counts by increasing the number of constant-performance cores on a die, rather than increasing the performance of a single core. At this writing the industry is transitioning from quad-core to oct-core CPUs. Programmers are forced to find four-fold to eight-fold parallelism to fully utilize these processors. Many of them resort to coarse-grained parallelism strategies where different tasks of an application are performed in parallel. Such applications must be rewritten often to have more parallel tasks for each successive doubling of core count. In contrast, the highly multi-threaded GPUs encourage the use of massive, fine-grained data parallelism in CUDA. Efficient threading support in GPUs allows applications to expose a much larger amount of parallelism than available hardware execution resources with little or no penalty. Each doubling of GPU core count provides more hardware execution resources that exploit more of the exposed parallelism for higher performance. That is, the GPU parallel programming model for graphics and parallel computing is designed for transparent and portable scalability. A graphics program or CUDA program is written once, and runs on a GPU with any number of processors.

Recent Developments

Academic and industrial work on applications using CUDA has produced hundreds of examples of successful CUDA programs. Many of these examples are presented in GPU *Computing Gems*, Emerald and Jade editions [Hwu2011a, Hwu2011b] with source code available at *www.gpucomputing. net*. These programs often run tens of times faster on a CPU−GPU system than on a CPU alone. With the introduction of tools like MCUDA [SSH2008], the parallel threads of a CUDA program can also run efficiently on a multicore CPU, although at a lower speed than GPUs due to a lower level of floating-point execution resources. Examples of these applications include *n*-body simulation, molecular modeling, computational finance, and

oil/gas reservoir simulation. Although many of these use single-precision floating-point arithmetic, some problems require double precision. The high-throughput double-precision floating-point arithmetic in more recent Fermi and Kepler GPUs enabled an even broader range of applications to benefit from GPU acceleration.

Future Trends

Naturally, the number of processor cores will continue to increase in proportion to increases in available transistors as silicon processes improve. In addition, GPUs will continue to go through vigorous architectural evolution. Despite their demonstrated high performance on data-parallel applications, GPU core processors are still of relatively simple design. More aggressive techniques will be introduced with each successive generation to increase the actual utilization of the calculating units. Because scalable parallel computing on GPUs is a still a young field, novel applications are rapidly being created. By studying them, GPU designers will continue to discover and implement new machine optimizations.

References and Further Reading

Akeley, K., & Jermoluk, T. (1988). High-Performance polygon rendering. *Proc. SIGGRAPH 1988*, 239−246.

Akeley, K. (1993). RealityEngine graphics. *Proc. SIGGRAPH 1993*, 109−116.

Blelloch, G. B. (1990). Prefix sums and their applications. In J. H. Reif (Ed.), *Synthesis of parallel algorithms*. San Francisco: Morgan Kaufmann.

Blythe, D. (2006). The direct3D 10 system. *ACM Trans Graphics*, *25*(3), 724−734.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahlian, K., & Houston, M., et al. (2004). Brook for GPUs: Stream computing on graphics hardware. *Proc. SIGGRAPH 2004*, 777−786 also Available at: <*http://doi.acm.org/10.1145/1186562.1015800*>

Elder, G. (2002). "Radeon 9700," eurographics/SIGGRAPH workshop on graphics hardware. *Hot3D Session*, Available at: <*http://www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt*>

Fernando, R., & Kilgard, M. J. (2003). *The Cg tutorial: The definitive guide to programmable real-time graphics.* Reading, MA: Addison-Wesley.

Fernando, R. (Ed.), (2004). *Gpu gems: Programming techniques, tips, and tricks for real-time graphics.* Reading, MA: Addison-Wesleyalso Available at: <*http://developer.nvidia.com/object/gpu_gems_home.html*>.

Foley, J., van Dam, A., Feiner, S., & Hughes, J. (1995). *Computer graphics: Principles and practice, second edition in C.* Reading, MA: Addison-Wesley.

Hillis, W. D., & Steele, G. L. (1986). Data parallel algorithms. *Commun. ACM*, *29*(12), 1170−1183 <http://doi.acm.org/10.1145/7902.7903>

IEEE 754R working group. *DRAFT standard for floating-point arithmetic P754.* <http://www.validlab.com/754R/drafts/archive/2006-10-04.pdf>.

Industrial light and magic (2003). *OpenEXR*, Available at: <//www.openexr.com>

Intel Corporation (2007). *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Available at: <*http://www3.intel.com/design/processor/manuals/248966.pdf*>

Kessenich, J. (2006). *The Opengl Shading Language, Language Version 1.20*, Available at: <*http://www.opengl.org/documentation/specs/*>

Kirk, D., & Voorhies, D. (1990). The rendering architecture of the DN10000VS. *Proc. SIGGRAPH 1990*, 299−307.

Lindholm, E., Kilgard, M. J., & Moreton, H. (2001). A user-programmable vertex engine. *Proc. SIGGRAPH 2001*, 149−158.

Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, *28*(2), 39−55.

Microsoft corporation. Microsoft DirectX Specification, Available at: <http://msdn.microsoft.com/directx/>.

Microsoft Corporation (2003). *Microsoft directx 9 programmable graphics pipeline* Readmond, WA: Microsoft Press.

Montrym, J., Baum, D., Dignam, D., & Migdal, C. (1997). InfiniteReality: A real-time graphics system. *Proc. SIGGRAPH 1997*, 293−301.

Montrym, J., & Moreton, H. (2005). The GeForce 6800. *IEEE Micro*, *25*(2), 41−51.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8)Avaialble at <http://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf>

Nguyen, H. (Ed.), (2008). *GPU gems 3* Reading, MA: Addison-Wesley.

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue*, *6*(2), 40−53.

NVIDIA (2012). *CUDA Zone*, Available at: *http://www.nvidia.com/CUDA*

NVIDIA (2007). *CUDA Programming Guide 1.1*, Available at: <http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf>

NVIDIA (2007). *PTX: Parallel Thread Execution ISA Version 1.1*, Available at: <*http://www.nvidia.com/object/io_1195170102263.html*>

Nyland, L., Harris, M., & Prins, J. (2007). Fast N-Body simulation with CUDA. In H. Nguyen (Ed.), *GPU gems 3*. Reading, MA: Addison-Wesley.

Oberman, S. F.and Siu,M. Y. A high-performance area-efficient multifunction interpolator. *Proc. 17th IEEE symp. computer arithmetic* (pp. 272−279). Seattle Washington, 2005.

Patterson, D. A., & Hennessy, J. L. (2004). *Computer organization and design: The hardware/software interface* (3rd ed.) San Francisco: Morgan Kaufmann.

Pharr, M. (Ed.), (2005). *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation.* Reading, MA: Addison-Wesley.

Satish, N. Harris, M. and Garland, M. Designing efficient sorting algorithms for proceedings of the 23rd ieee international parallel and distributed processing symposium. Rome, Italy, 2009

Segal, M., & Akeley, K. (2006). *The opengl graphics system: A specification, version 2.1*, Available at: <http://www.opengl.org/documentation/specs/>

Sengupta, S. Harris, M. Zhang, Y.and Owens, J. D. Scan primitives for GPU computing. *Proc. of graphics hardware 2007* (pp. 97−106). San Diego, California, Aug. 2007.

Hwu, W. (Ed.), (2011a). GPU computing gems, *emerald edition* San Francisco: Morgan Kauffman.

Hwu, W. (Ed.), (2011b). GPU computing gems*, jade edition* San Francisco: Morgan Kauffman.

Stratton, J. A., Stone, S. S., & Hwu, W. W. (2008). MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. *The 21st International Workshop on Languages and Compilers for Parallel Computing*, [Canada; also Available as Lecture Notes in Computer Science]

Volkov V.and Demmel, J. LU, QR and cholesky factorizations using vector capabilities of GPUs. Technical Report No. UCB/EECS-2008-49, 1−11; also Available at: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html>.

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., & Demmel, J. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Proc. Supercomputing 2007 (SC'07).* doi:10.1145/1362622.1362674 [Reno, Nevada]

This page intentionally left blank

Introduction to Data Parallelism and CUDA C

<div style="text-align:right; font-size:3em;">3</div>

CHAPTER OUTLINE

Our main objective is to teach the key concepts involved in writing massively parallel programs in a heterogeneous computing system. This requires many code examples expressed in a reasonably simple language that supports massive parallelism and heterogeneous computing. We have chosen CUDA C for our code examples and exercises. CUDA C is an extension to the popular C programming language[1] with new keywords and application programming interfaces for programmers to take advantage of heterogeneous computing systems that contain both CPUs and massively parallel GPU's. For the rest of this book, we will refer to CUDA C simply as CUDA. To a CUDA programmer, the computing system consists of a *host* that is a traditional CPU, such as an Intel architecture microprocessor in personal computers today, and one or more *devices* that are processors with a massive number of arithmetic units. A CUDA device is typically a GPU. Many modern software applications have sections that exhibit a rich amount of data parallelism, a phenomenon that allows arithmetic operations

---

[1]CUDA C also supports a growing subset of C++ features. Interested readers should refer to the *CUDA Programming Guide* for more information about the supported C++ features.

to be safely performed on different parts of the data structures in parallel. CUDA devices accelerate the execution of these applications by applying their massive number of arithmetic units to these data-parallel program sections. Since data parallelism plays such an important role in CUDA, we will first discuss the concept of data parallelism before introducing the basic features of CUDA.

3.1 DATA PARALLELISM

Modern software applications often process a large amount of data and incur long execution time on sequential computers. Many of them operate on data that represents or models real-world, physical phenomena. Images and video frames are snapshots of a physical world where different parts of a picture capture simultaneous, independent physical events. Rigid-body physics and fluid dynamics model natural forces and movements that can be independently evaluated within small time steps. Airline scheduling deals with thousands of flights, crews, and airport gates that operate in parallel. Such independent evaluation is the basis of data parallelism in these applications.
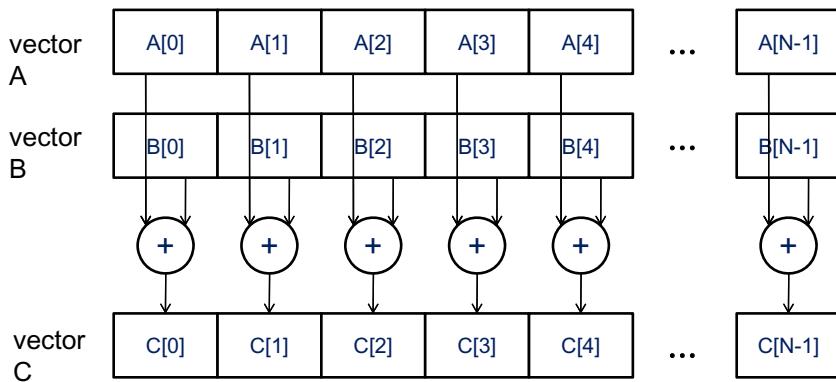
TASK PARALLELISM VERSUS DATA PARALLELISM

Data parallelism is not the only type of parallelism widely 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.

In large applications, there are usually a larger number of independent tasks and therefore a larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks includes vibrational forces, rotational forces, neighbor identification for nonbonding forces, nonbonding 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 data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce CUDA streams.

Let us illustrate the concept of data parallelism with a vector addition example in Figure 3.1. In this example, each element of the sum vector C is generated by adding an element of input vector A to an element of input vector B. For example, C[0] is generated by adding A[0] to B[0], and C[3] is generated by adding A[3] to B[3]. All additions can be performed in

FIGURE 3.1
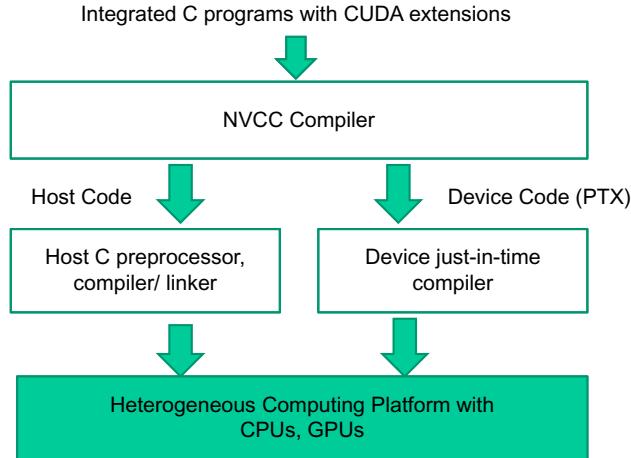
Data parallelism in vector addition.

parallel. Therefore, vector addition of two large vectors exhibits a rich amount of data parallelism. Data parallelism in real applications can be more complex and will be discussed in detail later.

3.2 CUDA PROGRAM STRUCTURE

The structure of a CUDA program reflects the coexistence of a *host* (CPU) and one or more *devices* (GPUs) in the computer. Each CUDA source file can have a mixture of both host and device code. By default, any traditional C program is a CUDA program that contains only host code. One can add device functions and data declarations into any C source file. The function or data declarations for the device are clearly marked with special CUDA keywords. These are typically functions that exhibit a rich amount of data parallelism.

Once device functions and data declarations are added to a source file, it is no longer acceptable to a traditional C compiler. The code needs to be compiled by a compiler that recognizes and understands these additional declarations. We will be using a CUDA C compiler by NVIDIA called NVCC (NVIDIA C Compiler). As shown at the top of Figure 3.2, the NVCC processes a CUDA program, using the CUDA keywords to separate the host code and device code. The host code is straight ANSI C code, which is further compiled with the host's standard C/C++ compilers and is run as a traditional CPU process. The device code is marked with CUDA keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. The device code is further compiled by a runtime component of NVCC and

FIGURE 3.2
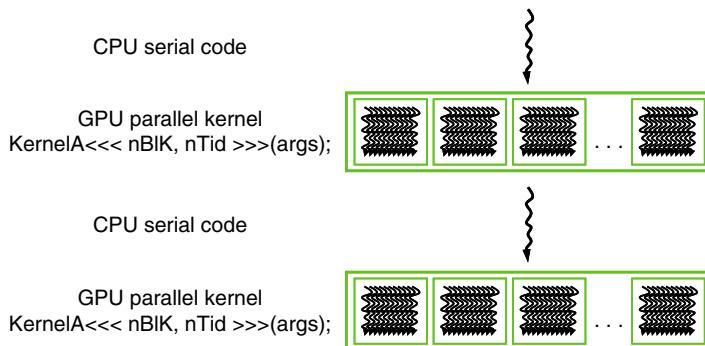
Overview of the compilation process of a CUDA program.

executed on a GPU device. In situations where there is no device available or a kernel can be appropriately executed on a CPU, one can also choose to execute the kernel on a CPU using tools like MCUDA [Stratton 2008].

The execution of a CUDA program is illustrated in Figure 3.3. The execution starts with host (CPU) execution. When a kernel function is called, or *launched*, it is executed by a large number of threads on a device. All the threads that are generated by a kernel launch are collectively called a *grid*. Figure 3.3 shows the execution of two grids of threads. We will discuss how these grids are organized soon. When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is launched. Note that Figure 3.3 shows a simplified model where the CPU execution and the GPU execution do not overlap. Many heterogeneous computing applications actually manage overlapped CPU and GPU execution to take advantage of both CPUs and GPUs.

THREADS

A thread is a simplified view of how a processor executes a program in modern computers. A thread consists of the code of the program, the particular 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

FIGURE 3.3

Execution of a CUDA program.

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.

Threads have been used in traditional CPU programming for many years. If a programmer wants to start parallel execution in an application, he or she needs to create and manage 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 launching kernel functions, which causes the underlying runtime mechanisms to create many threads that process different parts of the data in parallel.

Launching a kernel typically generates a large number of threads to exploit data parallelism. In the vector addition example, each thread can be used to compute one element of the output vector C. In this case, the number of threads that will be generated by the kernel is equal to the vector length. For long vectors, a large number of threads will be generated. CUDA programmers can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support. This is in contrast with traditional CPU threads that typically take thousands of clock cycles to generate and schedule.

3.3 A VECTOR ADDITION KERNEL

We now use vector addition to illustrate the CUDA programming model. Before we show the kernel code for vector addition, it is helpful to first review how a conventional CPU-only vector addition function works. Figure 3.4 shows a simple traditional C program that consists of a main function and a vector addition function. In each piece of host code, we will prefix the names of variables that are mainly processed by the host

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each

    …
    vecAdd(h_A, h_B, h_C, N);
}
```

FIGURE 3.4

A simple traditional vector addition C code example.

with h_ and those of variables that are mainly processed by a device d_ to remind ourselves the intended usage of these variables.
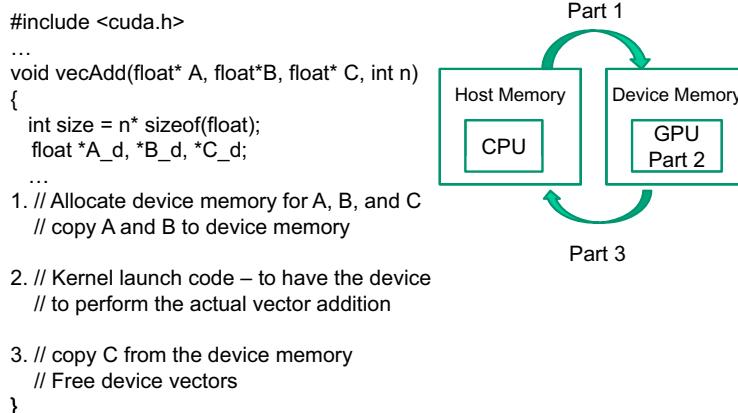
Assume that the vectors to be added are stored in arrays h_A and h_B that are allocated and initialized in the main program. The output vector is in array h_C, which is also initialized in the main program. For brevity, we do not show the details of how h_A, V, and h_C are allocated or initialized. A complete source code listing that contains more details is available in Appendix A. The pointers to these arrays are passed to the vecAdd() function, along with the variable N that contains the length of the vectors.

POINTERS IN THE C LANGUAGE

The function arguments A, B, and C in Figure 3.4 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 wit:

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 0th element. For example, the statement P = & (h_A[O]) makes P point to the 0th element of array h_A. P[i] becomes a synonym for h_A[i]. In fact, the array name h_A is in itself a pointer to its 0th element. In Figure 3.4, passing an array name h_A as the first argument to function call to vecAdd makes the function's first parameter A point to the 0th element of h_A . We say that h_A is passed by reference to vecAdd. As a result, A [i] in the function body can be used to access h_A[i]. See Patt& Patel [Patt] for an easy-to-follow explanation of the detailed usage of pointers in C.

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
    int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
    …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

Part 1

Host Memory   Device Memory

CPU   GPU
                 Part 2

Part 3
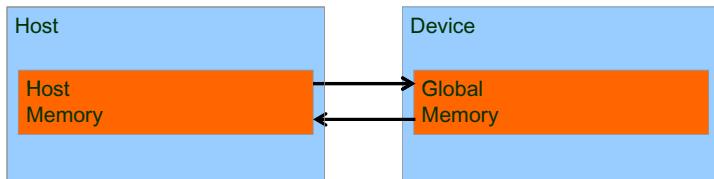
FIGURE 3.5

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

The vecAdd() function in Figure 3.4 uses a for loop to iterate through the vector elements. In the *i*th iteration, output element C[i] receives the sum of A[i] and B[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 parameters A, B, and C are passed by reference so the function reads the elements of h_A, h_B and writes the elements of h_C through the parameter pointers A, B, and C. When the vecAdd() function returns, the subsequent statements in the main function can access the new contents of h_C.

A straightforward way to execute vector addition in parallel is to modify the vecAdd() function and move its calculations to a CUDA device. The structure of such a modified vecAdd() function is shown in Figure 3.5. At the beginning of the file, we need to add a C preprocessor directive to include the CUDA.h header file. This file defines the CUDA API functions and built-in variables that we will be introducing soon. 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 vectors from the host memory to the device memory. Part 2 launches parallel execution of the actual vector addition kernel on the device. Part 3 copies the sum vector C from the device memory back to the host 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. The details of the revised

FIGURE 3.6

Host memory and device global memory.

function, as well as the way to compose the kernel function, will be shown as we introduce the basic features of the CUDA programming model.

3.4 DEVICE GLOBAL MEMORY AND DATA TRANSFER

In CUDA, host and devices have separate memory spaces. This reflects the current reality that devices are often hardware cards that come with their own DRAM. For example, the NVIDIA GTX480 comes with up to 4 GB[2] (billion bytes, or gigabytes) of DRAM, called global memory. We will also refer to global memory as device memory. To execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 of Figure 3.5. Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed. This corresponds to Part 3 of Figure 3.5. The CUDA runtime system provides Application Programming Interface (API) functions to perform these activities on behalf of the programmer. From this point on, we will simply say that a piece of data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device memory. The same holds for the opposite direction.

Figure 3.6 shows a CUDA host memory and device memory model for programmers to reason about the allocation of device memory and movement of memory between host and device. The device global memory can be accessed by the host to transfer data to and from the device, as illustrated by the bidirectional arrows between these memories and the

---

[2]There is a trend to integrate CPUs and GPUs into the same chip package, commonly referred to as *fusion*. Fusion architectures often have a unified memory space for host and devices. There are new programming frameworks, such as GMAC, that take advantage of the unified memory space and eliminate data copying costs.

host in Figure 3.6. There are more device memory types than shown in Figure 3.6. Constant memory can be accessed in a read-only manner by device functions, which will be described in Chapter 8. We will also discuss the use of registers and shared memory in Chapter 5. See the *CUDA Programming Guide* for the functionality of texture memory. For now, we will focus on the use of global memory.

The CUDA runtime system provides API functions for managing data in the device memory. For example, Parts 1 and 3 of the `vecAdd()` function in Figure 3.5 need to use these API functions to allocate device memory for A, B, and C; transfer A and B from host memory to device memory; transfer C from device memory to host memory; and free the device memory for A, B, and C. We will explain the memory allocation and free functions first. Figure 3.7 shows two API functions for allocating and freeing device global memory. Function `cudaMalloc()` can be called from the host code to allocate a piece of device global memory for an object. Readers should notice the striking similarity between `cudaMalloc()` and the standard C runtime library `malloc()`. This is intentional; CUDA is C with minimal extensions. CUDA uses the standard C runtime library `malloc()` function to manage the host memory 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 minimizes the time that a C programmer spends to relearn the use of these extensions.

The first parameter to the `cudaMalloc()` function is the address of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to `(void **)` because the function expects a generic pointer; the memory allocation function is a generic function that is not restricted to any particular type of objects.[3] This parameter allows the `cudaMalloc()` function to write the address of the allocated memory into the pointer variable.[4] The host code passes this pointer value to the kernels that need to access the allocated memory

---

[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 4.2.

[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 of which the 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.

- cudaMalloc()
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size** of allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memoryv
    - **Pointer** to freed object

FIGURE 3.7

CUDA API functions for managing device global memory.

object. The second parameter to the `cudaMalloc()` function gives the size of the data to be allocated, in terms of bytes. The usage of this second parameter is consistent with the size parameter to the C `malloc()` function.

We now use a simple code example to illustrate the use of `cudaMalloc()`. This is a continuation of the example in Figure 3.5. For clarity, we will start a pointer variable with `d_` to indicate that it points to an object in the device memory. The program passes the address of `d_A` (i.e., `&d_A`) as the first parameter after casting it to a void pointer. That is, `d_A` will point to the device memory region allocated for the A vector. The size of the allocated region will be *n* times the size of a single-precision floating number, which is 4 bytes in most computers today. After the computation, `cudaFree()` is called with pointer `d_A` as input to free the storage space for the A vector from the device global memory.

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

The addresses in `d_A`, `d_B`, and `d_C` are addresses in the device memory. These addresses should not be dereferenced in the host code. They should be mostly used in calling API functions and kernel functions. Dereferencing a device memory point in the host code can cause exceptions or other types of runtime errors during runtime.

Readers should complete Part 1 of the `vecAdd()` example in Figure 3.5 with similar declarations of `d_B` and `d_C` pointer variables as well as their corresponding `cudaMalloc()` calls. Furthermore, Part 3 in Figure 3.6 can be completed with the `cudaFree()` calls for `d_B` and `d_C`.

cudaMemcpy()
- – memory data transfer
- – Requires four parameters
  - • Pointer to destination
  - • Pointer to source
  - • Number of bytes copied
  - • Type/Direction of transfer

FIGURE 3.8

CUDA API function for data transfer between host and device.

Once the host code has allocated device 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. Figure 3.8 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 memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory. For example, the cudaMemcpy() function can be used to copy data from one location of the device memory to another location of the device memory.[5]

ERROR HANDLING IN CUDA

In general, it is very 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, line 1 in Figure 3.9 shows a call to cudaMalloc():

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

In practice, we should surround the call with code that tests for error conditions and prints 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 **) &d_A, size);
```

---

[5]Please note cudaMemcpy() cannot be used to copy between different GPUs in multi-GPU systems.

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

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

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 3.9

A more complete version of `vecAdd()`.

```
if (err != 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.

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

The `vecAdd()` function calls the `cudaMemcpy()` function to copy A and B vectors from host to device before adding them and to copy the C vector from the device to host after the addition is done. Assume that the value of A, B, d_A, d_B, and size have already been set as we discussed before; the three `cudaMemcpy()` calls are shown below. The two symbolic constants, `cudaMemcopyHostToDevice` and `cudaMemcopyDeviceToHost`, 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(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
```

To summarize, the main program in Figure 3.4 calls vecAdd(), which is also executed on the host. The vecAdd() function, outlined in Figure 3.5, allocates device memory, requests data transfers, and launches the kernel that performs the actual vector addition. We often refer to this type of host code as a *stub function* for launching a kernel. After the kernel finishes execution, vecAdd() also copies result data from device to the host. We show a more complete version of the vecAdd() function in Figure 3.9.

Compared to Figure 3.6, the vecAdd() function in Figure 3.9 is complete for Parts 1 and 3. Part 1 allocates device memory for d_A, d_B, and d_C and transfers A to d_A and B to d_B. This is done by calling the cudaMalloc() and cudaMemcpy() functions. Readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Figure 3.9. Part 2 invokes the kernel and will be described in the following section. Part 3 copies the sum data from device memory to host memory so that the value will be available to main(). This is accomplished with a call to the cudaMemcpy() function. It then frees the memory for d_A, d_B, and d_C from the device memory, which is done by calls to the cudaFree() function.
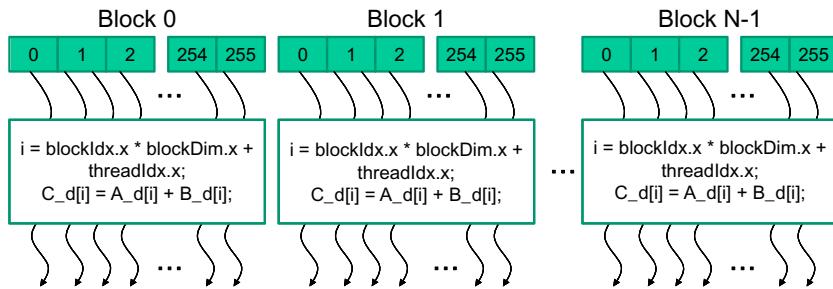
3.5 KERNEL FUNCTIONS AND THREADING

We are now ready to discuss more about the CUDA kernel functions and the effect of launching these kernel functions. In CUDA, 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 programming is an instance of the well-known SPMD (single program, multiple data) [Atallah1998] parallel programming style, a popular programming style for massively parallel computing systems.[6]

When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy. Each

---

[6]Note that SPMD is not the same as SIMD (single instruction, multiple data) [Flynn1972]. 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.

FIGURE 3.10

All threads in a grid execute the same kernel code.

grid is organized into an array of thread blocks, which will be referred to as blocks for brevity. All blocks of a grid are of the same size; each block can contain up to 1,024 threads.[7] Figure 3.10 shows an example where each block consists of 256 threads. The number of threads in each thread block is specified by the host code when a kernel is launched. The same kernel can be launched 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 the blockDim variable. In Figure 3.10, the value of the blockDim.x variable is 256. In general, the dimensions of thread blocks should be multiples of 32 due to hardware efficiency reasons. We will revisit this later.

Each thread in a block has a unique threadIdx value. For example, the first thread in block 0 has value 0 in its threadIdx variable, the second thread has value 1, the third thread has value 2, etc. This allows each thread to combine its threadIdx and blockIdx values to create a unique global index for itself with the entire grid. In Figure 3.10, a data index i is calculated as $i = blockIdx.x * blockDim.x + threadIdx.x$. Since blockDim is 256 in our example, the i values of threads in block 0 ranges 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 d_A, d_B, and d_C, these threads cover the first 768 iterations of the original loop. By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with *n* or more threads, one can process vectors of length *n*.

---

[7]Each thread block can have up to 1,024 threads in CUDA 3.0 and later. Some earlier CUDA versions allow only up to 512 threads in a block.

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

FIGURE 3.11

A vector addition kernel function and its launch statement.

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

FIGURE 3.12

CUDA C keywords for function declaration.

Figure 3.11 shows a kernel function for a vector addition. The syntax is ANSI C with some notable extensions. First, there is a CUDA specific keyword __global__ in front of the declaration of vecAddKernel(). This keyword indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device.

In general, CUDA extends C language with three qualifier keywords in function declarations. The meaning of these keywords is summarized in Figure 3.12. The __global__ keyword indicates that the function being declared is a CUDA kernel function. Note that there are two underscore characters on each side of the word "global." A __global__ function is to be executed on the device and can only be called from the host code. The __device__ keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can only be called from a kernel function or another device function.[8]

---

[8]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.

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 only be called from another host function. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration. This makes sense since many CUDA applications are ported from CPU-only execution environments. The programmer would add kernel functions and device functions during the porting process. The original functions remain as host functions. Having all functions to default into host functions spares the programmer the tedious work to change all original function declarations.

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 files for the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function. This 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 ANSI C in Figure 3.10 is the keywords threadIdx.x. blockIdx.x, and blockDim.x. Note that all threads execute the same kernel code. There needs to be a way for them to distinguish among themselves and direct each thread toward a particular part of the data. These keywords identify predefined variables that correspond to hardware registers that provide the identifying coordinates to threads. Different threads will see different values in their threadIdx.x, blockIdx.x, and blockDim.x variables. For simplicity, we will refer to a thread as thread$_{blockIdx.x, threadIdx.x}$. Note that the .x implies that there might be .y and .z. We will come back to this point soon.

There is an automatic (local) variable i in Figure 3.11. 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 kernel 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 again in Chapter 5.

A quick comparison between Figure 3.4 and Figure 3.11 reveals an important insight for CUDA kernels and a CUDA kernel launch. The kernel function in Figure 3.11 does not have a loop that corresponds to the one in Figure 3.4. Readers should ask where the loop went. The answer is that the loop is now replaced with the grid of threads. The entire

```
int vectAdd(float* A, float* B, float* C, int n)
{
// d_A, d_B, d_C allocations and copies omitted
// Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

FIGURE 3.13

A vector addition kernel function and its launch statement.

grid forms the equivalent of the loop. Each thread in the grid corresponds to one iteration of the original loop.

Note that there is an if (i<n) statement in addVecKernel() in Figure 3.11. This is because not all vector lengths can be expressed as multiples of the block size. For example, if the vector length is 100, the smallest efficient thread block dimension is 32. Assume that we picked 32 as the block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their i values against n, which is 100. With the if (i<n) statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.

When the host code launches a kernel, it sets the grid and thread block dimensions via *execution configuration* parameters. This is illustrated in Figure 3.13. The configuration parameters are given between the <<< and >>> before the traditional C function arguments. The first configuration parameter gives the number of thread blocks in the grid. The second specifies the number of threads in each thread block. In this example, there are 256 threads in each block. To ensure that we have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we have 1,000 threads, we would launch ceil(1,000/256.0) = 4 thread blocks. As a result, the statement will launch $4 \times 256 = 1,024$ threads. With the if (i < n) statement in the kernel as shown in Figure 3.11, the first 1,000 threads will perform addition on the 1,000 vector elements. The remaining 24 will not.

Figure 3.14 shows the final host code of vecAdd(). This source code completes the skeleton in Figure 3.5. Figures 3.10 and 3.14 jointly illustrate a simple CUDA program that consists of both the host code and a device

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

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

    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
        // Free device memory for A, B, C
     cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 3.14

A complete version of `vecAdd()`.

kernel. The code is hardwired to use thread blocks of 256 threads each. The number of thread blocks used, however, depends on the length of the vectors (n). If n is 750, three thread blocks will be used; if n is 4,000, 16 thread blocks will be used; if n is 2,000,000, 7,813 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. A small GPU with a small amount of execution resources may execute 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, same code runs at lower performance on small GPUs and higher performance on larger GPUs. We will revisit this point again in Chapter 4.

It is important to point out 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 de-allocating device memory will likely make the resulting code slower than the original sequential code in Figure 3.4. This is because the amount of calculation done by the kernel is small relative to the amount of data processed. Only one addition is performed for two floating-point input operands and one floating-point output operand. Real applications typically have kernels where much more work is needed relative to the amount of data processed, which makes the additional overhead worthwhile. They 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.

3.6 SUMMARY

This chapter provided a quick overview of the CUDA programming model. CUDA extends the C language to support parallel computing. We discussed a subset of these extensions in this chapter. For your convenience, we summarize the extensions that we have discussed in this chapter as follows.

Function Declarations

CUDA extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Figure 3.12. Using one of `__global__`, `__device__`, or `__host__`, a CUDA 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 are defaulted 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 extension keyword, the function defaults into a host function.

Kernel Launch

CUDA extends C function call syntax with kernel execution configuration parameters surrounded by $<<<$ and $>>>$. These execution configuration parameters are only used during a call to a kernel function, or a kernel launch. We discussed the execution configuration parameters that define the dimensions of the grid and the dimensions of each block. Readers should refer to the *CUDA Programming Guide* [NVIDIA2011] for more details of the kernel launch extensions as well as other types of execution configuration parameters.

Predefined Variables

CUDA kernels can access a set of predefined variables that allow each thread to distinguish among themselves and to determine the area of data each thread is to work on. We discussed the `threadIdx`, `blockDim`, and

`blockIdx` variables in this chapter. In Chapter 4, we will discuss more details of using these variables.

Runtime API

CUDA supports a set of API functions to provide services to CUDA programs. The services that we discussed in this chapter are the `cudaMalloc()`, `cudaFree()`, and `cudaMemcpy()` functions. These functions allocate device memory and transfer data between host and device on behalf of the calling program. Readers are referred to the *CUDA Programming Guide* [NVIDIA2011] for other CUDA API functions.

Our goal for this chapter is to introduce the core concepts of the CUDA programming model and the essential CUDA extensions to C for writing a simple CUDA program. The chapter is by no means a comprehensive account of all CUDA features. Some of these features will be covered in the remainder of the book. However, our emphasis will be on key concepts rather than details. We will only introduce enough CUDA features that are needed in our code examples for parallel programming techniques. In general, we would like to encourage readers to always consult the *CUDA Programming Guide* for more details of the CUDA features.

3.7 EXERCISES

3.1. A matrix addition takes two input matrices B and C and produces one output matrix A. Each element of the output matrix A is the sum of the corresponding elements of the input matrices B and C, that is, A[i][j] = = B[i][j] + C[i][j]. For simplicity, we will only handle square matrices of which the elements are single-precision floating-point numbers. Write a matrix addition kernel and the host stub function that can be called with four parameters: pointer to the output matrix, pointer to the first input matrix, pointer to the second input matrix, and the number of elements in each dimension. Use the following instructions:

   a. Write the host stub function by allocating memory for the input and output matrices, transferring input data to device, launch the kernel, transferring the output data to host, and freeing the device memory for the input and output data. Leave the execution configuration parameters open for this step.

    b. Write a kernel that has each thread producing one output matrix element. Fill in the execution configuration parameters for the design.

    c. Write a kernel that has each thread producing one output matrix row. Fill in the execution configuration parameters for the design.

    d. Write a kernel that has each thread producing one output matrix column. Fill in the execution configuration parameters for the design.

    e. Analyze the pros and cons of each preceding kernel design.

3.2. 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 only handle square matrices of which the 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.

3.3. 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?

3.4. Complete Parts 1 and 2 of the function in Figure 3.6.

3.5. If we need to use each thread to calculate one output element of a vector addition, what would be the expression for mapping the thread/block indices to data index:

    (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;

3.6. We want to use each thread to calculate two (adjacent) elements of a vector addition, 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?

(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

3.7. 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?

    (A) 2000

    (B) 2024

    (C) 2048

    (D) 2096

References

Atallah, M. J. (Ed.), (1998). *Algorithms and Theory of Computation Handbook* Baco Raton, FL: CRC Press.

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

NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, version 4.2, April 2012, Available at: <http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf>.

Patt, Y. N., & Patel, S. J. (1972). *Introduction to Computing Systems: From Bits and Gates to C and Beyond,* New York: McGraw-Hill.

Stratton, J. A., Stone, S. S., & Hwu, W. W. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs, *The 21st International Workshop on Languages and Compilers for Parallel Computing*, July 30−31, Canada, 2008. Also available as Lecture Notes in *Computer Science,* 2008.

Data-Parallel Execution Model

4

CHAPTER OUTLINE

Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA. As we explained in Chapter 3, launching a CUDA kernel creates a grid of threads that all execute the kernel function. That is, the kernel function specifies the C statements that are executed by each individual thread at runtime. Each thread uses a unique coordinate, or thread index, to identify the portion of the data structure to process. The thread index can be organized multidimensionally to facilitate access to multidimensional arrays. This chapter presents more details on the organization, resource assignment, synchronization, and scheduling of threads in a grid. A CUDA programmer who understands these details is well equipped to express and understand the parallelism in high-performance CUDA applications.

BUILT-IN VARIABLES

Many programming languages have built-in variables. These variables have special meaning and purpose. The values of these variables are often preinitialized by the runtime system.

For example, in a CUDA kernel function, `gridDim`, `blockDim`, `blockIdx`, and `threadIdx` are all built-in variables. Their values are preinitialized by the CUDA runtime systems and can be referenced in the kernel function. The programmers should refrain from using these variables for any other purpose.

4.1 CUDA THREAD ORGANIZATION

Recall from Chapter 3 that all CUDA threads in a grid execute the same kernel function and they rely on coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy: a grid consists of one or more blocks and each block in turn consists of one or more threads. All threads in a block share the same block index, which can be accessed as the `blockIdx` variable in a kernel. Each thread also has a thread index, which can be accessed as the `threadIdx` variable in a kernel. To a CUDA programmer, `blockIdx` and `threadIdx` appear as built-in, pre-initialized variables that can be accessed within kernel functions (see "Built-in Variables" sidebar). When a thread executes a kernel function, references to the `blockIdx` and `threadIdx` variables return the coordinates of the thread. The execution configuration parameters in a kernel launch statement specify the dimensions of the grid and the dimensions of each block. These dimensions are available as predefined built-in variables `gridDim` and `blockDim` in kernel functions.

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 three-digit area code. A telephone area is typically 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 10 million numbers. One can think of each phone line as a CUDA thread, the area code as the CUDA `blockIdx`, and the seven-digital local number as the CUDA `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 do not 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.

In general, a grid is a 3D array of blocks[1] and each block is a 3D array of threads. The programmer can choose to use fewer dimensions by setting the unused dimensions to 1. The exact organization of a grid is determined by the execution configuration parameters (within $<<<$ and $>>>$) of the kernel launch 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 is of `dim3` type, which is a C `struct` with three unsigned integer fields, *x*, *y*, and *z*. These three fields correspond to the three dimensions.

For 1D or 2D grids and blocks, the unused dimension fields should be set to 1 for clarity. For example, the following host code can be used to launch the `vecAddkernel()` kernel function and generate a 1D grid that consists of 128 blocks, each of which consists of 32 threads. The total number of threads in the grid is $128 \times 32 = 4,096$.

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

Note that `dimBlock` and `dimGrid` are host code variables defined by the programmer. These variables can have names as long as they are of `dim3` type and the kernel launch uses the appropriate names. For example, the following statements accomplish the same as the previous statements:

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

The grid and block dimensions can also be calculated from other variables. For example, the kernel launch in Figure 3.14 can be written as:

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

This allows the number of blocks to vary with the size of the vectors so that the grid will have enough threads to cover all vector elements. The value of variable `n` at kernel launch time will determine the dimension of the grid. If `n` is equal to 1,000, the grid will consist of four blocks. If `n` is equal to 4,000, the grid will have 16 blocks. In each case, there will be enough threads to cover all the vector elements. Once `vecAddKernel()` is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

For convenience, CUDA C provides a special shortcut for launching a kernel with 1D grids and blocks. Instead of using `dim3` variables, one can

---

[1]Devices with capability level less than 2.0 support grids with up to 2D arrays of blocks.

use arithmetic expressions to specify the configuration of 1D grids and blocks. In this case, the CUDA C 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 launch statement shown in Figure 3.14:

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

Within the kernel function, the *x* field of the predefined variables `gridDim` and `blockDim` are preinitialized according to the execution configuration parameters. For example, if n is equal to 4,000, references to `gridDim.x` and `blockDim.x` in the `vectAddkernel` kernel function 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` variables in the kernel function always reflect the dimensions of the grid and the blocks.

In CUDA C, the allowed values of `gridDim.x`, `gridDim.y`, and `gridDim.z` range from 1 to 65,536. All threads in a block share the same `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` values. Among all blocks, the `blockIdx.x` value ranges between 0 and `gridDim.x-1`, the `blockIdx.y` value between 0 and `gridDim.y-1`, and the `blockIdx.z` value between 0 and `gridDim.z-1`. For the rest of this book, we will use the notation (*x*, *y*, *z*) for a 3D grid with *x* blocks in the *x* direction, *y* blocks in the *y* direction, and *z* blocks in the *z* direction.
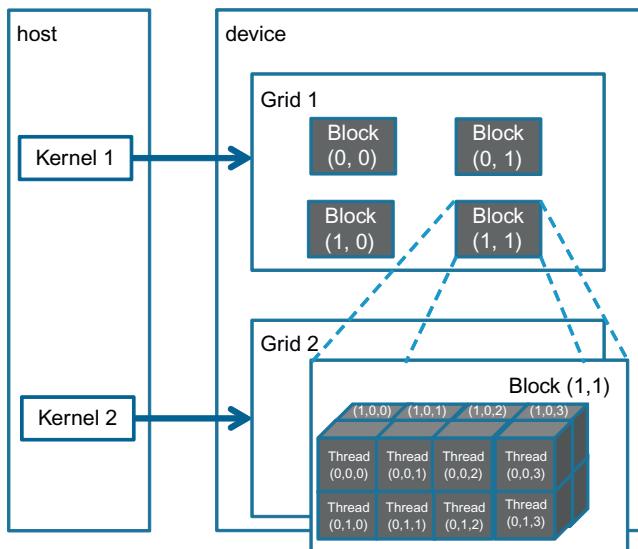
We now turn our attention to the configuration of blocks. Blocks are organized into 3D arrays of threads. Two-dimensional blocks can be created by setting the *z* dimension to 1. One-dimensional blocks can be created by setting both the *y* and *z* dimensions to 1, as in the `vectorAddkernel` example. As we mentioned before, all blocks in a grid have the same dimensions. The number of threads in each dimension of a block is specified by the second execution configuration parameter at the kernel launch. Within the kernel, this configuration parameter can be accessed as the *x*, *y*, and *z* fields of the predefined variable `blockDim`.

The total size of a block is limited to 1,024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1,024. For example, (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowable `blockDim` values, but (32, 32, 2) is not allowable since the total number of threads would exceed 1,024.[2]

Note that the grid can have higher dimensionality than its blocks and vice versa. For example, Figure 4.1 shows a small toy example of a 2D

---

[2]Devices with capability less than 2.0 allow blocks with up to 512 threads.

FIGURE 4.1

A multidimensional example of CUDA grid organization.

(2, 2, 1) grid that consists of 3D (4, 2, 2) blocks. The grid can be gener-ated with the following host code:

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

The grid consists of four blocks organized into a 2×2 array. Each block in Figure 4.1 is labeled with (blockIdx.y, blockIdx.x). For exam-ple, block(1,0) has blockIdx.y = 1 and blockIdx.x = 0. Note that the ordering of the labels is such that the highest dimension comes first. This is reverse of the ordering used in the configuration parameters where the lowest dimension comes first. This reversed ordering for labeling threads works better when we illustrate the mapping of thread coordinates into data indexes in accessing multidimensional arrays.
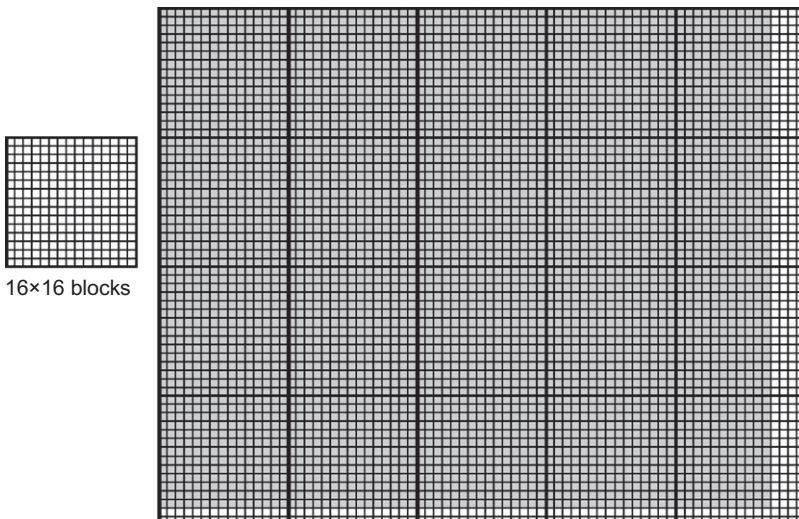
Each threadIdx also consists of three fields: the x coordinate threadId.x, the y coordinate threadIdx.y, and the z coordinate threadIdx.z. Figure 4.1 illustrates the organization of threads within a block. In this example, each block is organized into $4 \times 2 \times 2$ arrays of threads. Since all blocks within a grid have the same dimensions, we only need to show one of them. Figure 4.1 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 four 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.

4.2 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. It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture. Figure 4.2 shows such an arrangement for processing a 76×62 picture (76 pixels in the horizontal or x direction and 62 pixels in the vertical or y 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 five blocks in the x direction and four blocks in the y direction, which results in $5 \times 4 = 20$ blocks as shown in Figure 4.2. The heavy lines mark the block boundaries. The shaded area depicts the threads that cover pixels. Note that we have four extra threads in the x direction and two extra threads in the y direction. That is, we will generate 80×64 threads to process 76×62 pixels. This is similar to the situation where a

16×16 blocks

FIGURE 4.2

Using a 2D grid to process a picture.

1,000-element vector is processed by the 1D `vecAddKernel` in Figure 3.10 using four 256-thread blocks. Recall that an `if` statement is needed to prevent the extra 24 threads from taking effect. Analogously, we should expect that the picture processing kernel function will have `if` statements to test whether the thread indices `threadIdx.x` and `threadIdx.y` fall within the valid range of pixels.

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

```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0), 1);
dim3 dimGrid(16, 16, 1);
pictureKernel<<<dimGrid,dimBlock>>>(d_Pin,d_Pout,n,m);
```

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 $2,000 \times 1,500$ (3 M pixel) picture, we will generate 14,100 blocks, 150 in the *x* direction and 94 in the *y* direction. Within the kernel function, references to built-in variables `gridDim.x`, `gridDim.y`, `blockDim.x`, and `blockDim.y` will result in 150, 94, 16, and 16, respectively.

Before we show the kernel code, we need to first understand how C statements access elements of dynamically allocated multidimensional arrays. Ideally, we would like to access `d_Pin` as a 2D array where an element at row *j* and column *i* can be accessed as `d_Pin[j][i]`. However, the ANSI C standard based on which CUDA C was developed requires that the number of columns in `d_Pin` be known at compile time. Unfortunately, this information is not known at compiler 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 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. Note that the newer C99 standard allows multidimensional syntax for dynamically allocated arrays. It is likely that future CUDA C versions may support multidimensional syntax for dynamically allocated arrays.
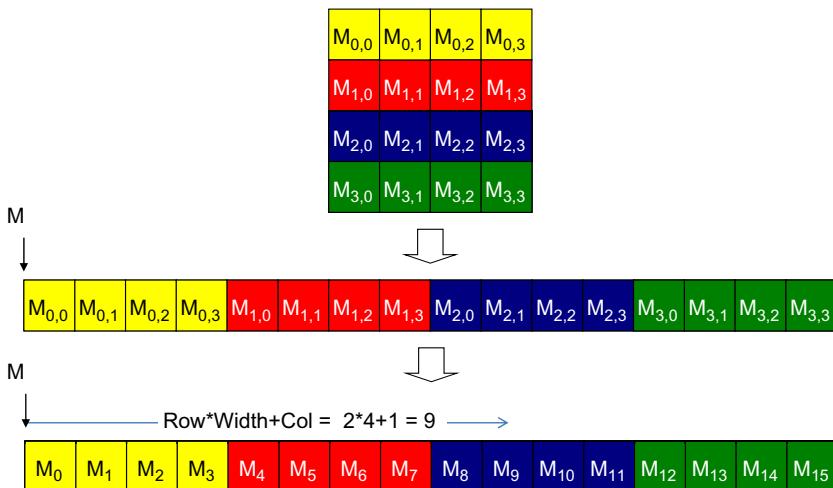
MEMORY SPACE

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. The processor gives the starting address (address of the starting byte location) and the number of bytes needed when accessing a data value from the memory space.

The locations in a memory space are like phones in a telephone system where everyone has a unique phone number. Most modern computers have at least 4 GB-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. Since there is only one address for every location, we say that the memory space has a "flat" organization. So, all multidimensional arrays are ultimately "flattened" into equivalent 1D arrays. While a C programmer can use a multidimensional 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 an offset calculated from these multidimensional indices.

In reality, all multidimensional arrays in C are linearized. This is due to the use of a "flat" memory space in modern computers (see "Memory Space" sidebar). In the case of statically allocated arrays, the compilers allow the programmers to use higher-dimensional indexing syntax such as d_Pin[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 due to lack of dimensional information.

There are at least two ways one can linearize a 2D array. 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 *row-major layout*, is illustrated in Figure 4.3. To increase the readability, we will use $M_{j,i}$ to denote an M element at the *j* row and the *i* column. $M_{j,i}$ is equivalent to the C expression M[j][i] but slightly more readable. Figure 4.3 shows an example where 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, etc. Therefore, the 1D equivalent index for the M element in row *j* and column *i* is $j \times 4 + i$. The $j \times 4$ 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 $2 \times 4 + 1 = 9$. This is illustrated in Figure 4.3, where M_9 is the 1D equivalent to $M_{2,1}$. This is the way C compilers linearize 2D arrays.

FIGURE 4.3

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression `Row*Width + Col` for an element that is in the `Row`[th] row and `Col`[th] column of an array of `Width` elements in each row.

Another way to linearize a 2D array is to place all elements of the same column into consecutive locations. The columns are then placed one after another into the memory space. This arrangement, called *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 mentioning that readers whose primary previous programming experience was with FORTRAN should be aware that CUDA C uses row-major layout rather than column-major layout. Also, many C libraries that are designed to be used by FORTRAN programs use column-major layout to match the FORTRAN compiler layout. As a result, the manual pages for these libraries, such as Basic Linear Algebra Subprograms (see "Linear Algebra Functions" sidebar), 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 `pictureKernel()`, shown in Figure 4.4. Let's assume that the kernel will scale every pixel value in the picture by a factor of 2.0. The kernel code is conceptually quite simple. There are a total of `blockDim.x*gridDim.x` threads in the horizontal direction. As we learned in the `vecAddKernel()` example, the

```
__global__ void PictureKernell(float* d_Pin, float* d_Pout, int n, int m) {

   // Calculate the row # of the d_Pin and d_Pout element to process
   int Row = blockIdx.y*blockDim.y + threadIdx.y;

   // Calculate the column # of the d_Pin and d_Pout element to process
   int Col = blockIdx.x*blockDim.x + threadIdx.x;

   // each thread computes one element of d_Pout if in range
   if ((Row < m) && (Col < n)) {
     d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
   }

}
```

FIGURE 4.4

Source code of `pictureKernel()` showing a 2D thread mapping to a data pattern.
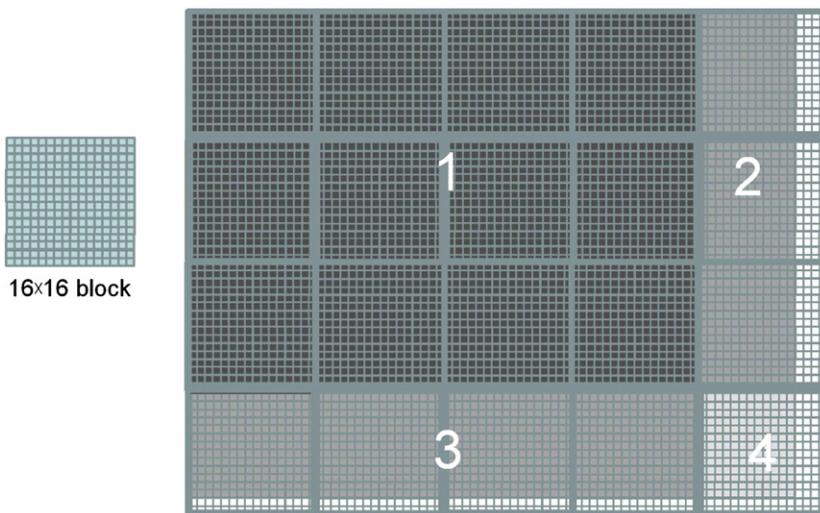
expression `Col = blockIdx.x*blockDim.x + threadIdx.x` generates every integer value from 0 to `blockDim.x*gridDim.x−1`. We know that `gridDim.x*blockDim.x` is greater than or equal to `n`. We have at least as many threads as the number of pixels in the horizontal direction. Similarly, we also know that there are at least as many threads as the number of pixels in the vertical direction. Therefore, as long as we test and make sure only the threads with both `Row` and `Col` values are within range, that is `(Col<n) && (Row<m)`, we will be able to cover every pixel in the picture. Since there are *n* pixels in a row, we can generate the 1D index for the pixel at row `Row` and column `Col` as `Row*n + Col`. This 1D index is used to read from the `d_Pin` array and write the `d_Pout` array.

Figure 4.5 illustrates the execution of `pictureKernel()` when processing our 76×62 example. Assuming that we use 16×16 blocks, launching `pictureKernel()` generates 80×64 threads. The grid will have 20 blocks, 5 in the horizontal direction and 4 in the vertical direction. During the execution, the execution behavior of blocks will fill into one of four different cases, shown as four major areas in Figure 4.5.

The first area, marked as 1 in Figure 4.5, 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 will 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 as 2 in Figure 4.5, contains the threads that belong to the 3 blocks in the medium-shaded area covering the upper-right

FIGURE 4.5

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

pixels of the picture. Although the Row values of these threads are always within range, the Col values of some of them exceed the n value (76). This is because the number of threads in the horizontal direction is always a multiple of the blockDim.x value chosen by the programmer (16 in this case). The smallest multiple of 16 needed to cover 76 pixels is 80. As a result, 12 threads in each row will find their Col values within range and will process pixels. On the other hand, 4 threads in each row will find their Col values out of range, and thus fail the if statement condition. These threads will not process any pixels. Overall, $12 \times 16 = 192$ out of the $16 \times 16 = 256$ threads will process pixels.

The third area, marked as 3 in Figure 4.5, accounts for the 3 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 m value (62). This is because the number of threads in the vertical direction is always multiples of the blockDim.y value chosen by the programmer (16 in this case). The smallest multiple of 16 to cover 62 is 64. As a result, 14 threads in each column will find their Row values within range and will process pixels. On the other hand, 2 threads in each column will fail the if statement of area 2, and will not process any pixels; $16 \times 14 = 224$ out of the 256 threads will process pixels.

The fourth area, marked as 4 in Figure 4.5, contains the threads that cover the lower-right light-shaded area of the picture. Similar to area 2, 4 threads in each of the top 14 rows will find their `Col` values out of range. Similar to area 3, the entire bottom two rows of this block will find their `Row` values out of range. So, only $14 \times 12 = 168$ of the $16 \times 16 = 256$ threads will be allowed to process threads.

We can easily extend our discussion of 2D arrays to 3D arrays by including another dimension when we linearize arrays. This is done by placing each "plane" of the array one after another. Assume that the programmer uses variables `m` and `n` to track the number of rows and columns in a 3D array. The programmer also needs to determine the values of `blockDim.z` and `gridDim.z` when launching 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 an array `P` will be in the form of `P[Plane*m*n + Row * n + Col]`. One would of course need to test if all the three global indices—`Plane`, `Row`, and `Col`—fall within the valid range of the array.

LINEAR ALGEBRA FUNCTIONS

Linear algebra operations are widely used in science and engineering applications. According to the widely used 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 amount of operations performed by the function increases. Level-1 functions perform vector operations of the form $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ 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 $\mathbf{y} = \alpha\mathbf{Ax} + \beta\mathbf{y}$, where $\mathbf{A}$ is a matrix, $\mathbf{x}$ and $\mathbf{y}$ are vectors, and α and β are scalars. We will be studying a form of level-2 function in the context of sparse linear algebra. Level-3 functions perform matrix—matrix operations in the form of $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ 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.

4.3 MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

Up to this point, we have studied `vecAddkernel()` and `pictureKernel()` where each thread performs only one floating-point arithmetic operation on

one array element. Readers should ask the obvious question: Do all CUDA threads perform only such a trivial amount of operation? The answer is no. Most real kernels have each thread to perform many more arithmetic operations and embody sophisticated control flows. These two simple kernels were selected for teaching the mapping of threads to data using threadIdx, blockIdx, blockDim, and gridDim variables. In particular, we introduce the following pattern of using global index values to ensure that every valid data element in a 2D array is covered by a unique thread:

    Row = blockIdx.x*blockDim.x + threadIdx.x
    and
    Col = blockIdx.y*blockDim.y + threadIdx.y
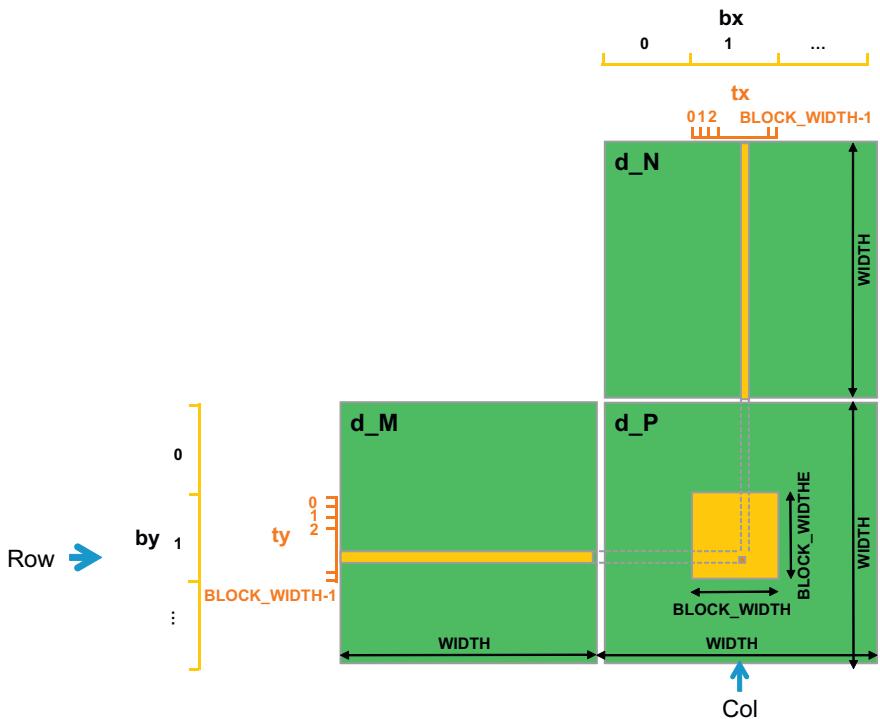
We also used vecAddKernel() and pictureKernel() to introduce the phenomenon that the number of threads that we create is a multiple of the block dimension. As a result, we will likely end up with more threads than data elements. Not all threads will process elements of an array. We use an if statement to test if the global index values of a thread are within the valid range. Now that we understand the mapping of threads to data, we are in a position to understand kernels that perform more complex computation.

Matrix−matrix multiplication between an $I \times J$ matrix d_M and a $J \times K$ matrix d_N produces an $I \times K$ matrix d_P. Matrix−matrix multiplication is an important component of the BLAS standard (see "Linear Algebra Functions" sidebar). For simplicity, we will limit our discussion to square matrices, where $I = J = K$. We will use variable Width for I, J, and K.

When performing a matrix−matrix multiplication, each element of the product matrix d_P is an inner product of a row of d_M and a column of d_N. We will continue to use the convention where $d\_P_{Row,Col}$ is the element at Row row and Col column. As shown in Figure 4.6, $d\_P_{Row, Col}$ (the small square in d_P) is the inner product of the Row row of d_M (shown as the horizontal strip in d_M) and the Col column of d_N (shown as the vertical strip in d_N). The inner product between two vectors is the sum of products of corresponding elements. That is, $d\_P_{Row,Col} = \sum d\_M_{Row,k} * d\_N_{k,Col}$, for k = 0, 1, ... Width-1. For example,

$$d\_P_{1,5} = d\_M_{1,0} * d\_N_{0,5} + d\_M_{1,1} * d\_N_{1,5} + d\_M_{1,2} * d\_N_{2,5} + ... + d\_M_{1,Width-1} * d\_N_{Width-1,5}$$

We map threads to d_P elements with the same approach as what we used for pictureKernel(). That is, each thread is responsible for calculating one d_P element. The d_P element calculated by a thread is in row blockIdx.y*blockDim.y + threadIdx.y and in column blockIdx.

FIGURE 4.6

Matrix multiplication using multiple blocks by tiling d_P.

x*blockDim.x + threadIdx.x. Figure 4.7 shows the source code of the kernel based on this thread-to-data mapping. Readers should immediately see the familiar pattern of calculating Row, Col, and the if statement testing if Row and Col are both within range. These statements are almost identical to their counterparts in pictureKernel(). The only significant difference is that we are assuming square matrices for the matrixMulKernel() so we replace both n and m with Width.

With our thread-to-data mapping, we effectively divide d_P into square tiles, one of which is shown as a large square in Figure 4.6. Some dimension sizes may be better for a device and others may be better for another device. This is why, in real applications, programmers often want to keep the block dimensions as an easily adjustable value in the host code. A common practice is to declare a compile-time constant and use this constant in the host statements for setting the kernel launch configuration.

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (intk = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

FIGURE 4.7

A simple matrix—matrix multiplication kernel using one thread to compute each
`d_P` element.

We will refer to this compile-time constant as `BLOCK_WIDTH`. To set
`BLOCK_WIDTH` to a value, say 16, we can use the following C statement in a
header file or the beginning of a file where `BLOCK_WIDTH` is used:

   `#define BLOCK_WIDTH 16`

Throughout the source code, instead of using a numerical value, the
programmer can use the name `BLOCK_WIDTH`. Using a named compile-time
constant allows the programmer to easily set `BLOCK_WIDTH` to a different
value when compiling for a particular hardware. It also allows an auto-
mated tuning system to search for the best `BLOCK_WIDTH` value by itera-
tively setting it to different values, compile, and run for the hardware of
interest. This type of process is often referred to as *autotuning*. In both
cases, the source code can remain largely unchanged while changing the
dimensions of the thread blocks.

Figure 4.8 shows the host code to be used to launch the
`matrixMulKernel()`. Note that the configuration parameter `dimGrid` is set
to ensure that for any combination of Width and `BLOCK_WIDTH` values,
there are enough thread blocks in both *x* and *y* dimensions to calculate all
`d_P` elements. Also, the name of the `BLOCK_WIDTH` constant rather than the
actual value is used in initializing the fields of `dimGrid` and `dimBlock`.
This allows the programmer to easily change the `BLOCK_WIDTH` value with-
out modifying any of the other statements. Assume that we have a Width
value of 1,000. That is, we need to do $1,000 \times 1,000$ matrix—matrix

multiplication. For a BLOCK_WIDTH value of 16, we will generate 16×16 blocks. There will be 64×64 blocks in the grid to cover all d_P elements. By changing the #define statement in Figure 4.8 to

 #define BLOCK_WIDTH 32

we will generate 32×32 blocks. There will be 32×32 blocks in the grid. We can make this change to the kernel launch configuration without changing any of the statements that initialize dimGrid and dimBlock.

We now turn our attention to the work done by each thread. Recall that $d\_P_{Row,\ Col}$ is the inner product of the Row row of d_M and the Col column of d_N. In Figure 4.7, we use a for loop to perform this inner product operation. Before we enter the loop, we initialize a local variable Pvalue to 0. Each iteration of the loop accesses an element in the Row row of d_M, an element in the Col column of d_N, multiplies the two elements together, and accumulates the product into Pvalue.

Let's first focus on accessing the Row row of d_M within the for loop. Recall that d_M is linearized into an equivalent 1D array where the rows of d_M are placed one after another in the memory space, starting with the 0 row. Therefore, the beginning element of the 1 row is d_M[1*Width] because we need to account for all elements of the 0 row. In general, the beginning element of the Row row is d_M[Row*Width]. Since all elements of a row are placed in consecutive locations, the k element of the Row row is at d_M[Row*Width + k]. This is what we used in Figure 4.7.

We now turn to accessing the Col column of d_N. As shown in Figure 4.3, the beginning element of the Col column is the Col element of the 0 row, which is d_N[Col]. Accessing each additional element in the Col column requires skipping over entire rows. This is because the next

```
#define BLOCK_WIDTH 16

// Setup the execution configuration
   int NumBlocks = Width/BLOCK_WIDTH;
   if (Width % BLOCK_WIDTH) NumBlocks++;
   dim3 dimGrid(NumBlocks, NumbBlocks);
   dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

// Launch the device computation threads!
   matrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

FIGURE 4.8

Host code for launching the matrixMulKernel() using a compile-time constant BLOCK_WIDTH to set up its configuration parameters.

element of the same column is actually the same element in the next row. Therefore, the k element of the Col column is d_N[k*Width + Col].

After the execution exits the for loop, all threads have their d_P element value in its Pvalue variable. It then uses the 1D equivalent index expression Row*Width + Col to write its d_P element. Again, this index pattern is similar to that used in the pictureKernel(), with n replaced by Width.

We now use a small example to illustrate the execution of the matrix−matrix multiplication kernel. Figure 4.9 shows a 4×4 d_P with BLOCK_WIDTH = 2. The small sizes allow us to fit the entire example in one picture. The d_P matrix is now divided into four tiles and each block calculates one tile. (Whenever it is clear that we are discussing a device memory array, we will drop the d_ part of the name to improve readability. In Figure 4.10, we use P instead of d_P since it is clear that we are discussing a device memory array.) 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 $P_{2,0}$. It is easy to verify that one can identify the P element calculated by thread(0,0) of block(1,0) with the formula:

$$P_{blockIdx.y*blockDim.y + threadIdx.y, blockIdx.x*BLOCK\_WIDTH + threadIdx.x} = P_{1*2+0,0*2+0}$$
$$= P_{2,0}$$

Readers should work through the index derivation for as many threads as it takes to become comfortable with the mapping.

Row and Col in the matrixMulKernel() identify the P element to be calculated by a thread. Row also identifies the row of M and Col identifies the column of N for input values for the thread. Figure 4.10 illustrates the
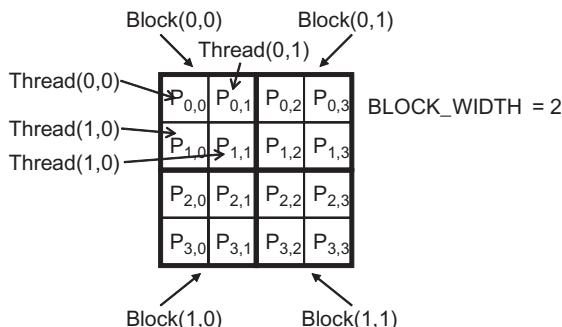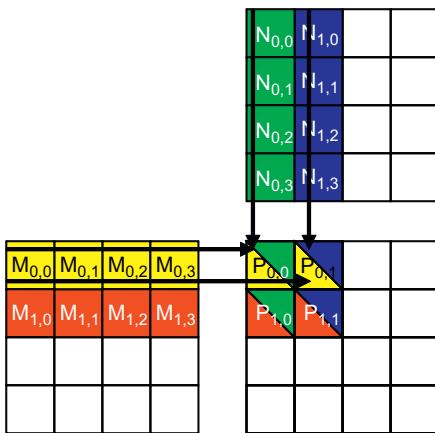
FIGURE 4.9

A small execution example of matrixMulKernel().

FIGURE 4.10

Matrix multiplication actions of one thread block. For readability, d_M, d_N, and d_P are shown as M, N, and P.

multiplication actions in each thread block. For the small matrix multiplication, threads in block(0,0) produce four dot products. The Row and Col variables of thread(0,0) in block(0,0) are $0*0 + 0 = 0$ and $0*0 + 0 = 0$. It maps to $P_{0,0}$ and calculates the dot product of row 0 of M and column 0 of N.

We now walk through the execution of the for loop of Figure 4.7 for thread(0,0) in block(0,0). During the 0 iteration ($k=0$), Row*Width + k = $0*4 + 0 = 0$ and k*Width + Col = $0*4 + 0 = 0$. Therefore, the we are accessing d_M[0] and d_N[0], which according to Figure 4.3 are the 1D equivalent of $d\_M_{0,0}$ and $d\_N_{0,0}$. Note that these are indeed the 0 elements of row 0 of d_M and column 0 of d_N.

During the first iteration ($k=1$), Row*Width + k = $0*4 + 1 = 1$ and k*Width + Col = $1*4 + 0 = 4$. We are accessing d_M[1] and d_N[4], which according to Figure 4.3 are the 1D equivalent of $d\_M_{0,1}$ and $d\_N_{1,0}$. These are the first elements of row 0 of d_M and column 0 of d_N.

During the second iteration ($k=2$), Row*Width + k = $0*4 + 2 = 2$ and k*Width + Col = 8, which results in d_M[2] and d_N[8]. Therefore, the elements accessed are the 1D equivalent of $d\_M_{0,2}$ and $d\_N_{2,0}$.

Finally, during the third iteration ($k=3$), Row*Width + k = $0*4 + 3$ and k*Width + Col = 12, which results in d_M[3] and d_N[12], the 1D equivalent of $dM_{0,3}$ and $d\_N_{3,0}$. We now have verified that the for loop performs the inner product between the 0 row of d_M and the 0 column of d_N. After the loop, the thread writes d_P[Row*Width + Col], which is d_P[0].

This is the 1D equivalent of $d\_P_{0,0}$ so thread(0,0) in block(0,0) successfully calculated the inner product between the 0 row of d_M and the 0 column of d_N and deposited the result in $d\_P_{0,0}$.

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.

Note that `matrixMulKernel()` can handle matrices of up to $16 \times 65{,}535$ elements in each dimension. In the situation where matrices larger than this limit are to be multiplied, one can divide up the P matrix into submatrices of which the size can be covered by a kernel. We can then either use the host code to iteratively launch kernels to complete the P matrix or have the kernel code of each thread to calculate more P elements.

4.4 SYNCHRONIZATION AND TRANSPARENT SCALABILITY
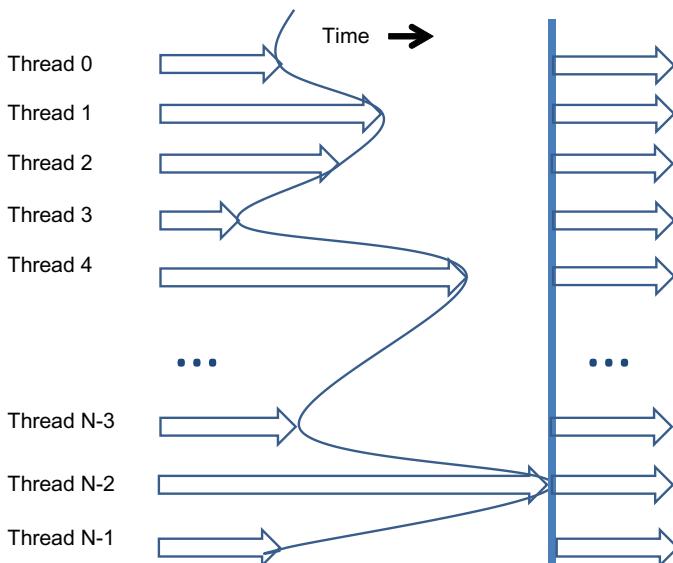
So far, we have discussed how to launch a kernel for execution by a grid of threads. We have also explained how one can map threads to parts of the data structure. However, we have not yet presented any means to coordinate the execution of multiple threads. We will now study a basic coordination mechanism. CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function `__syncthreads()`. Note that "__" actually consists of two "_" characters. When a kernel function calls `__syncthreads()`, all threads in a block will be held at the calling location until every thread in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase. We will discuss an important use case of `__syncthreads()` in Chapter 5.

Barrier synchronization is a simple and popular method of coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities of multiple persons. For example, assume that four friends go to a shopping mall in a car. They can all go to different stores to shop for their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit all the stores of interest. However, barrier synchronization is needed before they leave the mall. They have to wait until all four friends have returned to the car before they can leave—the ones who finish earlier need to wait for those who finish later. Without the barrier synchronization, one or more persons can be left in the mall when the car leaves, which can seriously damage their friendship!

Figure 4.11 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 of them much later. The ones that reach the barrier early will wait for those that arrive late. When the latest one arrives at the barrier, everyone can continue their execution. With barrier synchronization, "No one is left behind."

In CUDA, a __syncthreads() statement, if present, must be executed by all threads in a block. When a __syncthread() 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 __syncthreads() on the then path or all of them execute the else path. The two __syncthreads() are different barrier synchronization points. If a thread in a block executes the then path and another executes the else path, they would be waiting at different barrier synchronization points. They would end up waiting for each other forever. It is the responsibility of the programmers to write their code so that these requirements are satisfied.

The ability to synchronize also imposes execution constraints on threads within a block. These threads should execute in close time

FIGURE 4.11

An example execution timing of barrier synchronization.

proximity with each other to avoid excessively long waiting times. In fact, one 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 arrived at the barrier synchronization point can cause everyone else to wait forever. CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit. A block can begin execution only when the runtime system has secured all the resources needed for all threads in the block to complete execution. When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization.
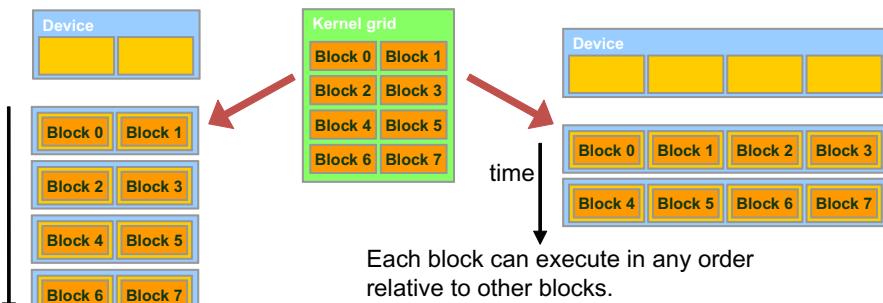
This leads us to a major trade-off in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other since none of them need to wait for each other. This flexibility enables scalable implementations as shown in Figure 4.12, where time 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; two blocks executing at a time is shown on the left side of Figure 4.12. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time; four blocks executing at a time is shown on the right side of Figure 4.12.

The ability to execute the same application code at a wide range of speeds allows the production of a wide range of implementations according to the cost, power, and performance requirements of particular 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 exactly the same application program with no change to the code. The ability to execute the same application code on hardware with a different number of execution resources is referred to as *transparent scalability*, which reduces the burden on application developers and improves the usability of applications.

4.5 ASSIGNING RESOURCES TO BLOCKS
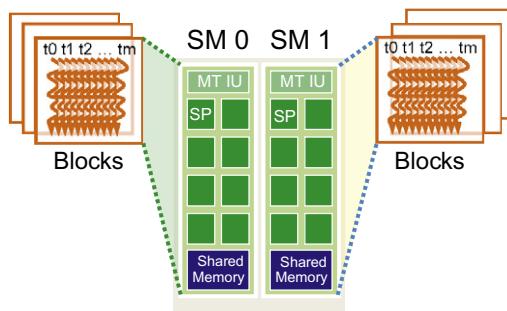
Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. As we discussed in the previous section, these

FIGURE 4.12

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

threads are assigned to execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into streaming multiprocessors (SMs). Figure 4.13 illustrates that multiple thread blocks can be assigned to each SM. Each device has a limit on the number of blocks that can be assigned to each SM. For example, a CUDA device may allow up to eight blocks to be assigned to each SM. In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of eight blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until their combined resource usage falls under the limit. With a limited numbers of SMs and a limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in a CUDA device. Most grids contain many more blocks than this number. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete executing the blocks previously assigned to them.

Figure 4.13 shows an example in which three thread blocks are assigned to each SM. One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled. It takes hardware resources for SMs to maintain the thread and block indices and track their execution status. In more recent CUDA device designs, up to 1,536 threads can be assigned to each SM. This could be in the form of 6 blocks of 256 threads each, 3 blocks of 512 threads each, etc. If the device only allows up to 8 blocks in an SM, it should be obvious that 12 blocks of 128

FIGURE 4.13

Thread block assignment to SMs.

threads each is not a viable option. If a CUDA device has 30 SMs and each SM can accommodate up to1,536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution.

4.6 QUERYING DEVICE PROPERTIES

Our discussions on assigning execution resources to blocks raise an important question: How do we find out the amount of resources available? When a CUDA application executes on a system, how can it find out the number of SMs in a device and the number of threads that can be assigned to each SM? Obviously, there are also other resources that we have not discussed so far but can be relevant to the execution of a CUDA application. 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 to take advantage of the more capable systems while compensating for the less capable systems.

RESOURCE AND CAPABILITY QUERIES

In everyday life, we often query the resources and capabilities. 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 have them.

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

> If the room comes with a microwave oven and a refrigerator, we can take the leftover from dinner and expect to eat it the second day. If the hotel has a pool, we can bring swim suits and take a dip after business meetings. If the hotel does not have a pool but has an exercise room, we can bring running shoes and exercise clothes. Some high-end Asian hotels even provide exercise clothing!
>
> These hotel amenities are part of the properties, or resources and capabilities, of the hotels. Veteran travelers check these properties at hotel web sites, choose the hotels that better match their needs, and pack more efficiently and effectively using the information.

In CUDA C, there is a built-in mechanism for host code to query the properties of the devices available in the system. The CUDA runtime system 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 using the following statements:

```
int dev_count;
cudaGetDeviceCount( &dev_count );
```

While it may not be obvious, a modern PC system can easily have 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 system numbers all the available devices in the system from `0` to `dev_count-1`. It provides an API function `cudaGetDeviceProperties()` that returns the properties of the device of which the 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 dev_prop;
for (I = 0; i < dev_count; i++) {
  cudaGetDeviceProperties( &dev_prop, i);
  // decide if device has sufficient resources and capabilities
}
```

The built-in type `cudaDeviceProp` is a C structure with fields that represent the properties of a CUDA device. Readers are referred to the *CUDA 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 `dev_prop` variable of which the fields are set by the `cudaGetDeviceProperties()` function. If readers choose 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 `dev_prop.maxThreadsPerBlock` gives the maximal number of threads allowed in a block in the queried device. Some devices allow up to 1,024 threads in each block and other devices allow fewer. It is possible that future devices may even allow more than 1,024 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 `dev_prop.multiProcessorCount`. As we discussed earlier, some devices have only a small number of SMs (e.g., 2) and some have a much larger number of SMs (e.g., 30). If the application requires a large number of SMs to achieve satisfactory performance, it should definitely check this property of the prospective device. Furthermore, the clock frequency of the device is in `dev_prop.clockRate`. The combination of the clock rate and the number of SMs gives a good indication of the hardware execution capacity of the device.

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

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

4.7 THREAD SCHEDULING AND LATENCY TOLERANCE

Thread scheduling is strictly an implementation concept and thus must be discussed in the context of specific hardware implementations. In most
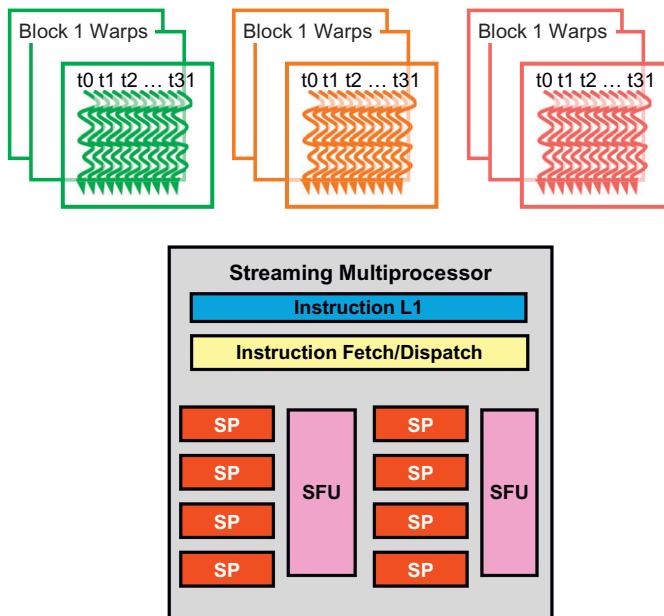
implementations to date, once a block is assigned to a SM, it is further divided into 32-thread units called *warps*. The size of warps is implementation-specific. In fact, warps are not part of the CUDA specification. However, knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices. The size of warps is a property of a CUDA device, which is in the `dev_prop.warpSize` field of the device query variable (`dev_prop` in this case).

The warp is the unit of thread scheduling in SMs. Figure 4.14 shows the division of blocks into warps in an implementation. Each warp consists of 32 threads of consecutive `threadIdx` values: threads 0−31 form the first warp, 32−63 the second warp, and so on. 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.

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. For example, in Figure 4.14, if each block has 256 threads, we can determine that each block has $256 \div 32$ or 8 warps. With three blocks in each SM, we have $8 \times 3 = 24$ warps in each SM.

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. This is illustrated in Figure 4.14 with a single instruction fetch/dispatch shared among execution units in the SM. Note that these threads will apply the same instruction to different portions of the data. As a result, all threads in a warp will always have the same execution timing.

Figure 4.14 also shows a number of hardware streaming processors (SPs) that actually execute instructions. In general, there are fewer SPs than the number of threads assigned to each SM. That is, each SM has only enough hardware to execute instructions from a small subset of all threads assigned to the SM at any point in time. In earlier GPU design, 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 for a small subset of all warps in the SM. A legitimate question is, why do we need to have so many warps in an SM if it can only execute a small subset of them at any instant? The answer is that this is how CUDA processors efficiently execute long-latency operations such as global memory accesses.

FIGURE 4.14

Blocks are partitioned into warps for thread scheduling.

When an instruction executed by the threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Another resident warp that is no longer waiting for results 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 with work from other threads is often called *latency tolerance* or *latency hiding* (see "Latency Tolerance" sidebar).

LATENCY TOLERANCE

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

When there is a long line in front of the service desk, it is important to maximize the productivity of the service clerks. Letting a person fill out the form in front of the clerk while everyone waits is not a good approach. The clerk should be helping the next customers who

are waiting in line while the person fills out the form. These other customers are "ready to go" and should not be blocked by the customer who needs more time to fill out a form.

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

We can think of these post office customers as warps and the clerk as a hardware execution unit. The customer who needs to fill out the form corresponds to a warp of which the 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 in spite of these long-latency operations. The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as *zero-overhead thread scheduling*. With warp scheduling, the long waiting time of warp instructions is "hidden" by executing instructions from other warps. This ability to tolerate long operation latencies is the main reason why GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as CPUs. As a result, GPUs can dedicate more of its chip area to floating-point execution resources.

We are now ready to do a simple exercise.[3] Assume that a CUDA device allows up to 8 blocks and 1,024 threads per SM, whichever becomes a limitation first. Furthermore, it allows up to 512 threads in each block. For matrix−matrix multiplication, should we use 8×8, 16×16, or 32×32 thread blocks? To answer the question, we can analyze the pros and cons of each choice. If we use 8×8 blocks, each block would have only 64 threads. We will need $1,024 \div 64 = 12$ blocks to fully occupy an SM. However, since there is a limitation of up to 8 blocks in each SM, we will end up with only $64 \times 8 = 512$ threads in each SM. This means that the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long-latency operations.

The 16×16 blocks give 256 threads per block. This means that each SM can take $1,024 \div 256 = 4$ blocks. This is within the 8-block limitation.

---

[3]Note that this is an oversimplified exercise. As we will explain in Chapter 5, the usage of other resources, such as registers and shared memory, must also be considered when determining the most appropriate block dimensions. This exercise highlights the interactions between the limit on the number of thread blocks and the limit on the number of threads that can be assigned to each SM.

This is a good configuration since we will have full thread capacity in each SM and a maximal number of warps for scheduling around the long-latency operations. The 32×32 blocks would give 1,024 threads in each block, exceeding the limit of 512 threads per block for this device.

4.8 SUMMARY

The kernel execution configuration defines the dimensions of a grid and its blocks. Unique coordinates in `blockIdx` and `threadIdx` variables 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. This model of programming compels the programmer to organize threads and their data into hierarchical and multidimensional organizations.

Once a grid is launched, its blocks are assigned to SMs in arbitrary order, resulting in transparent scalability of CUDA applications. The transparent scalability comes with a limitation: threads in different blocks cannot synchronize with each other. To allow a kernel to maintain transparent scalability, the simple way for threads in different blocks to synchronize with each other is to terminate the kernel and start a new kernel for the activities after the synchronization point.

Threads are assigned to SMs for execution on a block-by-block basis. 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 thread blocks and the number of threads each of its SMs can accommodate, whichever becomes a limitation first. For each kernel, one or more of these resource limitations can become the limiting factor for the number of threads that simultaneously reside in a CUDA device.

Once a block is assigned to an SM, it is further partitioned into warps. All threads in a warp have identical execution timing. 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 execution units.

4.9 EXERCISES

4.1 If a CUDA device's SM can take up to 1,536 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. 1,024 threads per block

4.2 For a vector addition, assume that the vector length is 2,000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?

    a. 2,000

    b. 2,024

    c. 2,048

    d. 2,096

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

    a. 1

    b. 2

    c. 3

    d. 6

4.4 You need to write a kernel that operates on an image of size 400×900 pixels. You would like to assign one thread to each pixel. You would like your thread blocks to be square and to use the maximum number of threads per block possible on the device (your device has compute capability 3.0). How would you select the grid dimensions and block dimensions of your kernel?

4.5 For the previous question, how many idle threads do you expect to have?

4.6 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, 2.9, and spend the rest of their time waiting for the barrier. What percentage of the threads' summed-up execution times is spent waiting for the barrier?

4.7 Indicate which of the following assignments per multiprocessor is possible. In the case where it is not possible, indicate the limiting factor(s).

    a. 8 blocks with 128 threads each on a device with compute capability 1.0

    b. 8 blocks with 128 threads each on a device with compute capability 1.2

    c. 8 blocks with 128 threads each on a device with compute capability 3.0

    d. 16 blocks with 64 threads each on a device with compute capability 1.0

    e. 16 blocks with 64 threads each on a device with compute capability 1.2

    f. 16 blocks with 64 threads each on a device with compute capability 3.0

4.8 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.

4.9 A student mentioned that he was able to multiply two $1,024 \times 1,024$ matrices using a tiled matrix multiplication code with 32×32 thread blocks. He is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

4.10 The following kernel is executed on a large matrix, which is tiled into submatrices. To manipulate tiles, a new CUDA programmer has written the following device kernel, which will transpose each tile in the 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, but could be set anywhere from 1 to 20.

```
dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);
```

```
BlockTranspose<< <gridDim, blockDim>> >(A, A_width,
A_height);
  __global__ void
  BlockTranspose(float* A_elements, int A_width, int
A_height)
  {
    __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];
    int baseIdx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    baseIdx + = (blockIdx.y * BLOCK_SIZE + threadIdx.y) *
A_width;
    blockA[threadIdx.y][threadIdx.x] = A_elements
[baseIdx];
    A_elements[baseIdx] = blockA[threadIdx.x][threadIdx.y];
  }
```

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

b. If the code does not execute correctly for all BLOCK_SIZE values, suggest a fix to the code to make it work for all BLOCK_SIZE values.

CUDA Memories

5

So far, we have learned to write a CUDA kernel function that is executed by a massive number of threads. The data to be processed by these threads is first transferred from the host memory to the device global memory. The threads then access their portion of the data from the global memory using their block IDs and thread IDs. We have also learned more details of the assignment and scheduling of threads for execution. Although this is a very good start, these simple CUDA kernels will likely achieve only a small fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that global memory, which is typically implemented with dynamic random access memory (DRAM), tends to have long access latencies (hundreds of clock cycles) and finite access bandwidth. While having many threads available for execution can theoretically tolerate long memory access latencies, one can easily run into a situation where traffic congestion in the global memory access paths prevents all but very few threads from making progress, thus rendering some of the streaming multiprocessors (SMs) idle. To circumvent such congestion, CUDA provides a number of additional methods for accessing memory that can remove the majority of data requests to the global memory. In this chapter, you will learn to use these memories to boost the execution efficiency of CUDA kernels.

5.1 **IMPORTANCE OF MEMORY ACCESS EFFICIENCY**

We can illustrate the effect of memory access efficiency by calculating the expected performance level of the matrix multiplication kernel code in Figure 4.7, replicated in Figure 5.1. The most important part of the kernel in terms of execution time is the `for` loop that performs inner product calculation.

```
for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width + k] * d_N[k*Width + Col];
```

In every iteration of this loop, two global memory accesses are per-formed for one floating-point multiplication and one floating-point addi-tion. One global memory access fetches a `d_M[]` element and the other fetches a `d_N[]` element. One floating-point operation multiplies the `d_M[]` and `d_N[]` elements fetched and the other accumulates the product into `Pvalue`. Thus, the ratio of floating-point calculation to global memory access operation is 1:1, or 1.0. We will refer to this ratio as the *compute to global memory access (CGMA) ratio*, defined as the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

CGMA has major implications on the performance of a CUDA kernel. In a high-end device today, the global memory bandwidth is around 200 GB/s. With 4 bytes in each single-precision floating-point value, one can expect to load no more than 50 (200/4) giga single-precision operands per second. With a CGMA ration of 1.0, the matrix multiplication kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, intWidth) {

  // Calculate the row index of the d_P element and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

FIGURE 5.1

A simple matrix—matrix multiplication kernel using one thread to compute each `d_P` element (copied from Figure 4.7).

will execute no more than 50 giga floating-point operations per second (GFLOPS). While 50 GFLOPS is a respectable number, it is only a tiny fraction of the peak single-precision performance of 1,500 GFLOPS or higher for these high-end devices. We need to increase the CGMA ratio to achieve a higher level of performance for the kernel. For the matrix multiplication code to achieve the peak 1,500 GFLOPS rating of the processor, we need a CGMA value of 30. The desired CGMA ratio has roughly doubled in the past three generations of devices.

THE VON NEUMANN MODEL

In his seminal 1945 report, John von Neumann described a model for building electronic computers that 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 here. The computer has an I/O 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 used 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 actions of a computer by storing a different program into its memory.

5.2 CUDA DEVICE MEMORY TYPES

CUDA supports several types of memory that can be used by programmers to achieve a high CGMA ratio and thus a high execution speed in their kernels. Figure 5.2 shows these CUDA device memories. At the bottom of the figure, we see global memory and constant memory. These types of memory can be written (W) and read (R) by the host by calling API functions.[1] We have already introduced global memory in Chapter 3. The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.

Registers and shared memory in Figure 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 only access its own registers. A kernel function typically uses registers to hold frequently accessed variables that are

---
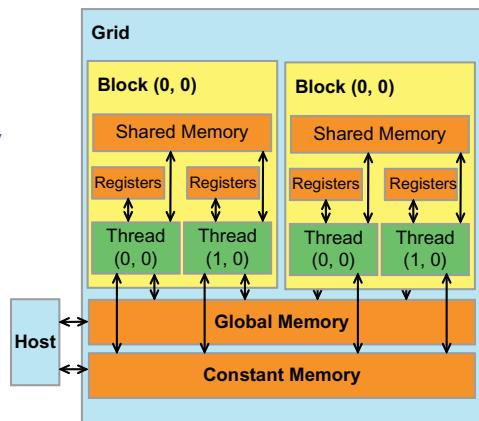
[1]See the *CUDA Programming Guide* for zero-copy access to the global memory.

Device code can:

- – R/W per-thread registers
- – R/W per-thread local memory
- – R/W per-block shared memory
- – R/W per-grid global memory
- – Read only per-grid constant memory

Host code can

- – Transfer data to/from per grid global and constant memories

FIGURE 5.2

Overview of the CUDA device memory model.

private to each thread. Shared memory is allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block. Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work. By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.
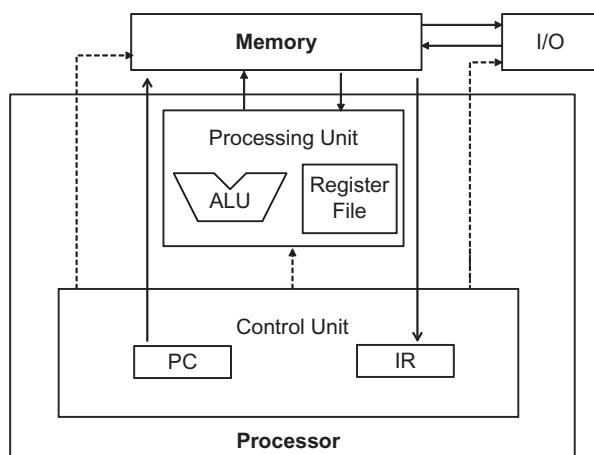
To fully appreciate the difference between registers, shared memory, and global memory, we need to go into a little more detail of how these different types of memories are realized and used in modern processors. The global memory in the CUDA programming model maps to the memory of the von Neumann model (see "The von Neumann Model" sidebar). The processor box in Figure 5.3 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. It is on the processor chip, which implies very short access latency and drastically higher access bandwidth. In a typical device, the aggregated access bandwidth of the register files is about two orders of magnitude of 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 increase in the CGMA ratio.

A more subtle point is that each access to registers involves fewer instructions than global memory. In Figure 5.3, the processor uses the PC value to fetch instructions from memory into the IR (see "The von Neumann Model" sidebar). The bits of the fetched instructions are then used to control the activities of the components of the computer. Using the instruction bits to control the activities of the computer is referred to as *instruction execution*. The number of instructions that can be fetched and executed in each clock cycle is limited. Therefore, the more instructions that need to be executed for a program, the more time it can take to execute the program.

Arithmetic instructions in most modern processors have "built-in" register operands. For example, a typical floating addition instruction is of the 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, there is no additional instruction required to make the operand value available to the arithmetic and logic unit (ALU) where the arithmetic calculation is done.

FIGURE 5.3

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

On the other hand, if an operand value is in global memory, one 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 global memory of a typical computer today, the instructions involved will likely be

```
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`. The `fadd` instruction then performs the floating addition using the values in `r2` and `r3` and places the result into `r1`. Since the processor can only fetch and execute a limited number of instructions per clock cycle, the version with an additional load will likely take more time to process than the one without. This is another reason why placing the operands in registers can improve execution speed.

PROCESSING UNITS AND THREADS
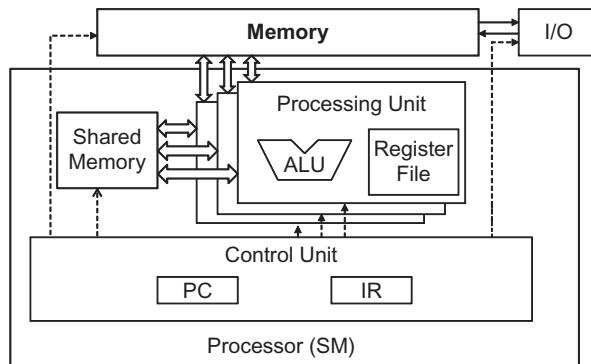
Now that we have introduced the von Neumann model, we are ready to discuss how threads are implemented. A thread in modern computers is a virtualized von Neumann processor. Recall that a thread consists of the code of a program, the particular point in the code that is being executed, and the value of 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 particular point of the program that is being executed. The IR holds the instruction that is fetched from the point execution. 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 timeshare 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.

Some processors provide multiple processing units, which allow multiple threads to make simultaneous progress. Figure 5.4 shows a single instruction, multiple data (SIMD) design style where all processing units share a PC and IR. Under this design, all threads making simultaneous progress execute the same instruction in the program.

Finally, there is another subtle reason why placing an operand value in registers is preferable. In modern computers, the energy 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. We will look at more details of the speed and energy difference in accessing these two hardware structures in modern computers soon. However, as we will soon learn, the number of registers available to each thread is quite limited in today's GPUs. We need to be careful not to oversubscribe to this limited resource.

FIGURE 5.4

Shared memory versus registers in a CUDA device SM.

Figure 5.4 shows 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 (see Section 4.2). When the processor accesses data that resides in the shared memory, it needs to perform a memory load operation, just like accessing data in the global memory. However, because shared memory resides on-chip, it can be accessed with much lower latency and much higher bandwidth than the global memory. Because of the need to perform a load operation, share memory has longer latency and lower bandwidth than registers. In computer architecture, share memory is a form of *scratchpad memory*.

One important difference between the share memory and registers in CUDA is that variables that reside in the shared memory are accessible by all threads in a block. This is in contrast to 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 Figure 5.4, a CUDA device SM typically employs multiple processing units, referred to as SPs in Figure 4.14, to allow multiple threads to make simultaneous progress (see "Processing Units and Threads" sidebar). Threads in a block can be spread across these processing units. Therefore, the hardware implementations of 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, 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 types of device memory. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of all grids. If a variable's scope is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable. For example, if a kernel declares a variable of which the 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 kernel's execution or throughout the entire application. If a variable's lifetime is within a kernel'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 to use them. 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.

As shown in Table 5.1, all automatic scalar variables declared in kernel and device functions are placed into registers. We refer to variables that are not arrays as *scalar* variables. 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

Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

that executes the kernel function. When a thread terminates, all its automatic variables also cease to exist. In Figure 5.1, variables `Row`, `Col`, and `Pvalue` 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. We will address this point in Chapter 6.

Automatic array variables are not stored in registers.[2] Instead, they are stored into the global memory and may incur long access delays and potential access congestions. The scope of these arrays is, like automatic scalar variables, limited to individual threads. That is, a private version of each automatic array is created for and used by every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist. 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 keyword `__shared__` (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 declaration typically resides within a kernel function or a device function. Shared variables reside in 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 thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel. When a kernel terminates its 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 are heavily used in an execution phase of a kernel. One may need to adjust the algorithms used to create execution phases that heavily focus on small portions of the global memory data, as we will demonstrate with matrix multiplication in Section 5.3.

If a variable declaration is preceded by the 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

---

[2]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.

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. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at 65,536 bytes. One may need to break up the input data volume to fit within this limitation, as we will illustrate in Chapter 8.

A variable of which the 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. One must, however, be aware of the fact that there is currently no easy way to synchronize between threads from different thread blocks or to ensure data consistency across threads when accessing global memory other than terminating the current kernel execution.[3] Therefore, global variables are often used to pass information from one kernel invocation to another kernel invocation.

In CUDA, pointers are used to point to data objects in global memory. There are two typical ways in which pointer usage arises in kernel and device functions. First, if an object is allocated by a host function, the pointer to the object is initialized by cudaMalloc() and can be passed to the kernel function as a parameter. For example, the parameters d_M, d_N, and d_P in Figure 5.1 are such pointers. The second type of usage is to assign the address of a variable 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. Readers should refer to the *CUDA Programming Guide* for using pointers in other memory types.

---
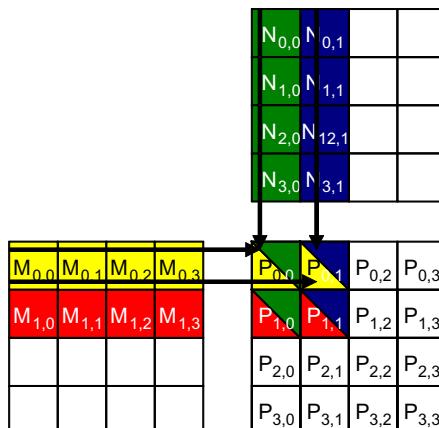
[3]Note that one can use CUDA memory fencing to ensure data coherence between thread blocks if the number of thread blocks is smaller than the number of SMs in the CUDA device. See the *CUDA Programming Guide* for more details.

5.3 A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

We have an intrinsic trade-off in the use of device memories in CUDA: global memory is large but slow, whereas the shared memory is small but fast. A common strategy is 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 tiles (i.e., subsets that each can fit into the shared memory). An important criterion is that the kernel computation on these tiles can be done independently of each other. Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example. Figure 5.5 shows a small example of matrix multiplication. It corresponds to the kernel function in Figure 5.1. For brevity, we abbreviate `d_P[y*Width + x]`, `d_M[y*Width + x]`, and `d_N[y*Width + x]` into $P_{y,x}$, $M_{y,x}$, and $N_{y,x}$, respectively. This example assumes that we use four 2×2 blocks to compute the P matrix. Figure 5.5 highlights the computation done by the four threads of block(0,0). These four threads compute $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$. The accesses to the M and N elements by thread(0,0) and thread(0,1) of block(0,0) are highlighted with black arrows. For example, thread(0,0) reads $M_{0,0}$ and $N_{0,0}$, followed by $M_{0,1}$, and $N_{1,0}$ followed by $M_{0,2}$ and $N_{2,0}$, followed by $M_{0,3}$ and $N_{3,0}$.

FIGURE 5.5

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

Figure 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 to the 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 terms of the M and N elements they access. For example, thread$_{0,0}$ and thread$_{0,1}$ both access M$_{0,0}$ as well as the rest of row 0 of M. Similarly, thread$_{0,1}$ and thread$_{1,1}$ both access N$_{0,1}$ as well as the rest of column 1 of N.

The kernel in Figure 5.1 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$_{1,0}$ to collaborate so that these M elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In general, we can see that every M and N element is accessed exactly twice during the execution of block$_{0,0}$. Therefore, if we can have all four threads to collaborate in their accesses to global memory, we can reduce the traffic to the global memory by half.

Readers should verify that the potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used. With $N \times N$ blocks, the potential reduction of global memory traffic would be N. That is, if we use 16×16 blocks, one can potentially reduce the global memory traffic to 1/16 through collaboration between threads.

Traffic congestion obviously does not only arise in computing. Most of us have experienced traffic congestion in highway systems, as illustrated in Figure 5.7. The root cause of highway traffic congestion is that there are too many cars all squeezing through a road that is designed for a much smaller number of vehicles. When congestion occurs, the travel time for

Access order

thread$_{0,0}$	M$_{0,0}$ * N$_{0,0}$	M$_{0,1}$ * N$_{1,0}$	M$_{0,2}$ * N$_{2,0}$	M$_{0,3}$ * N$_{3,0}$
thread$_{0,1}$	M$_{0,0}$ * N$_{0,1}$	M$_{0,1}$ * N$_{1,1}$	M$_{0,2}$ * N$_{2,1}$	M$_{0,3}$ * N$_{3,1}$
thread$_{1,0}$	M$_{1,0}$ * N$_{0,0}$	M$_{1,1}$ * N$_{1,0}$	M$_{1,2}$ * N$_{2,0}$	M$_{1,3}$ * N$_{3,0}$
thread$_{1,1}$	M$_{1,0}$ * N$_{0,1}$	M$_{1,1}$ * N$_{1,1}$	M$_{1,2}$ * N$_{2,1}$	M$_{1,3}$ * N$_{3,1}$
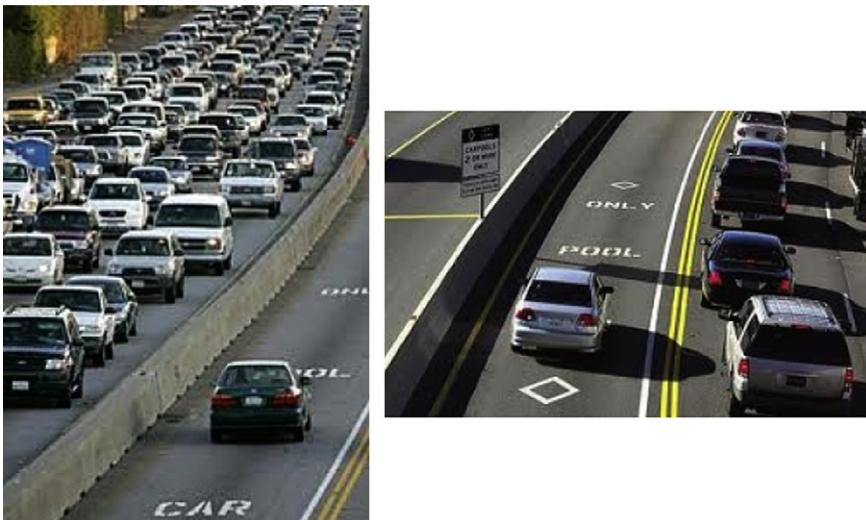
FIGURE 5.6

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

each vehicle is greatly increased. Commute time to work can easily double or triple during traffic congestion.

All proposed solutions for reduced traffic congestion involve reduction of cars on the road. Assuming that the number of commuters is constant, people need to share rides to reduce the number of cars on the road. A common way to share rides in the United States is carpools, where a group of commuters take turns to drive the group to work in one vehicle. In some countries, the government simply disallows certain classes of cars to be on the road on a daily basis. For example, cars with odd license plates may not be allowed on the road on Monday, Wednesday, or Friday. This encourages people whose cars are allowed on different days to form a car-pool group. There are also countries where the government makes gasoline so expensive that people form carpools to save money. In other countries, the government may provide incentives for behavior that reduces the num-ber of cars on the road. In the United States, 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. All these measures for encouraging carpooling are designed to overcome the fact that carpooling requires extra effort, as we show in Figure 5.8.

The top half of Figure 5.8 shows a good schedule pattern for carpool-ing. Time goes from left to right. Worker A and worker B have similar
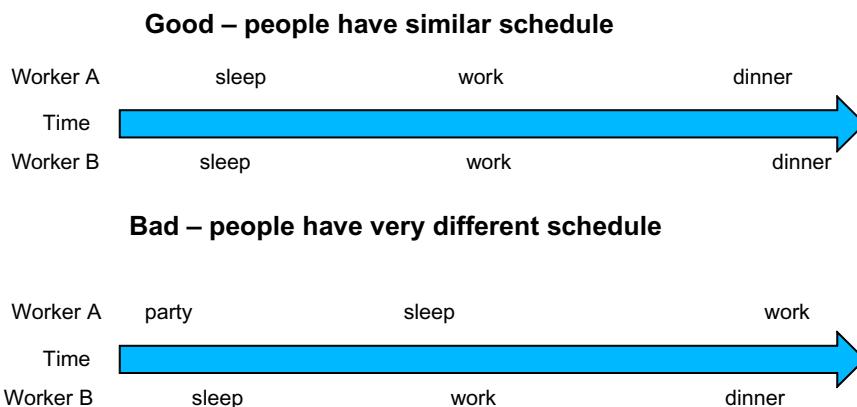
FIGURE 5.7

Reducing traffic congestion in highway systems.

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, however, not the case of the schedules shown in the bottom half of Figure 5.8. Worker A and worker B have very different habits 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 p.m. The schedules are so wildly different that there is no way these two workers can coordinate a common time to drive to work and return home in one car. For these workers to form a carpool, they need to negotiate a common schedule similar to what is shown in the top half of Figure 5.8.

Tiled algorithms are very similar to carpooling arrangements. We can think of data values accessed by each thread as commuters and DRAM requested as vehicles. When the rate of DRAM requests exceeds the provisioned bandwidth of the DRAM system, traffic congestion arises and the arithmetic units become idle. If multiple threads access data from the same DRAM location, they can form a "carpool" and combine their accesses into one DRAM request. This, however, requires the threads to have a similar execution schedule so that their data accesses can be combined into one. This is shown in Figure 5.9, where the top portion shows two threads that access the same data elements with similar timing. The bottom half shows two threads that access their common data in very different times. The reason why the bottom half is a bad arrangement is that

FIGURE 5.8

Carpooling requires synchronization among people.

data elements brought back from the DRAM need to be kept in the on-chip memory for a long time, waiting for thread 2 to consume them. This will likely require a large number of data elements to be kept around, thus large on-chip memory requirements. As we will show in the next section, we will use barrier synchronization to keep the threads that form the "car-pool" group to follow approximately the same execution timing.

5.4 A TILED MATRIX—MATRIX MULTIPLICATION KERNEL

We now present an algorithm where threads collaborate to reduce the traffic to the global memory. The basic idea is to have the threads to collaboratively load 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 Figure 5.10.

In Figure 5.10, we divide the M and N matrices into 2×2 tiles, as delineated by the thick lines. The dot product calculations performed by
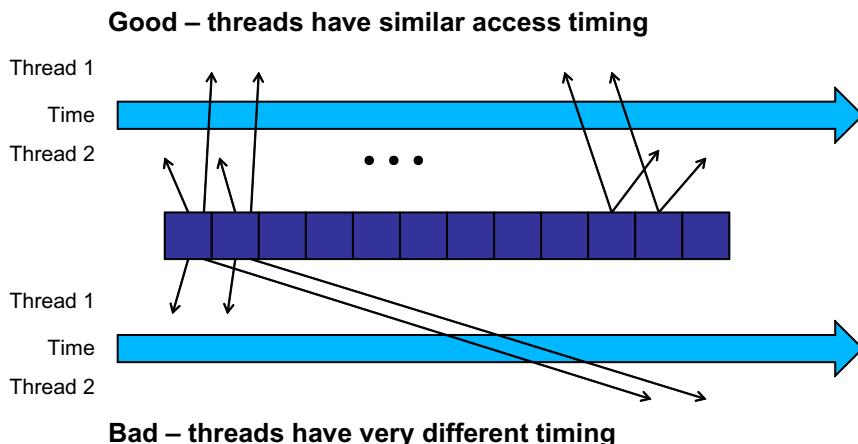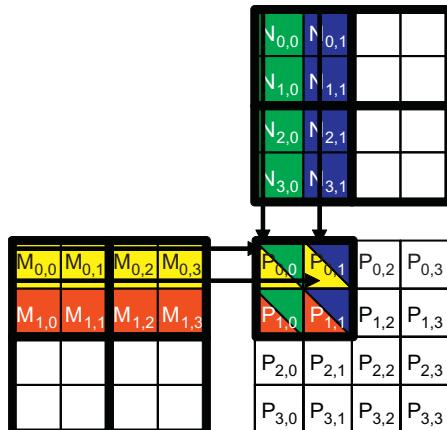
FIGURE 5.9

Tiled algorithms require synchronization among threads.

each thread are now divided into phases. In each phase, all threads in a block collaborate to load a tile of M elements and a tile of N elements into the shared memory. This is done by having every thread in a block to load one M element and one N element into the shared memory, as illustrated in Figure 5.11. Each row of Figure 5.11 shows the execution

FIGURE 5.10

Tiling M and N matrices to utilize shared memory.

FIGURE 5.11

Execution phases of a tiled matrix multiplication.

activities of a thread. Note that time progresses from left to right. We only need to show the activities of threads in block$_{0,0}$; the other blocks all have the same behavior. The shared memory array for the M elements is called Mds. The shared memory array for the N elements is called Nds. At the beginning of phase 1, the four threads of block$_{0,0}$ collaboratively load a tile of M elements 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 $Mds_{1,1}$. See the second column of Figure 5.11. A tile of N elements is also loaded in a similar manner, shown in the third column of Figure 5.11.

After the two tiles of M and N elements are loaded into the shared memory, these values 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$_{0,1}$ 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 half. Readers 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 in Figure 5.6 is now performed in two phases, shown as phase 1 and phase 2 in Figure 5.11. In each phase, products of two pairs of the input matrix elements are accumulated into the Pvalue variable. Note 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 Figure 5.11; the second phase in the seventh column. In general, if an input matrix is of dimension N and the tile size is TILE_WIDTH, the dot product would be performed in N/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 to hold the input values. In each phase, the same Mds and Nds are used 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 due to the fact that 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 return to the concept of locality in Chapter 6.

We are now ready to present the tiled kernel function that uses shared memory to reduce the traffic to global memory. The kernel shown in Figure 5.12 implements the phases illustrated in Figure 5.11. In Figure 5.12, lines 1 and 2 declare Mds and Nds as shared memory variables. Recall that the scope of shared memory variables is a block. Thus, one pair of Mds and Nds will be created for each block and all threads of a block have access to the same Mds and Nds. This is important since all threads in a block must have access to the M and N values loaded into Mds and Nds by their peers so that they can use these values to satisfy their input needs.

#define TILE_WIDTH 16

Lines 3 and 4 save the threadIdx and blockIdx values into automatic variables and thus into registers for fast access. Recall that automatic

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    int Width) {

1.    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.    int bx = blockIdx.x;  int by = blockIdx.y;
4.    int tx = threadIdx.x; int ty = threadIdx.y;

      // Identify the row and column of the d_P element to work on
5.    int Row = by * TILE_WIDTH + ty;
6.    int Col = bx * TILE_WIDTH + tx;

7.    float Pvalue = 0;
      // Loop over the d_M and d_N tiles required to compute d_P element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

        // Coolaborative loading of d_M and d_N tiles into shared memory
9.      Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
10.     Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

12.     for (int k = 0; k < TILE_WIDTH; ++k) {
13.       Pvalue += Mds[ty][k] * Nds[k][tx];
        }
14.     __syncthreads();
      }
15.   d_P[Row*Width + Col] = Pvalue;
    }
```

FIGURE 5.12
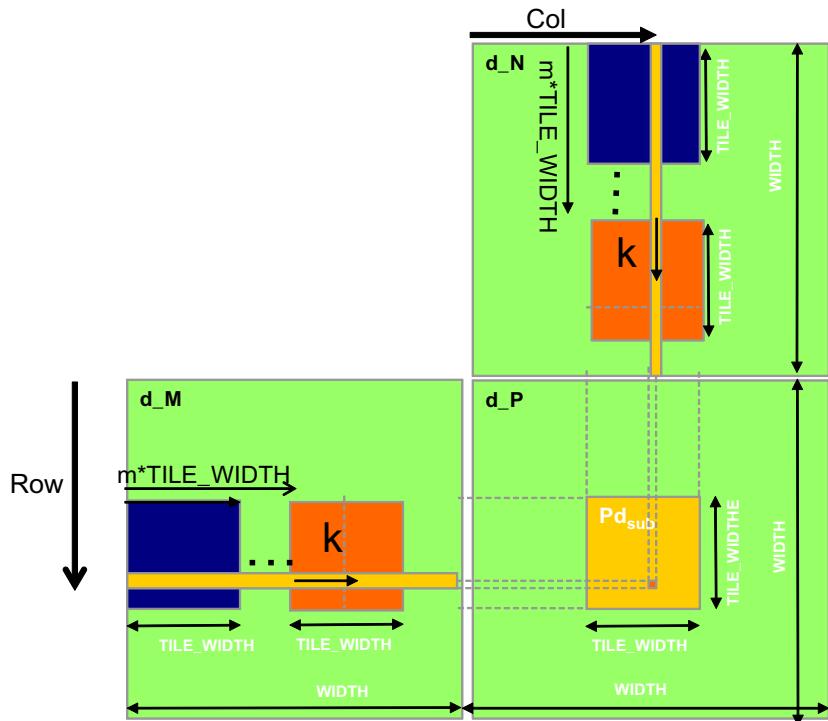
Tiled matrix multiplication kernel using shared memory.

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. They will reside in registers that are accessible by one thread. They are initialized with the threadIdx and blockIdx values and used many times during the lifetime of the thread. Once the thread ends, the values of these variables also cease to exist.

Lines 5 and 6 determine the row index and column index of the d_P element that the thread is to produce. As shown in line 6, the horizontal (*x*) position, or the column index of the d_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 in the horizontal dimension. A thread in block bx would have bx blocks of threads, or (bx*TILE_WIDTH) threads, before it; they cover bx*TILE_WIDTH elements of d_P. Another tx thread within the same block would cover another tx element of d_P. Thus, the thread with bx and tx should be responsible for calculating the d_P element of which the xindex is bx*TILE_WIDTH+tx. This horizontal index is saved in the variable Col (for column) for the thread and is also illustrated in Figure 5.13. For the example in Figure 5.10, the x index of the d_P element to be calculated by thread$_{0,1}$ of block$_{1,0}$ is $0 \times 2 + 1 = 1$. Similarly, the y index can be calculated as by*TILE_WIDTH + ty. This vertical index is saved in the variable Row for the thread. Thus, as shown in Figure 5.10, each thread calculates the d_P element at the Col column and the Row row. Going back to the example in Figure 5.10, the y index of the d_P element to be calculated by thread$_{1,0}$ of block$_{0,1}$ is $1 \times 2 + 0 = 2$. Thus, the d_P element to be calculated by this thread is d_P$_{2,1}$.

Line 8 of Figure 5.12 marks the beginning of the loop that iterates through all the phases of calculating the final d_P element. Each iteration of the loop corresponds to one phase of the calculation shown in Figure 5.11. The m variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of d_M and one tile of d_N elements. Therefore, at the beginning of each phase, m*TILE_WIDTH pairs of d_M and d_N elements have been processed by previous phases.

In each phase, line 9 loads the appropriate d_M element into the shared memory. Since we already know the row of d_M and column of d_N to be processed by the thread, we will focus on the column index of d_M and row index of d_N. As shown in Figure 5.11, each block has TILE_WIDTH^2 threads that will collaborate to load TILE_WIDTH^2 d_M elements into the shared memory. Thus, all we need to do is to assign each thread to load one d_M element. This is conveniently done using the blockIdx and

FIGURE 5.13

Calculation of the matrix indices in tiled multiplication.

`threadIdx`. Note that the beginning column index of the section of `d_M` elements to be loaded is `m*TILE_WIDTH`. Therefore, an easy approach is to have every thread load an element from at an offset `tx` that contains `threadIdx.x` value. This is precisely what we have in line 9, where each thread loads `d_M[Row*Width + m*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 `d_M` element into the shared memory. Together, these threads will load the dark square subset of `d_M` in Figure 5.13. Readers should use the small example in Figures 5.5 and 5.6 to verify that the address calculation works correctly.

The barrier `__syncthreads()` in line 11 ensures that all threads have finished loading the tiles of `d_M` and `d_N` into `Mds` and `Nds` before any of them can move forward. The loop in line 12 then performs one phase of the dot product based on these tile elements. The progression of the loop for `thread(ty,tx)` is shown in Figure 5.13, with the direction of `d_M` and

d_N elements usage along the arrow marked with k, the loop variable in line 12. Note that these elements will be accessed from Mds and Nds, the shared memory arrays holding these d_M and d_N elements. The barrier __syncthreads() in line 14 ensures that all threads have finished using the d_M and d_N elements in the shared memory before any of them move on to the next iteration and load the elements in the next tiles. This way, none of the threads would load the elements too early and corrupt the input values for other threads.

After all sections of the dot product are complete, the execution exits the loop of line 8. All threads write to their d_P element using the 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. If one uses 16×16 tiles, we can reduce the global memory accesses by a factor of 16. This increases the CGMA from 1 to 16. This improvement allows the memory bandwidth of a CUDA device to support a computation rate close to its peak performance. For example, this improvement allows a 150 GB/s global memory bandwidth to support $(150/4) \times 16 = 600$ GFLOPS!

5.5 MEMORY AS A LIMITING FACTOR TO PARALLELISM

While CUDA registers and shared memory can be extremely effective in reducing the number of accesses to global memory, one must be careful not to exceed the capacity of these memories. These memories are forms of resources that are needed for thread execution. Each CUDA device offers a limited amount of 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 can reside in each SM, and thus the fewer number of threads that can reside in the entire device.

Let's use an example to illustrate the interaction between register usage of a kernel and the level of parallelism that a device can support. Assume that in a device D, each SM can accommodate up to 1,536 threads and has 16,384 registers. While 16,384 is a large number, it only allows each thread to use a very limited number of registers considering the number of threads that can reside in each SM. To support 1,536 threads, each thread can use only $16,384 \div 1,536 = 10$ registers. If each thread uses 11 registers, the number of threads able to be executed concurrently in each SM will be reduced. Such reduction is done at the block granularity. For example, if

each block contains 512 threads, the reduction of threads will be done by reducing 512 threads at a time. Thus, the next lower number of threads from 1,536 would be 512, a one-third reduction of threads that can simultaneously reside in each SM. This can greatly reduce the number of warps available for scheduling, thus reducing the processor's ability to find useful work in the presence of long-latency operations.

Note that the number of registers available to each SM varies from device to device. An application can dynamically determine the number of registers available in each SM of the device used and choose a version of the kernel that uses the number of registers appropriate for the device. This can be done by calling the `cudaGetDeviceProperties()` function, the use of which was discussed in Section 4.6. Assume that variable `&dev_prop` is passed to the function for the device property, and the field `dev_prop.regsPerBlock` gives the number of registers available in each SM. For device D, the returned value for this field should be 16,384. The application can then divide this number by the target number of threads to reside in each SM to determine the number of registers that can be used in the kernel.

Shared memory usage can also limit the number of threads assigned to each SM. Assume device D has 16,384 (16 K) bytes of shared memory in each SM. Keep in mind that shared memory is used by blocks. Assume that each SM can accommodate up to eight blocks. To reach this maximum, each block must not use more than 2 K bytes of shared memory. If each block uses more than 2 K bytes of memory, the number of blocks that can reside in each SM is such that the total amount of shared memory used by these blocks does not exceed 16 K bytes. For example, if each block uses 5 K bytes of shared memory, no more than three blocks can be assigned to each SM.

For the matrix multiplication example, shared memory can become a limiting factor. For a tile size of 16×16, each block needs a $16 \times 16 \times 4 = 1$ K bytes of storage for `Mds`. Another 1 KB is needed for `Nds`. Thus, each block uses 2 K bytes of shared memory. The 16 K−byte shared memory allows eight blocks to simultaneous reside in an SM. Since this is the same as the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size. In this case, the real limitation is the threading hardware limitation that only 768 threads are allowed in each SM. This limits the number of blocks in each SM to three. As a result, only 3×2 KB = 6 KB of the shared memory will be used. These limits do change from device generation to the next but are

properties that can be determined at runtime, for example, the GT200 series of processors can support up to 1,024 threads in each SM.

Note that the size of shared memory in each SM can also vary from device to device. Each generation or model of device can have a different amount of shared memory in each SM. It is often desirable for a kernel to be able to use a different amount of shared memory according to the amount available in the hardware. That is, we may want to have a kernel to dynamically determine the size of the shared memory and adjust the amount of shared memory used. This can be done by calling the `cudaGetDeviceProperties()` function, the general use of which was discussed in Section 4.6. Assume that variable `&dev_prop` is passed to the function, the field `dev_prop.sharedMemPerBlock` gives the number of registers available in each SM. The programmer can then determine the number of amount of shared memory that should be used by each block.

Unfortunately, the kernel in Figure 5.12 does not support this. The declarations used in Figure 5.12 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. For example, assume that the file 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 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. We can add a C `extern` keyword in front of the shared memory declaration and omit the size of the array in the declaration. Based on this style, the declaration for `Mds` and `Nds` become:

```
extern __shared__ Mds[];
extern __shared__ Nds[];
```

Note that the arrays are now one dimensional. We will need to use a linearized index based on the vertical and horizontal indices.

At runtime, when we launch the kernel, we can dynamically determine the amount of shared memory to be used according to the device query result and supply that as a third configuration parameter to the kernel

launch. For example, the kernel launch statement in Figure 4.18 could be replaced with the following statements:

```
    size_t size =
        calculate_appropriate_SM_usage(dev_prop.
sharedMemPerBlock,...);
    matrixMulKernel<<<dimGrid, dimBlock, size>>>(Md, Nd, Pd,
Width);
```

where `size_t` is a built-in type for declaring a variable to hold the size information for dynamically allocated data structures. We have omitted the details of the calculation for setting the value of size at runtime.

5.6 SUMMARY

In summary, CUDA defines registers, shared memory, and constant memory that can be accessed at a higher speed and in a more parallel manner than the global memory. Using these memories effectively will likely require redesign of the algorithm. We use matrix multiplication as an example to illustrate tiled algorithms, a popular strategy to enhance locality of data access and enable effective use of shared memory. We demonstrate that with 16×16 tiling, global memory accesses are no longer the major limiting factor for matrix multiplication performance.

It is, however, important for CUDA programmers to be aware of the limited sizes of these types of memory. Their capacities are implementation dependent. Once their capacities are exceeded, they become limiting factors for the number of threads that can be simultaneously executing in each SM. The ability to reason about hardware limitations when developing an application is a key aspect of computational thinking. Readers are also referred to Appendix B for a summary of resource limitations of several different devices.

Although we introduced tiled algorithms in the context of CUDA 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. Therefore, readers will find the tiled algorithm useful when they develop a parallel application for other types of parallel computing systems using other programming models.

Our goal for this chapter is to introduce the different types of CUDA memory. We introduced tiled algorithm as an effective strategy for using shared memory. We have not discussed the use of constant memory, which will be explained in Chapter 8.

5.7 EXERCISES

5.1. Consider the matrix addition in Exercise 3.1. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: analyze the elements accessed by each thread and see if there is any commonality between threads.

5.2. Draw the equivalent of Figure 5.6 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.

5.3. What type of incorrect execution behavior can happen if one forgots to use `syncthreads()` in the kernel of Figure 5.12?

5.4. Assuming capacity was not an issue for registers or shared memory, give one case that it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.

5.5. 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?

    a. 1/8 of the original usage

    b. 1/16 of the original usage

    c. 1/32 of the original usage

    d. 1/64 of the original usage

5.6. Assume that a kernel is launched with 1,000 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?

    a. 1

    b. 1,000

      c. 512

      d. 512,000

5.7. 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?

      a. 1

      b. 1,000

      c. 512

      d. 51,200

5.8. Explain the difference between shared memory and L1 cache.

5.9. 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?

5.10. A kernel performs 36 floating-point operations and 7 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.

      a. Peak FLOPS = 200 GFLOPS, peak memory bandwidth = 100 GB/s.

      b. Peak FLOPS = 300 GFLOPS, peak memory bandwidth = 250 GB/s.

5.11. Indicate which of the following assignments per streaming multiprocessor is possible. In the case where it is not possible, indicate the limiting factor(s).

      a. 4 blocks with 128 threads each and 32 B shared memory per thread on a device with compute capability 1.0.

b. 8 blocks with 128 threads each and 16 B shared memory
per thread on a device with compute capability 1.0.

c. 16 blocks with 32 threads each and 64 B shared memory
per thread on a device with compute capability 1.0.

d. 2 blocks with 512 threads each and 32 B shared memory
per thread on a device with compute capability 1.2.

e. 4 blocks with 256 threads each and 16 B shared memory
per thread on a device with compute capability 1.2.

f. 8 blocks with 256 threads each and 8 B shared memory
per thread on a device with compute capability 1.2.

This page intentionally left blank

Performance Considerations

6

CHAPTER OUTLINE

The execution speed of a CUDA kernel can vary greatly depending on the resource constraints of the device being used. In this chapter, we will discuss the major types of resource constraints in a CUDA device and how they can affect the kernel execution performance in this device. To achieve his or her goals, a programmer often has to find ways to achieve a required level of performance that is higher than that of an initial version of the application. In different applications, different constraints may dominate and become the limiting factors. One can improve the performance of an application on a particular CUDA device, sometimes dramatically, by trading one resource usage for another. This strategy works well if the resource constraint alleviated was actually the dominating constraint before the strategy was applied, and the one exacerbated does not have negative effects on parallel execution. Without such understanding, performance tuning would be guess work; plausible strategies may or may not lead to performance enhancements. Beyond insights into these resource constraints, this chapter further offers principles and case studies designed to cultivate intuition about the type of algorithm patterns that can result in high-performance execution. It is also establishes idioms and ideas that

will likely lead to good performance improvements during your performance tuning efforts.
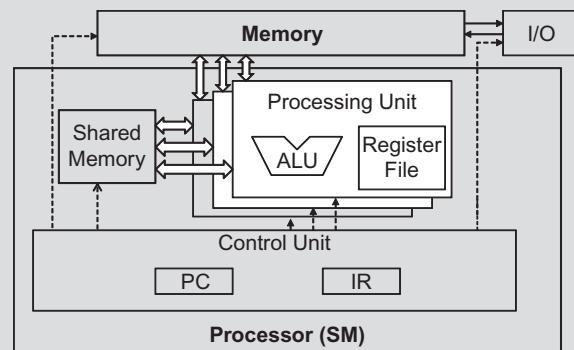
6.1 WARPS AND THREAD EXECUTION

Let's first discuss some aspects of thread execution that can limit performance. Recall that launching a CUDA kernel generates a grid of threads that are organized as a two-level hierarchy. At the top level, a grid consists of a 1D, 2D, or 3D array of blocks. At the bottom level, each block, in turn, consists of a 1D, 2D, or 3D array of threads. In Chapter 4, we saw that blocks can execute in any order relative to each other, which allows for transparent scalability in parallel execution of CUDA kernels. However, we did not say much about the execution timing of threads within each block.

WARPS AND SIMD HARDWARE

The motivation for executing threads as warps is illustrated in the following diagram (same as Figure 5.4). The processor has only one control unit that fetches and decodes instructions. The same control signal goes to multiple processing units, each of which executes one of the threads in a warp. Since all processing units are controlled by the same instruction, 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 are controlled by an instruction

```
add r1, r2, r3
```
the r2 and r3 values 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 memory. They include on-chip instruction caches to reduce the latency of instruction fetch. Having multiple processing

units share a control unit can result in significant reduction in hardware manufacturing cost and power consumption.

As the processors are increasingly power-limited, new processors will likely use SIMD designs. In fact, we may see even more processing units sharing a control unit in the future.

Conceptually, one should assume that threads in a block can execute in any order with respect to each other. Barrier synchronizations should be used whenever we want to ensure all threads have completed a common phase of their execution before any of them start the next phase. The correctness of executing a kernel should not depend on the fact that certain threads will execute in synchrony with each other. Having said this, we also want to point out that due to various hardware cost considerations, current CUDA devices actually bundle multiple threads for execution. Such an implementation strategy leads to performance limitations for certain types of kernel function code constructs. It is advantageous for application developers to change these types of constructs to other equivalent forms that perform better.

As we discussed in Chapter 4, current CUDA devices bundle several threads for execution. Each thread block is partitioned into *warps*. The execution of warps are implemented by an SIMD hardware (see "Warps and SIMD Hardware" sidebar). This implementation technique helps to reduce hardware manufacturing cost, lower runtime operation electricity cost, and enable some optimizations in servicing memory accesses. In the foreseeable future, we expect that warp partitioning will remain as a popular implementation technique. However, the size of a warp can easily vary from implementation to implementation. Up to this point in time, all CUDA devices have used similar warp configurations where each warp consists of 32 threads.
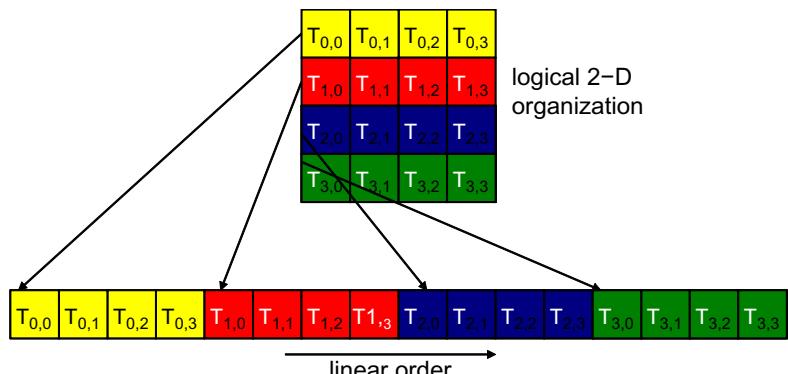
Thread blocks are partitioned into warps based on thread indices. If a thread block is organized into a 1D array (i.e., only `threadIdx.x` is used), the partition is straightforward; `threadIdx.x` values within a warp are consecutive and increasing. For a warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63. In general, warp n starts with thread $32 \times n$ and ends with thread $32(n + 1) - 1$. For a block of which the size is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32 threads. For example, if a block has 48 threads, it will be partitioned into two warps, and its warp 1 will be padded with 16 extra threads.

For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linear order before partitioning into warps.

The linear order 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 would form the linear order by placing all threads of which `threadIdx.y` is 1 after those of which `threadIdx.y` is 0, threads of which `threadIdx.y` is 2 after those of which `threadIdx.y` is 1, and so on.

Figure 6.1 shows an example of placing threads of a 2D block into linear order. The upper part shows the 2D view of the block. Readers should recognize the similarity with the row-major layout of 2D arrays in C, as shown in Figure 4.3. Each thread is shown as $T_{y,x}$, x being `threadIdx.x` and y being `threadIdx.y`. The lower part of Figure 6.1 shows the linear view of the block. The first four threads are those threads of which the `threadIdx.y` value is 0; they are ordered with increasing `threadIdx.x` values. The next four threads are those threads of which the `threadIdx.y` value is 1; they are also placed with increasing `threadIdx.x` values. For this example, all 16 threads form half a warp. The warp will be padded with another 16 threads to complete a 32-thread warp. Imagine a 2D 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 a useful exercise to draw out the picture as an exercise.

For a 3D block, we first place all threads of which the `threadIdx.z` value is 0 into the linear order. Among these threads, they are treated as a 2D block as shown in Figure 6.1. All threads of which the `threadIdx.z` value is 1 will then be placed into the linear order, and so on. For a 3D


FIGURE 6.1

Placing 2D threads into linear order.

thread block of dimensions $2 \times 8 \times 4$ (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.

The SIMD hardware executes all threads of a warp as a bundle. An instruction is run for all threads in the same warp. It works well when all threads within a warp follow the same execution path, or more formally referred to as control flow, when working their data. For example, for an `if-else` construct, the execution works well when either all threads execute the `if` part or all execute the `else` part. When threads within a warp take different control flow paths, the SIMD hardware will take multiple passes through these divergent paths. One pass executes those threads that follow the `if` part and another pass executes those that follow the `else` part. During each pass, the threads that follow the other path are not allowed to take effect. These passes are sequential to each other, thus they will add to the execution time.

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 the threads take effect in each pass only, allowing every thread to take its own control flow path. This preserves the independence of threads while taking advantage of the reduced cost of SIMD hardware.

When threads in the same warp follow different paths of control flow, we say that these threads *diverge* in their execution. In the `if-else` example, divergence arises if some threads in a warp take the `then` path and some the `else` path. The cost of divergence is the extra pass the hardware needs to take to allow the threads in a warp to make their own decisions. Divergence also can arise in other constructs; for example, if threads in a warp execute a `for` loop that can iterate six, seven, or eight times for different threads. All threads will finish the first six iterations together. Two passes will be used to execute the seventh iteration, one for those that take the iteration and one for those that do not. Two passes will be used to execute the eighth iteration, one for those that take the iteration and one for those that do not.

In terms of source statements, a control construct can result in thread divergence when its decision condition is based on `threadIdx` values. For example, the statement `if (threadIdx.x > 2) {}` causes the threads to follow two divergent control flow paths. Threads 0, 1, and 2 follow a different path than threads 3, 4, 5, etc. Similarly, a loop can cause thread

divergence if its loop condition is based on thread index values. Such usages arise naturally in some important parallel algorithms. We will use a reduction algorithm to illustrate this point.

A reduction algorithm derives a single value from an array of values. The single value could be the sum, the maximal value, the minimal value, etc. among all elements. All these types of reductions share the same computation structure. A reduction can be easily done by sequentially going through every element of the array. When an element is visited, the action to take depends on the type of reduction being performed. For a sum reduction, the value of the element being visited at the current step, or the current value, is added to a running sum. For a maximal reduction, the current value is compared to a running maximal value of all the elements visited so far. If the current value is larger than the running maximal, the current element value becomes the running maximal value. For a minimal reduction, the value of the element currently being visited is compared to a running minimal. If the current value is smaller than the running minimal, the current element value becomes the running minimal. The sequential algorithm ends when all the elements are visited. The sequential reduction algorithm is work-efficient in that every element is only visited once and only a minimal amount of work is performed when each element is visited. Its execution time is proportional to the number of elements involved. That is, the computational complexity of the algorithm is $O(N)$, where N is the number of elements involved in the reduction.

The time needed to visit all elements of a large array motivates parallel execution. A parallel reduction algorithm typically resembles the structure of a soccer tournament. In fact, the elimination process of the World Cup is a reduction of "maximal" where the maximal is defined as the team that "beats" all other teams. The tournament "reduction" is done by 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, the winners of which advance to the third round, etc. With 16 teams entering a tournament, 8 winners will emerge from the first round, 4 from the second round, 2 from the third round, and 1 final winner from the fourth round. It should be easy to see that even with 1,024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough soccer fields 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 fields, even with 60,000 teams, we can determine the final winner in just 16 rounds. Of course, one would need to have enough

soccer fields and enough officials to accommodate the 30,000 games in the first round, etc.

Figure 6.2 shows a kernel function that performs parallel sum reduction. The original array is in the global memory. Each thread block reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction. The code that loads the elements from global memory into the shared memory is omitted from Figure 6.2 for brevity. The reduction is done *in place*, which means the elements in the shared memory will be replaced by partial sums. Each iteration of the while loop in the kernel function implements a round of reduction. The __syncthreads() statement (line 5) in the while loop ensures that all partial sums for the previous iteration have been generated and thus all threads are ready to enter the current iteration before any one of them is allowed to do so. This way, all threads that enter the second iteration will be using the values produced in the first iteration. After the first round, the even elements will be replaced by the partial sums generated in the first round. After the second round, the elements of which the indices are multiples of four will be replaced with the partial sums. After the final round, the total sum of the entire section will be in element 0.

In Figure 6.2, line 3 initializes the stride variable to 1. During the first iteration, the if statement in line 6 is used to select only the even threads to perform addition between two neighboring elements. The execution of the kernel is illustrated in Figure 6.3. The threads and the array element values are shown in the horizontal direction. The iterations taken by the threads are shown in the vertical direction with time progressing from top to bottom. Each row of Figure 6.3 shows the contents of the array elements after an iteration of the for loop.
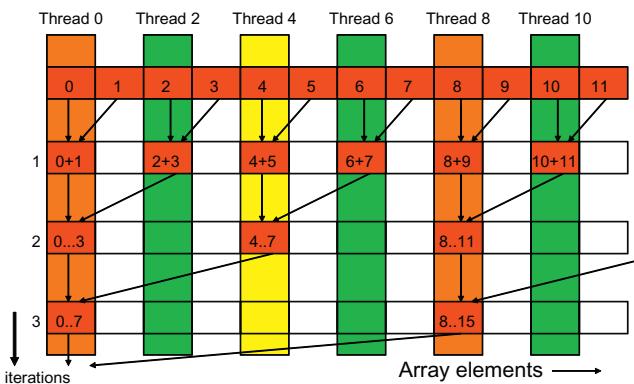
```
1. __shared__ float partialSum[]
...
2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = 1;  stride < blockDim.x; stride *= 2)
4. {
5.     __syncthreads();
6.     if (t % (2*stride) == 0)
7.         partialSum[t] += partialSum[t+stride];
8 }
```

FIGURE 6.2

A simple sum reduction kernel.

FIGURE 6.3

Execution of the sum reduction kernel.

As shown in Figure 6.3, the even elements of the array hold the pairwise partial sums after iteration 1. Before the second iteration, the value of the `stride` variable is doubled to 2. During the second iteration, only those threads of which the indices are multiples of four will execute the `add` statement in line 8. Each thread generates a partial sum that includes four elements, as shown in row 2. With 512 elements in each section, the kernel function will generate the sum of the entire section after nine iterations. By using `blockDim.x` as the loop bound in line 4, the kernel assumes that it is launched with the same number of threads as the number of elements in the section. That is, for a section size of 512, the kernel needs to be launched with 512 threads.[1]

Let's analyze the total amount of work done by the kernel. Assume that the total number of elements to be reduced is N. The first round requires $N/2$ additions. The second round requires $N/4$ additions. The final round has only one addition. There are $\log_2(N)$ rounds. The total number of additions performed by the kernel is $N/2 + N/4 + N/8 + \ldots + 1 = N - 1$. Therefore, the computational complexity of the reduction algorithm is $O(N)$. The algorithm is work-efficient. However, we also need to make sure that the hardware is efficiently utilized while executing the kernel.

---

[1]Note that using the same number of threads as the number of elements in a section is wasteful. Half of the threads in a block will never execute. Readers are encouraged to modify the kernel and the kernel launch execution configuration parameters to eliminate this waste (see Exercise 6.1).

```
1. __shared__ float partialSum[]

2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = blockDim.x;  stride > 1;  stride /= 2)
4. {
5.    __syncthreads();
6.    if (t < stride)
7.     partialSum[t] += partialSum[t+stride];
8. }
```

FIGURE 6.4
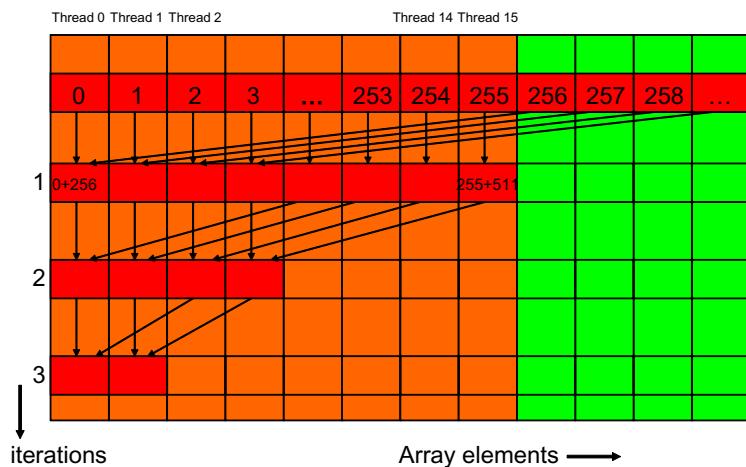
A kernel with fewer thread divergence.

The kernel in Figure 6.2 clearly has thread divergence. During the first iteration of the loop, only those threads of which the threadIdx.x are even will execute the add statement. One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute line 8. In each successive iteration, fewer threads will execute line 8 but two passes will be still needed to execute all the threads during each iteration. This divergence can be reduced with a slight change to the algorithm.

Figure 6.4 shows a modified kernel with a slightly different algorithm for sum reduction. Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other. It does so by initializing the stride to be half the size of the section. All pairs added during the first round are half the section size away from each other. After the first iteration, all the pairwise sums are stored in the first half of the array. The loop divides the stride by 2 before entering the next iteration. Thus, for the second iteration, the stride variable value is one-quarter of the section size—that is, the threads add elements that are one-quarter a section away from each other during the second iteration.

Note that the kernel in Figure 6.4 still has an if statement (line 6) in the loop. The number of threads that execute line 7 in each iteration is the same as in Figure 6.2. So, why should there be a performance difference between the two kernels? The answer lies in the positions of threads that execute line 7 relative to those that do not.

Figure 6.5 illustrates the execution of the revised kernel. During the first iteration, all threads of which the threadIdx.x values are less than half of the size of the section execute line 7. For a section of 512 elements, threads 0−255 execute the add statement during the first iteration

FIGURE 6.5

Execution of the revised algorithm.

while threads 256−511 do not. The pairwise sums are stored in elements 0−255 after the first iteration. Since the warps consist of 32 threads with consecutive threadIdx.x values, all threads in warps 0−7 execute the add statement, whereas warps 8−15 all skip the add statement. Since all threads in each warp take the same path, there is no thread divergence!

The kernel in Figure 6.4 does not completely eliminate the divergence due to the if statement. Readers should verify that starting with the fifth iteration, the number of threads that execute line 7 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 10 to 5.

6.2 GLOBAL MEMORY BANDWIDTH

One of the most important factors of CUDA kernel performance is accessing data in the global memory. CUDA applications exploit massive data parallelism. Naturally, CUDA applications tend to process a massive amount of data from the global memory within a short period of time. In Chapter 5, we discussed tiling techniques that utilize shared memories to reduce the total amount of data that must be accessed by a collection of threads in the thread block. In this chapter, we will further discuss memory

coalescing techniques that can more effectively move data from the global memory into shared memories and registers. Memory coalescing techniques are often used in conjunction with tiling techniques to allow CUDA devices to reach their performance potential by more efficiently utilizing the global memory bandwidth.[2]
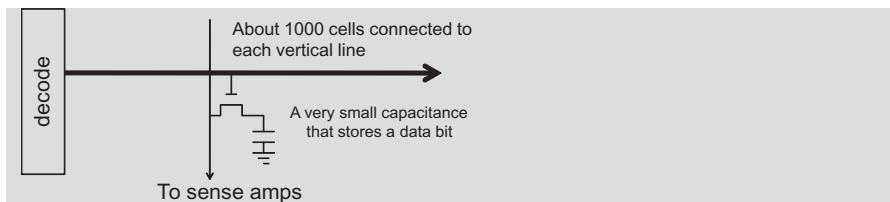
The global memory of a CUDA device is implemented with DRAMs. Data bits are stored in DRAM cells that are small capacitors, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1. Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to a sensor and set off its detection mechanism that determines whether a sufficient amount of charge is present in the capacitor to qualify as a "1" (see "Why Are DRAMs So Slow?" sidebar). This process takes tens of nanoseconds in modern DRAM chips. Because this is a very slow process relative to the desired data access speed (sub-nanosecond access per byte), modern DRAMs use parallelism to increase their rate of data access.

Each time a DRAM location is accessed, many consecutive locations that include the requested location are actually accessed. Many sensors are provided in each DRAM chip and they 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 very high speed to the processor. If an application can make focused use of data from consecutive locations, the DRAMs can supply the data at a much higher rate than if a truly random sequence of locations were accessed.

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.

---

[2]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 a significant effect on kernel execution performance in the foreseeable future.

About 1000 cells connected to
each vertical line

A very small capacitance
that stores a data bit

decode

To sense amps

A more formidable challenge is for the cell to drive the line to the sense amplifiers and allow the sense amplifier to detect its content. This is based on electrical charge sharing. The gate lets out the tiny amount of electrical charge stored in the cell. If the cell content is "1," the tiny amount of charge must raise the potential of the large capacitance formed by the long bit line and the input 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 for another person to smell the aroma propagated through the hallway to determine the flavor of the coffee.

One could speed up the process by using a larger, stronger capacitor in each cell. However, the DRAMs have been going in the opposite direction. The capacitors in each cell have been steadily reduced in size 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.

Recognizing the organization of modern DRAMs, current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. That is, 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 N,[3] thread 1 location $N + 1$, thread 2 location $N + 2$, and so on, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAMs. Such coalesced access allows the DRAMs to deliver data at a rate close to the peak global memory bandwidth.
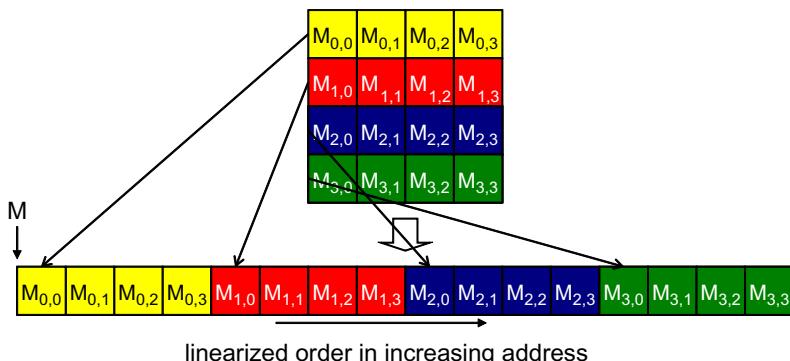
To understand how to effectively use coalescing hardware, we need to review how the memory addresses are formed in accessing C

---

[3]Different CUDA devices may also impose alignment requirements on N. For example, in some CUDA devices, N is required to be aligned to 16-word boundaries. That is, the lower 6 bits of N should all be 0 bits. We will discuss techniques that address this alignment requirement in Chapter 12.
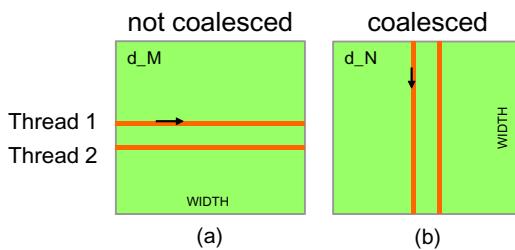
multidimensional array elements. As we showed in Chapter 4 (Figure 4.3, replicated as Figure 6.6 for convenience), multidimensional array elements in C and CUDA are placed into the linearly addressed memory space according to the row-major convention. That is, the elements of row 0 of a matrix are first placed in order into consecutive locations. They are followed by the elements of row 1 of the matrix, and so on. In other words, all elements in a row are placed into consecutive locations and entire rows are placed one after another. The term *row major* 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. Figure 6.6 shows a small example where the 16 elements of a 4×4 matrix M are placed into linearly addressed locations. 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 2D matrix, are placed four locations away in the linearly addressed memory.

Figure 6.7 illustrates the favorable versus unfavorable CUDA kernel 2D row-major array data access patterns for memory coalescing. Recall from Figure 4.7 that in our simple matrix−matrix multiplication kernel, each thread accesses a row of the d_M array and a column of the d_N array. Readers should review Section 4.3 before continuing. Figure 6.7(a) illustrates the data access pattern of the d_M array, where threads in a warp read adjacent rows. That is, during iteration 0, threads in a warp read element 0 of rows 0−31. During iteration 1, these same threads read element 1 of rows 0−31. None of the accesses will be coalesced. A more favorable access pattern is shown in Figure 6.7(b), where each thread reads a column


linearized order in increasing address
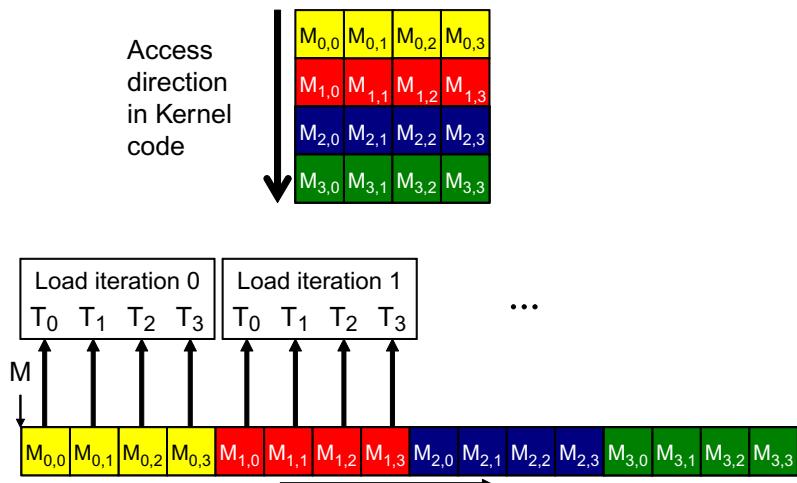
FIGURE 6.6

Placing matrix elements into linear order.

FIGURE 6.7

Memory access patterns in C 2D arrays for coalescing.

FIGURE 6.8

A coalesced access pattern.

of d_N. During iteration 0, threads in warp 0 read element 1 of columns 0−31. All these accesses will be coalesced.

To understand why the pattern in Figure 6.7(b) is more favorable than that in Figure 6.7(a), we need to review how these matrix elements are accessed in more detail. Figure 6.8 shows a small example of the favorable access pattern in accessing a 4×4 matrix. The arrow in the top portion of Figure 6.8 shows the access pattern of the kernel code for one thread. This access pattern is generated by the access to d_N in Figure 4.7:

```
d_N[k*Width + Col]
```

Within a given iteration of the `k` loop, the `k*Width` value is the same across all threads. Recall that `Col = blockIdx.x*blockDim.x + threadIdx.x`. Since the value of `blockIndx.x` and `blockDim.x` are of the same value for all threads in the same block, the only part of `k*Width + Col` that varies across a thread block is `threadIdx.x`. For example, in Figure 6.8, assume that we are using 4×4 blocks and that the warp size is 4. That is, for this toy example, we are using only one block to calculate the entire P matrix. The values of `Width`, `blockDim.x`, and `blockIdx.x` are 4, 4, and 0, respectively, for all threads in the block. In iteration 0, the `k` value is 0. The index used by each thread for accessing `d_N` is

```
d_N[k*Width + Col]=d_N[k*Width + blockIdx.x*blockDim.x
 + threadIdx.x]
= d_N[0*4 + 0*4 + threadidx.x]
= d_N[threadIdx.x]
```

That is, the index for accessing `d_N` is simply the value of `threadIdx.x`. The `d_N` elements accessed by T_0, T_1, T_2, and T_3 are `d_N[0]`, `d_N[1]`, `d_N[2]`, and `d_N[3]`, respectively. This is illustrated with the "Load iteration 0" box of Figure 6.8. These elements are in consecutive locations in the global memory. The hardware detects that these accesses are made by threads in a warp and to consecutive locations in the global memory. It coalesces these accesses into a consolidated access. This allows the DRAMs to supply data at a high rate.

During the next iteration, the `k` value is 1. The index used by each thread for accessing `d_N` becomes
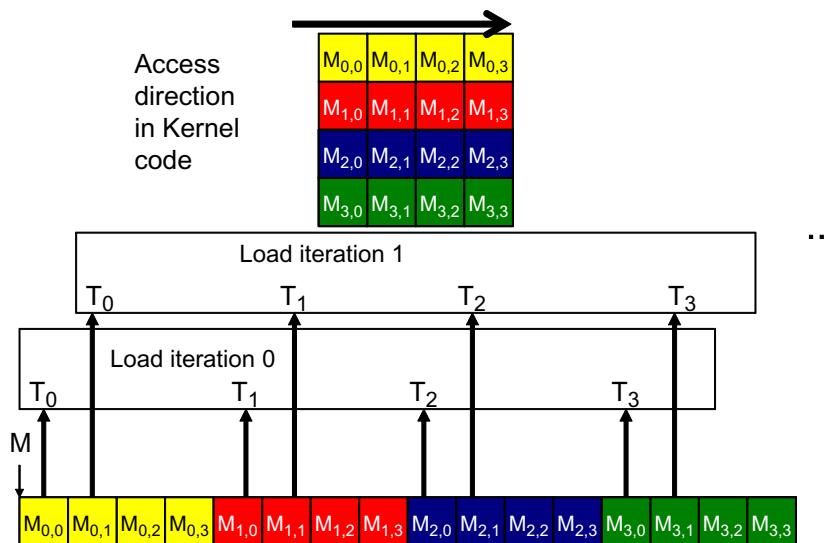
```
d_N[k*Width + Col]=d_N[k*Width + blockIdx.x*blockDim.x
 + threadIdx.x]
= d_N[1*4 + 0*4 + threadidx.x]
= d_N[4 + threadIdx.x]
```

The `d_N` elements accessed by T_0, T_1, T_2, and T_3 are `d_N[5]`, `d_N[6]`, `d_N[7]`, and `d_N[8]`, respectively, as shown with the "Load iteration 1" box in Figure 6.8. All these accesses are again coalesced into a consolidated access for improved DRAM bandwidth utilization.

Figure 6.9 shows an example of a matrix data access pattern that is not coalesced. The arrow in the top portion of the figure shows that the kernel code for each thread accesses elements of a row in sequence. The arrow in the top portion of Figure 6.9 shows the access pattern of the kernel code for one thread. This access pattern is generated by the access to `d_M` in Figure 4.7:

```
d_M[Row*Width + k]
```

Within a given iteration of the `k` loop, the `k*Width` value is the same across all threads. Recall that `Row = blockIdx.y*blockDim.y +`

FIGURE 6.9

An uncoalesced access pattern.

`threadIdx.y`. Since the value of `blockIndx.y` and `blockDim.y` are of the same value for all threads in the same block, the only part of `Row*Width + k` that can vary across a thread block is `threadIdx.y`. In Figure 6.9, assume again that we are using 4×4 blocks and that the warp size is 4. The values of `Width`, `blockDim.y`, and `blockIdx.y` are 4, 4, and 0, respectively, for all threads in the block. In iteration 0, the `k` value is 0. The index used by each thread for accessing `d_N` is

```
d_M[Row*Width + k] = d_M[(blockIdx.y*blockDim.y + threadIdx.y)*
 Width + k]
 = d_M[((0*4 + threadIdx.y)*4 + 0]
 = d_M[threadIdx.x*4]
```

That is, the index for accessing `d_M` is simply the value of `threadIdx.x*4`. The `d_M` elements accessed by T_0, T_1, T_2, and T_3 are `d_M[0]`, `d_M[4]`, `d_M[8]`, and `d_M[12]`. This is illustrated with the "Load iteration 0" box of Figure 6.9. These elements are not in consecutive locations in the global memory. The hardware cannot coalesce these accesses into a consolidated access.

During the next iteration, the `k` value is 1. The index used by each thread for accessing `d_M` becomes
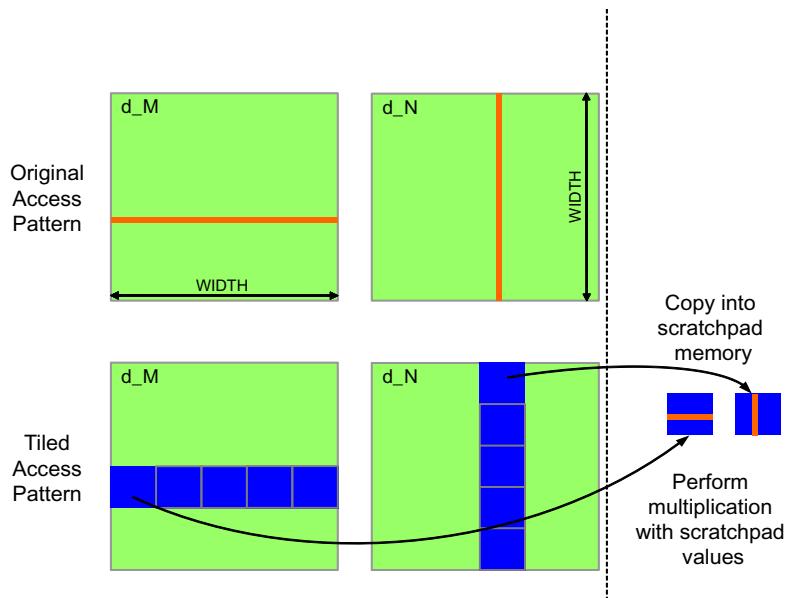
```
d_M[Row*Width + k] = d_M[(blockIdx.y*blockDim.y + threadIdx.y)*
 Width + k]
```

```
= d_M[(0*4 + threadidx.x)*4 + 1]
= d_M[threadIdx.x*4 + 1]
```

The d_M elements accessed by T_0, T_1, T_2, T_3 are d_M[1], d_M[5], d_M[9], and d_M[13], respectively, as shown with the "Load iteration 1" box in Figure 6.9. All these accesses again cannot be coalesced into a consolidated access.

For a realistic matrix, there are typically hundreds or even thousands of elements in each dimension. The elements accessed in each iteration by neighboring threads can be hundreds or even thousands of elements apart. The "Load iteration 0" box in the bottom portion shows how the threads access these nonconsecutive locations in the 0 iteration. The hardware will determine that accesses to these elements are far away from each other and cannot be coalesced. As a result, when a kernel loop iterates through a row, the accesses to global memory are much less efficient than the case where a kernel iterates through a column.

If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the shared memory to enable memory coalescing. The technique is illustrated in Figure 6.10 for matrix multiplication. Each thread reads a row from d_M, a pattern that cannot be


FIGURE 6.10

Using shared memory to enable coalescing.

coalesced. Fortunately, a tiled algorithm can be used to enable coalescing. As we discussed in Chapter 5, threads of a block can first cooperatively load the tiles into the shared memory. Care must be taken to ensure that these tiles are loaded in a coalesced pattern. Once the data is in shared memory, it can be accessed either on a row basis or a column basis with much less performance variation because the shared memories are implemented as intrinsically high-speed, on-chip memory that does not require coalescing to achieve a high data access rate.

We replicate Figure 5.7 here as Figure 6.11, where the matrix multiplication kernel loads two tiles of matrix d_M and d_N into the shared memory. Note that each thread in a thread block is responsible for loading one d_M element and one d_N element into Mds and Nds in each phase as defined by the for loop in line 8. Recall that there are TILE_WIDTH² threads involved in each tile. The threads use threadIdx.y and threadIdx.y to determine the element of each matrix to load.

The d_M elements are loaded in line 9, where the index calculation for each thread uses m to locate the left end of the tile. Each row of the tile is then loaded by TILE_WIDTH threads of which the threadIdx differ in the *x* dimension. Since these threads have consecutive threadIdx.x values, they are in

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
1.   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the d_P element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.     float Pvalue = 0;
// Loop over the d_M and d_N tiles required to compute the d_P element
8.     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Coolaborative loading of d_M and d_N tiles into shared memory
9.       Mds[tx][ty] = d_M[Row*Width + m*TILE_WIDTH+tx];
10.      Nds[tx][ty] = d_N[(m*TILE_WIDTH+ty)*Width + Col];
11.      __syncthreads();

12.      for (int k = 0; k < TILE_WIDTH; ++k)
13.        Pvalue += Mds[tx][k] * Nds[k][ty];
14.      __syncthreads();
    }
15.    d_P[Row*Width+Col] = Pvalue;
}
```

FIGURE 6.11

Tiled matrix multiplication kernel using shared memory.

the same warp. Also, the index calculation `d_M[row][m*TILE_SIZE + tx]` makes these threads access elements in the same row. The question is whether adjacent threads in the warp indeed access adjacent elements in the row. Recall that elements in the same row are placed into consecutive locations of the global memory. Since the column index `m*TILE_SIZE + tx` is such that all threads with adjacent `tx` values will access adjacent row elements, the answer is yes. The hardware detects that these threads in the same warp access consecutive locations in the global memory and combines them into a coalesced access.

In the case of `d_N`, the row index `m*TILE_SIZE + ty` has the same value for all threads in the same warp; they all have the same `ty` value. Thus, threads in the same warp access the same row. The question is whether the adjacent threads in a warp access adjacent elements of a row. Note that the column index calculation for each thread `Col` is based on `bx*TILE_SIZE + tx` (see line 4). Therefore, adjacent threads in a warp access adjacent elements in a row. The hardware detects that these threads in the same warp access consecutive locations in the global memory and combine them into a coalesced access.

Readers shall find it useful to draw a picture based on the kernel code in Figure 6.11 and identify the `threadIdx.y` and `threadIdx.x` values of the thread that loads each element of the tile. Lines 5, 6, 9, and 10 in Figure 6.11 form a frequently used programming pattern for loading matrix elements into shared memory in tiled algorithms. We would also like to encourage readers to analyze the data access pattern by the dot-product loop in lines 12 and 13. Note that the threads in a warp do not access consecutive location of `Mds`. This is not a problem since `Mds` is in shared memory, which does not require coalescing to achieve high-speed data access.

6.3 DYNAMIC PARTITIONING OF EXECUTION RESOURCES

The execution resources in a streaming multiprocessor (SM) include registers, shared memory, thread block slots, and thread slots. These resources are dynamically partitioned and assigned to threads to support their execution. In Chapter 4, we have seen that the current generation of devices have 1,536 thread slots, each of which can accommodate one thread. These thread slots are partitioned and assigned to thread blocks during runtime. If each thread block consists of 512 threads, the 1,536 thread slots

are partitioned and assigned to three blocks. In this case, each SM can accommodate up to three thread blocks due to limitations on thread slots. If each thread block contains 128 threads, the 1,536 thread slots are partitioned and assigned to 12 thread blocks. The ability to dynamically partition the thread slots among thread blocks makes SMs versatile. They can either execute many thread blocks each having few threads, or execute few thread blocks each having many threads. This is in contrast to a fixed partitioning method where each block receives a fixed amount of resources regardless of their real needs. Fixed partitioning results in wasted thread slots when a block has few threads and fails to support blocks that require more thread slots than the fixed partition allows.

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. For example, if each block has 128 threads, the 1,536 thread slots can be partitioned and assigned to 12 blocks. However, since there are only 8 block slots in each SM, only 8 blocks will be allowed. This means that only 1,024 of the thread slots will be utilized. Therefore, to fully utilize both the block slots and thread slots, one needs at least 256 threads in each block.

As we mentioned in Chapter 4, the automatic variables declared in a CUDA kernel are placed into registers. Some kernels may use lots of automatic variables and others may use few of them. Thus, one should expect that some kernels require many registers and some require fewer. By dynamically partitioning the registers among blocks, the SM can accommodate more blocks if they require few registers and fewer blocks if they require more registers. One does, however, need to be aware of potential interactions between register limitations and other resource limitations.

In the matrix multiplication example, assume that each SM has 16,384 registers and the kernel code uses 10 registers per thread. If we have 16×16 thread blocks, how many threads can run on each SM? We can answer this question by first calculating the number of registers needed for each block, which is $10 \times 16 \times 16 = 2,560$. The number of registers required by six blocks is 15,360, which is under the 16,384 limit. Adding another block would require 17,920 registers, which exceeds the limit. Therefore, the register limitation allows blocks that altogether have 1,536 threads to run on each SM, which also fits within the limit of block slots and 1,536 thread slots.

Now assume that the programmer declares another two automatic variables in the kernel and bumps the number of registers used by each thread to 12. Assuming the same 16×16 blocks, each block now requires

$12 \times 16 \times 16 = 3,072$ registers. The number of registers required by six blocks is now 18,432, which exceeds the register limitation. The CUDA runtime system deals with this situation by reducing the number of blocks assigned to each SM by one, thus reducing the number of registered required to 15,360. This, however, reduces the number of threads running on an SM from 1,536 to 1,280. That is, by using two extra automatic variables, the program saw a one-sixth reduction in the warp parallelism in each SM. This is sometimes a referred to as a "performance cliff" where a slight increase in resource usage can result in significant reduction in parallelism and performance achieved [RRS2008]. Readers are referred to the CUDA Occupancy Calculator [NVIDIA], which is a downloadable Excel sheet that calculates the actual number of threads running on each SM for a particular device implementation given the usage of resources by a kernel.
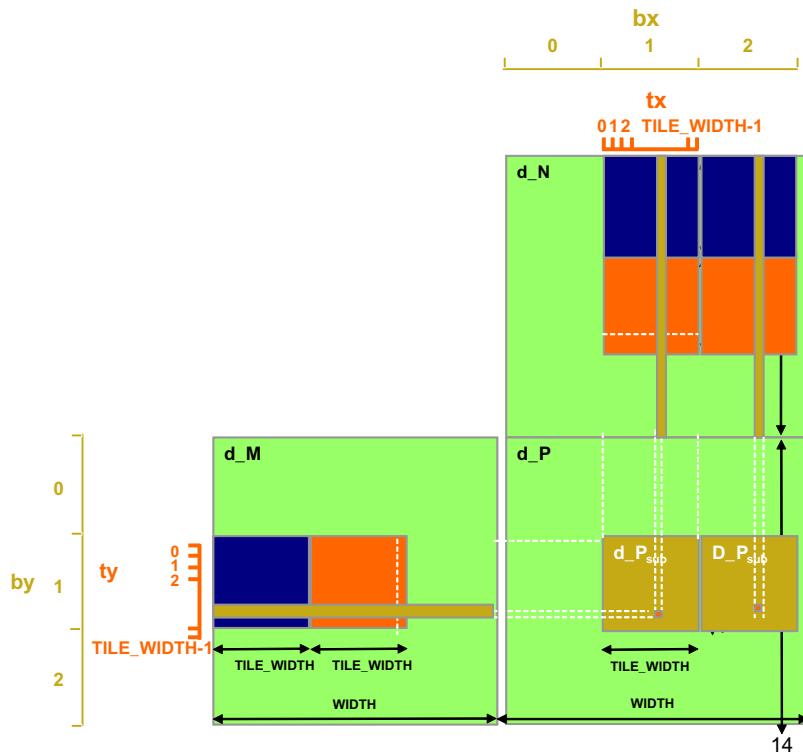
6.4 INSTRUCTION MIX AND THREAD GRANULARITY

An important algorithmic decision in performance tuning is the granularity of threads. It is often advantageous to put more work into each thread and use fewer threads. Such advantage arises when some redundant work exists between threads. In the current generation of devices, each SM has limited instruction processing bandwidth. Every instruction consumes instruction processing bandwidth, whether it is a floating-point calculation instruction, a load instruction, or a branch instruction. Eliminating redundant instructions can ease the pressure on the instruction processing bandwidth and improve the overall execution speed of the kernel.

Figure 6.12 illustrates such an opportunity in matrix multiplication. The tiled algorithm in Figure 6.11 uses one thread to compute one element of the output d_P matrix. This requires a dot product between one row of d_M and one column of d_N.

The opportunity of thread granularity adjustment comes from the fact that multiple blocks redundantly load each d_M tile. As shown in Figure 6.12, the calculation of two d_P elements in adjacent tiles uses the same d_M row. With the original tiled algorithm, the same d_M row is redundantly loaded by the two blocks assigned to generate these two Pd tiles. One can eliminate this redundancy by merging the two thread blocks into one. Each thread in the new thread block now calculates two d_P elements. This is done by revising the kernel so that two dot products are computed by the innermost loop of the kernel. Both dot products use the

FIGURE 6.12

Increased thread granularity with rectangular tiles.

same `Mds` row but different `Nds` columns. This reduces the global memory access by one-quarter. Readers are encouraged to write the new kernel as an exercise.

The potential downside is that the new kernel now uses even more registers and shared memory. As we discussed in the previous section, the number of blocks that can be running on each SM may decrease. It also reduces the total number of thread blocks by half, which may result in an insufficient amount of parallelism for matrices of smaller dimensions. In practice, we found that combining up to four adjacent horizontal blocks to compute adjacent horizontal tiles improves the performance of large $(2{,}048 \times 2{,}048$ or more) matrix multiplication.

6.5 SUMMARY

In this chapter, we reviewed the major aspects of CUDA C application performance on a CUDA device: control flow divergence, global memory coalescing, dynamic resource partitioning, and instruction mixes. We presented practical techniques for creating good program patterns for these performance aspects. We will continue to study practical applications of these techniques in the case studies in the next few chapters.

6.6 EXERCISES

6.1 The kernels in Figures 6.2 and 6.4 are wasteful in their use of threads; half of the threads in each block never execute. Modify the kernels to eliminate such waste. Give the relevant execute configuration parameter values at the kernel launch. Is there a cost in terms of an extra arithmetic operation needed? Which resource limitation can be potentially addressed with such modification? (Hint: line 2 and/or line 4 can be adjusted in each case; the number of elements in the section may increase.)

6.2 Compare the modified kernels you wrote for Exercise 6.1. Which modification introduced fewer additional arithmetic operations?

6.3 Write a complete kernel based on Exercise 6.1 by (1) adding the statements that load a section of the input array from global memory to shared memory, (2) using `blockIdx.x` to allow multiple blocks to work on different sections of the input array, and (3) writing the reduction value for the section to a location according to the `blockIdx.x` so that all blocks will deposit their section reduction value to the lower part of the input array in global memory.

6.4 Design a reduction program based on the kernel you wrote for Exercise 6.3. The host code should (1) transfer a large input array to the global memory, and (2) use a loop to repeatedly invoke the kernel you wrote for Exercise 6.3 with adjusted execution configuration parameter values so that the reduction result for the input array will eventually be produced.

6.5 For the matrix multiplication kernel in Figure 6.11, draw the access patterns of threads in a warp of lines 9 and 10 for a small 16×16 matrix size. Calculate the `tx` and `ty` values for each thread in a warp

and use these values in the d_M and d_N index calculations in lines 9 and 10. Show that the threads indeed access consecutive d_M and d_N locations in global memory during each iteration.

6.6 For the simple matrix−matrix multiplication (M − × N) based on row-major layout, which input matrix will have coalesced accesses?

   a. M

   b. N

   c. Both

   d. Neither

6.7 For the tiled matrix−matrix multiplication (M × N) based on row-major layout, which input matrix will have coalesced accesses?

   a. M

   b. N

   c. Both

   d. Neither

6.8 For the simple reduction kernel, if the block size is 1,024 and warp size is 32, how many warps in a block will have divergence during the fifth iteration?

   a. 0

   b. 1

   c. 16

   d. 32

6.9 For the improved reduction kernel, if the block size is 1,024 and warp size is 32, how many warps will have divergence during the fifth iteration?

   a. 0

   b. 1

   c. 16

   d. 32

6.10 Write a matrix multiplication kernel function that corresponds to the design illustrated in Figure 6.12.

6.11 The following scalar product code tests your understanding of the basic CUDA model. The following code computes 1,024 dot products, each of which is calculated from a pair of 256-element vectors. Assume that the code is executed on G80. Use the code to answer the following questions.

```
1 #define VECTOR_N 1024
2 #define ELEMENT_N 256
3 const int DATA_N    = VECTOR_N * ELEMENT_N;
4 const int DATA_SZ   = DATA_N * sizeof(float);
5 const int RESULT_SZ = VECTOR_N * sizeof(float);
...
6 float *d_A, *d_B, *d_C;
...
7 cudaMalloc((void **)&d_A, DATA_SZ);
8 cudaMalloc((void **)&d_B, DATA_SZ);
9 cudaMalloc((void **)&d_C, RESULT_SZ);
...
10 scalarProd<<<VECTOR_N, ELEMENT_N>>>(d_C, d_A,
d_B, ELEMENT_N);
11
12 __global__ void
13 scalarProd(float *d_C, float *d_A, float *d_B, int
ElementN)
14 {
15 __shared__ float accumResult[ELEMENT_N];
16 //Current vectors bases
17 float *A = d_A + ElementN * blockIdx.x;
18 float *B = d_B + ElementN * blockIdx.x;
19 int tx = threadIdx.x;
20
21 accumResult[tx] = A[tx] * B[tx];
22
23 for(int stride = ElementN /2; stride > 0; stride >> = 1)
24 {
25 __syncthreads();
26 if(tx < stride)
27 accumResult[tx] + = accumResult[stride + tx];
28 }
30 d_C[blockIdx.x] = accumResult[0];
31 }
```

a. How many threads are there in total?

      b. How many threads are there in a warp?

      c. How many threads are there in a block?

      d. How many global memory loads and stores are done for each thread?

      e. How many accesses to shared memory are done for each block?

      f. List the source code lines, if any, that cause shared memory bank conflicts.

      g. How many iterations of the `for` loop (line 23) will have branch divergence? Show your derivation.

      h. Identify an opportunity to significantly reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate?

6.12 In Exercise 4.2, out of the possible range of values for `BLOCK_SIZE`, for what values of `BLOCK_SIZE` will the kernel completely avoid uncoalesced accesses to global memory?

6.13 In an attempt to improve performance, a bright young engineer changed the CUDA code in Figure 6.4 into the following.

```
__shared__ float partialSum[];
unsigned int tid = threadIdx.x;
for (unsigned int stride = n >> 1; stride >= 32; stride
>> = 1) {
  __syncthreads();
  if (tid < stride)
  shared[tid] + = shared[tid + stride];
}
  __syncthreads();
 if (tid < 32) { // unroll last 5 predicated steps
  shared[tid] + = shared[tid + 16];
  shared[tid] + = shared[tid + 8];
  shared[tid] + = shared[tid + 4];
  shared[tid] + = shared[tid + 2];
  shared[tid] + = shared[tid + 1];
}
```

      a. Do you believe that the performance will be improved? Why or why not?

      b. Should the engineer receive a reward or a lecture? Why?

References

CUDA Occupancy Calculator.

CUDA C (2012). *Best Practices Guide*, *v. 4.2*.

Ryoo, S., Rodrigues, C., Stone, S., Baghsorkhi, S., Ueng, S., Stratton, J., & Hwu, W. Program optimization space pruning for a multithreaded GPU, Proceedings of the 6th ACM/IEEE International Symposium on Code Generation and Optimization, April 6–9, 2008.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W. W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2008.

This page intentionally left blank

Floating-Point Considerations

7

CHAPTER OUTLINE

In the early days of computing, floating-point arithmetic capability was found only in mainframes and supercomputers. Although many microprocessors designed in the 1980s started to have floating-point coprocessors, their floating-point arithmetic speed was extremely slow, about three orders of magnitude slower than that of mainframes and supercomputers. With advances in microprocessor technology, many microprocessors designed in the 1990s, such as Intel Pentium III and AMD Athlon, started to have high-performance floating-point capabilities that rival supercomputers. High-speed floating-point arithmetic has become a standard feature for microprocessors and GPUs today. As a result, it has also become important for application programmers to understand and take advantage of floating-point arithmetic in developing their applications. In particular, we will focus on the accuracy of floating-point arithmetic, the precision of floating-point number representation, and how they should be taken into consideration in parallel computing.

7.1 FLOATING-POINT FORMAT

The IEEE-754 Floating-Point Standard is an effort for the computer manufacturers to conform to a common representation and arithmetic behavior for floating-point data [IEEE2008]. Most, if not all, of the computer manufacturers in the world have accepted this standard. In particular, virtually all microprocessors designed in the future will either fully conform to or almost fully conform to the IEEE-754 Floating-Point Standard and its more recent IEEE-754 2008 revision [IEEE2008]. Therefore, it is important for application developers to understand the concept and practical considerations of this standard.

A floating-point number system starts with the representation of a numerical value as bit patterns. In the IEEE-754 Floating-Point Standard, a numerical value is represented in three groups of bits: sign (S), exponent (E), and mantissa (M). With some exceptions that will be detailed later, each (S, E, M) pattern uniquely identifies a numeric value according to the following formula:

$$\text{value} = (-1)^S \times 1.M \times \{2^{E-\text{bias}}\} \tag{7.1}$$

The interpretation of S is simple: $S = 0$ means a positive number and $S = 1$ a negative number. Mathematically, any number, including -1, when raised to the power of 0, results in 1. Thus, the value is positive. On the other hand, when -1 is raised to the power of 1, it is -1 itself. With a multiplication by -1, the value becomes negative. The interpretation of M and E bits are, however, much more complex. We will use the following example to help explain the interpretation of M and E bits.

Assume for the sake of simplicity that each floating-point number consists of a 1-bit sign, 3-bit exponent, and 2-bit mantissa. We will use this hypothetical 6-bit format to illustrate the challenges involved in encoding E and M. As we discuss numeric values, we will sometimes need to express a number either in decimal place value or in binary place value. Numbers expressed in decimal place value will have subscript D and those as binary place value will have subscript B. For example, 0.5_D (5×10^{-1} since the place to the right of the decimal point carries a weight of 10^{-1}) is the same as 0.1_B (1×2^{-1} since the place to the right of the decimal point carries a weight of 2^{-1}).

Normalized Representation of *M*

Equation (7.1) requires that all values are derived by treating the mantissa value as $1.M$, which makes the mantissa bit pattern for each floating-point

number unique. For example, the only one mantissa bit pattern allowed for 0.5_D is the one where all bits that represent M are 0's:

$$0.5_D = 1.0_B \times 2^{-1}$$

Other potential candidates would be $0.1_B \times 2^0$ and $10.0_B \times 2^{-2}$, but neither fits the form of $1.M$. The numbers that satisfy this restriction will be referred to as normalized numbers. Because all mantissa values that satisfy the restriction are of the form 1.XX, we can omit the "1." part from the representation. Therefore, the mantissa value of 0.5 in a 2-bit mantissa representation is 00, which is derived by omitting "1." from 1.00. This makes a 2-bit mantissa effectively a 3-bit mantissa. In general, with IEEE format, an m-bit mantissa is effectively an $(m + 1)$-bit mantissa.

Excess Encoding of *E*

The number of bits used to represent E determines the range of numbers that can be represented. Large positive E values result in very large floating-point absolute values. For example, if the value of E is 64, the floating-point number being represented is between 2^{64} ($> 10^{18}$) and 2^{65}. You would be extremely happy if this was the balance of your savings account! Large negative E values result in very small floating-point values. For example, if the E value is -64, the number being represented is between 2^{-64} ($< 10^{-18}$) and 2^{-63}. This is a very tiny fractional number. The E field allows a floating-point number format to represent a wider range of numbers than integer number formats. We will come back to this point when we look at the representable numbers of a format.

The IEEE standard adopts an excess or biased encoding convention for E. If e bits are used to represent the exponent E, $(2^{e-1} - 1)$ is added to the 2's complement representation for the exponent to form its excess representation. A 2's complement representation is a system where the negative value of a number can be derived by first complementing every bit of the value and add 1 to the result. In our 3-bit exponent representation, there are 3 bits in the exponent ($e = 3$). Therefore, the value $2^{3-1} - 1 = 011$ will be added to the 2's complement representation of the exponent value.

The advantage of excess representation is that an unsigned comparator can be used to compare signed numbers. As shown in Figure 7.1, in our 3-bit exponent representation, the excess-3 bit patterns increase monotonically from -3 to 3 when viewed as unsigned numbers. We will refer to each of these bit patterns as the code for the corresponding value. For

example, the code for -3 is 000 and that for 3 is 110. Thus, if one uses an unsigned number comparator to compare excess-3 code for any number from -3 to 3, the comparator gives the correct comparison result in terms of which number is larger, smaller, etc. For another example, if one compares excess-3 codes 001 and 100 with an unsigned comparator, 001 is smaller than 100. This is the right conclusion since the values that they represent, -2 and 1, have exactly the same relation. This is a desirable property for hardware implementation since unsigned comparators are smaller and faster than signed comparators.

Figure 7.1 also shows that the pattern of all 1's in the excess representation is a reserved pattern. Note that a 0 value and an equal number of positive and negative values results in an odd number of patterns. Having the pattern 111 as either even number or odd number would result in an unbalanced number of even and odd numbers. The IEEE standard uses this special bit pattern in special ways that will be discussed later.

Now we are ready to represent 0.5_D with our 6-bit format:

$$0.5_D = 0\ 010\ 00, \text{ where } S = 0,\ E = 010, \text{ and } M = (1.)00$$

That is, the 6-bit representation for 0.5_D is 001000.

In general, with a normalized mantissa and excess-coded exponent, the value of a number with an n-bit exponent is

$$(-1)^S \times 1.M \times 2^{(E-(2^{(n-1)}-1))}$$

2's complement	Decimal value	Excess-3
101	−3	000
110	−2	001
111	−1	010
000	0	011
001	1	100
010	2	101
011	3	110
100	Reserved pattern	111

FIGURE 7.1

Excess-3 encoding, sorted by excess-3 ordering.

7.2 REPRESENTABLE NUMBERS

The representable numbers of a number format are the numbers that can be exactly represented in the format. For example, if one uses a 3-bit unsigned integer format, the representable numbers are shown in Figure 7.2.

Neither -1 nor 9 can be represented in the format given in Figure 7.2. We can draw a number line to identify all the representable numbers, as shown in Figure 7.3 where all representable numbers of the 3-bit unsigned integer format are marked with stars.

The representable numbers of a floating-point format can be visualized in a similar manner. In Figure 7.4, we show all the representable numbers of what we have so far and two variations. We use a 5-bit format to keep the size of the table manageable. The format consists of 1-bit S, 2-bit E (excess-1 coded), and 2-bit M (with the "1." part omitted). The no-zero column gives the representable numbers of the format we discussed thus far. Readers are encouraged to generate at least part of the no-zero column based on the formula given in Section 7.1. Note that with this format, 0 is not one of the representable numbers.

A quick look at how these representable numbers populate the number line, as shown in Figure 7.5, provides further insights about these representable numbers. In Figure 7.5, we show only the positive

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

FIGURE 7.2

Representable numbers of a 3-bit unsigned integer format.
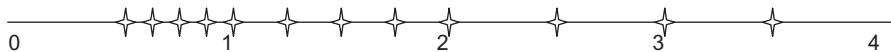
FIGURE 7.3

Representable numbers of a 3-bit unsigned integer format.

E	M	No-zero		Abrupt underflow		Denorm	
		S=0	S=1	S=0	S=1	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	0	0	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^0	$-(2^0)$	2^0	$-(2^0)$	2^0	$-(2^0)$
	01	2^0+1*2^{-2}	$-(2^0+1*2^{-2})$	2^0+1*2^{-2}	$-(2^0+1*2^{-2})$	2^0+1*2^{-2}	$-(2^0+1*2^{-2})$
	10	2^0+2*2^{-2}	$-(2^0+2*2^{-2})$	2^0+2*2^{-2}	$-(2^0+2*2^{-2})$	2^0+2*2^{-2}	$-(2^0+2*2^{-2})$
	11	2^0+3*2^{-2}	$-(2^0+3*2^{-2})$	2^0+3*2^{-2}	$-(2^0+3*2^{-2})$	2^0+3*2^{-2}	$-(2^0+3*2^{-2})$
10	00	2^1	$-(2^1)$	2^1	$-(2^1)$	2^1	$-(2^1)$
	01	2^1+1*2^{-1}	$-(2^1+1*2^{-1})$	2^1+1*2^{-1}	$-(2^1+1*2^{-1})$	2^1+1*2^{-1}	$-(2^1+1*2^{-1})$
	10	2^1+2*2^{-1}	$-(2^1+2*2^{-1})$	2^1+2*2^{-1}	$-(2^1+2*2^{-1})$	2^1+2*2^{-1}	$-(2^1+2*2^{-1})$
	11	2^1+3*2^{-1}	$-(2^1+3*2^{-1})$	2^1+3*2^{-1}	$-(2^1+3*2^{-1})$	2^1+3*2^{-1}	$-(2^1+3*2^{-1})$
11		Reserved pattern					

FIGURE 7.4

Representable numbers of no-zero, abrupt underflow, and denorm formats.

FIGURE 7.5

Representable numbers of the no-zero representation.

representable numbers. The negative numbers are symmetric to their positive counterparts on the other side of 0.

We can make five observations. First, the exponent bits define the major intervals of representable numbers. In Figure 7.5, there are three major intervals on each side of 0 because there are two exponent bits. Basically, the major intervals are between powers of 2's. With 2 bits of exponents and one reserved bit pattern (11), there are three powers of 2 $(2^{-1} = 0.5_D, \ 2^0 = 1.0_D, \ 2^1 = 2.0_D)$, and each starts an interval of representable numbers. Keep in mind that there are also three powers of 2 $(-2^{-1} = -0.5_D, \ -2^0 = -1.0_D, \ -2^1 = -2.0_D)$ to the left of 0 that are not shown in Figure 7.5.

The second observation is that the mantissa bits define the number of representable numbers in each interval. With two mantissa bits, we have four representable numbers in each interval. In general, with N mantissa bits, we have 2^N representable numbers in each interval. If a value to be represented falls within one of the intervals, it will be rounded to one of

these representable numbers. Obviously, the larger the number of representable numbers in each interval, the more precisely we can represent a value in the region. Therefore, the number of mantissa bits determines the *precision* of the representation.

The third observation is that 0 is not representable in this format. It is missing from the representable numbers in the no-zero column of Figure 7.5. Because 0 is one of the most important numbers, not being able to represent 0 in a number representation system is a serious deficiency. We will address this deficiency soon.

The fourth observation is that the representable numbers become closer to each other toward the neighborhood of 0. Each interval is half the size of the previous interval as we move toward 0. In Figure 7.5, the rightmost interval is of width 2, the next one is of width 1, and the next one is of width 0.5. While not shown in Figure 7.5, there are three intervals to the left of 0. They contain the representable negative numbers. The leftmost interval is of width 2, the next one is of width 1, and the next one is of width 0.5. Since every interval has the same representable numbers, four in Figure 7.5, the representable numbers becomes closer to each other as we move toward 0. In other words, the representative numbers become closer as their absolute values become smaller. This is a desirable trend, because as the absolute value of these numbers become smaller, it is more important to represent them more precisely. The distance between representable numbers determines the maximal rounding error for a value that falls into the interval. For example, if you have one billion dollars in your bank account, you may not even notice that there is a 1 dollar rounding error in calculating your balance. However, if the total balance is 10 dollars, having a 1 dollar rounding error would be much more noticeable!

The fifth observation is that, unfortunately, the trend of increasing density of representable numbers and thus increasing precision of representing numbers in the intervals as we move toward 0 does not hold for the very vicinity of 0. That is, there is a gap of representable numbers in the immediate vicinity of 0. This is because the range of normalized mantissa precludes 0. This is another serious deficiency. The representation introduces significantly larger ($4 \times$) errors when representing numbers between 0 and 0.5 compared to the errors for the larger numbers between 0.5 and 1.0. In general, with *m* bits in the mantissa, this style of representation would introduce 2^m times more error in the interval closest to 0 than the next interval. For numerical methods that rely on accurate detection of convergence conditions based on very small data values, such deficiency can cause instability in execution time and accuracy of results.

Furthermore, some algorithms generate small numbers and eventually use them as denominators. The errors in representing these small numbers can be greatly magnified in the division process and cause numerical instability in these algorithms.

One method that can accommodate 0 into a normalized floating-point number system is the *abrupt underflow* convention, which is illustrated in the second column of Figure 7.4. Whenever E is 0, the number is interpreted as 0. In our 5-bit format, this method takes away eight representable numbers (four positive and four negative) in the vicinity of 0 (between -1.0 and $+1.0$) and makes them all 0. Due to its simplicity, some mini-computers in the 1980s used abrupt underflow. Even to this day, some arithmetic units that need to operate in high speed still use abrupt underflow convention. Although this method makes 0 a representable number, it creates an even larger gap between representable numbers in 0's vicinity, as shown in Figure 7.6. It is obvious, when compared with Figure 7.5, that the gap of representable numbers has been enlarged significantly (by $2 \times$) from 0.5 to 1.0. As we explained before, this is very problematic for many numerical algorithms of which the correctness reply on accurate representation of small numbers near 0.

The actual method adopted by the IEEE standard is called *denormalization*. The method relaxes the normalization requirement for numbers very close to 0. As shown later in Figure 7.8, whenever $E = 0$, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0. XX. The value of the exponent is assumed to be the same as the previous interval. For example, in Figure 7.4, the denormalized representation 00001 has exponent value 00 and mantissa value 01. The mantissa is assumed to be 0.01 and the exponent value is assumed to be the same as that of the previous interval: 0 rather than -1. That is, the value that
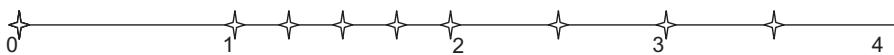

FIGURE 7.6

Representable numbers of the abrupt underflow format.
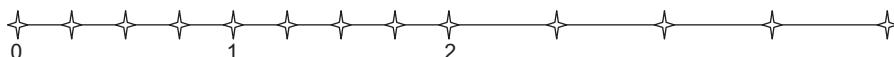
FIGURE 7.7

Representable numbers of a denormalization format.

00001 represents is now $0.01 \times 2^0 = 2^{-2}$. Figure 7.7 shows the representable numbers for the denormalized format. The representation now has uniformly spaced representable numbers in the close vicinity of 0. Intuitively, the denormalized convention takes the four numbers in the last interval of representable numbers of a no-zero representation and spreads them out to cover the gap area. This eliminates the undesirable gap in the previous two methods. Note that the distances between representable numbers in the last two intervals are actually identical. In general, if the *n*-bit exponent is 0, the value is

$$0.M \times 2^{-2\hat{}(n+1)+2}$$

As we can see, the denormalization formula is quite complex. The hardware also needs to be able to detect whether a number falls into the denormalized interval and choose the appropriate representation for that number. The amount of hardware required to implement denormalization in high speed is quite significant. Implementations that use a moderate amount of hardware often introduce thousands of clock cycles of delay whenever a denormalized number needs to be generated or used. This was the reason why early generations of CUDA devices did not support denormalization. However, virtually all recent generations of CUDA devices, thanks to the increasing number of available transistors of more recent fabrication processes, support denormalization. More specifically, all CUDA devices of compute capability 1.3 and later support denormalized double-precision operands, and all devices of compute capability 2.0 and later support denormalized single-precision operands.

In summary, the precision of a floating-point representation is measured by the maximal error that we can introduce to a floating-point number by representing that number as one of the representable numbers. The smaller the error is, the higher the precision. The precision of a floating-point representation can be improved by adding more bits to mantissa. Adding 1 bit to the representation of the mantissa improves the

exponent	mantissa	meaning
11…1	$\neq 0$	NaN
11…1	$= 0$	$(-1)^S *_\infty$
00…0	$\neq 0$	denor malized
00…0	$= 0$	0

FIGURE 7.8

Special bit patterns in the IEEE standard format.

precision by reducing the maximal error by half. Thus, a number system has higher precision when it uses more bits for mantissa. This is reflected in double-precision versus single-precision numbers in the IEEE standard.

7.3 SPECIAL BIT PATTERNS AND PRECISION IN IEEE FORMAT

We now turn to more specific details of the actual IEEE format. When all exponent bits are 1's, the number represented is an infinity value if the mantissa is 0. It is a not a number (NaN) if the mantissa is not 0. All special bit patterns of the IEEE floating-point format are described in Figure 7.8.

All other numbers are normalized floating-point numbers. Single-precision numbers have 1-bit S, 8-bit E, and 23-bit M. Double-precision numbers have 1-bit S, 11-bit E, and 52-bit M. Since a double-precision number has 29 more bits for mantissa, the largest error for representing a number is reduced to $1/2^{29}$ of that of the single-precision format! With the additional 3 bits of exponent, the double-precision format also extends the number of intervals of representable numbers. This extends the range of representable numbers to very large as well as very small values.

All representable numbers fall between $-\infty$ (negative infinity) and $+\infty$ (positive infinity). An ∞ can be created by overflow, for example, a large number divided by a very small number. Any representable number divided by $+\infty$ or $-\infty$ results in 0.

NaN is generated by operations of which the input values do not make sense, for example, $0/0$, $0 \times \infty$, ∞/∞, $\infty - \infty$. They are also used for data that has not been properly initialized in a program. There are two types of NaNs in the IEEE standard: signaling and quiet. Signaling NaNs (sNaNs) should be represented with the most significant mantissa bit cleared, whereas quiet NaNs (qNaNs) are represented with the most significant mantissa bit set.

An sNaN causes an exception when used as input to arithmetic operations. For example, the operation $(1.0 + sNaN)$ raises an exception signal to the operating system. Signaling NaNs are used in situations where the programmer would like to make sure that the program execution be interrupted whenever any NaN values are used in floating-point computations. These situations usually mean that there is something wrong with the execution of the program. In mission-critical applications, the execution cannot continue until the validity of the execution can be verified with a

separate means. For example, software engineers often mark all the uninitialized data as sNaN. This practice ensures the detection of using uninitialized data during program execution. The current generation of GPU hardware does not support sNaN. This is due to the difficulty of supporting accurate signaling during massively parallel execution.

A qNaN generates another qNaN without causing an exception when used as input to arithmetic operations. For example, the operation (1.0 + qNaN) generates a qNaN. Quiet NaNs are typically used in applications where the user can review the output and decide if the application should be rerun with a different input for more valid results. When the results are printed, qNaNs are printed as "NaN" so that the user can spot them in the output file easily.

7.4 ARITHMETIC ACCURACY AND ROUNDING

Now that we have a good understanding of the IEEE floating-point format, we are ready to discuss the concept of arithmetic accuracy. While the precision is determined by the number of mantissa bits used in a floating-point number format, the accuracy is determined by the operations performed on a floating number. The accuracy of a floating-point arithmetic operation is measured by the maximal error introduced by the operation. The smaller the error is, the higher the accuracy. The most common source of error in floating-point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding. Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly. For example, a multiplication generates a product value that consists of twice the number of bits than either of the input values. For another example, adding two floating-point numbers can be done by adding their mantissa values together if the two floating-point values are identical exponents. When two input operands to a floating-point addition have different exponents, the mantissa of the one with the smaller exponent is repeatedly divided by 2 or right-shifted (i.e., all the mantissa bits are shifted to the right by 1 bit position) until the exponents are equal. As a result, the final result can have more bits than the format can accommodate.

Alignment shifting of operands can be illustrated with a simple example based on the 5-bit representation in Figure 7.4. Assume that we need to add $1.00_B - 2^{-2}(0, 00, 01)$ to $1.00 \times 2^1{}_D (0, 10, 00)$; that is, we need to perform $1.00_B \times 2^1 + 1.00_B \times 2^{-2}$. Due to the difference in exponent

values, the mantissa value of the second number needs to be right-shifted by 3 bit positions before it is added to the first mantissa value. That is, the addition becomes $1.00_B \times 2^1 + 0.001_B \times 2^1$. The addition can now be performed by adding the mantissa values together. The ideal result would be $1.001_B \times 2^1$. However, we can see that this ideal result is not a representable number in a 5-bit representation. It would have required three mantissa bits and there are only two mantissa bits in the format. Thus, the best one can do is to generate one of the closest representable numbers, which is either $1.01_B \times 2^1$ or $1.00_B \times 2^1$. By doing so, we introduce an error, $0.001_B \times 2^1$, which is half the place value of the least significant place. We refer to this as 0.5_D ULP (units in the last place). If the hardware is designed to perform arithmetic and rounding operations perfectly, the most errors that one should introduce should be no more than 0.5_D ULP. To our knowledge, this is the accuracy achieved by the addition and subtraction operations in all CUDA devices today.

In practice, some of the more complex arithmetic hardware units, such as division and transcendental functions, are typically implemented with polynomial approximation algorithms. If the hardware does not use a sufficient number of terms in the approximation, the result may have an error larger than 0.5_D ULP. For example, if the ideal result of an inversion operation is $1.00_B \times 2^1$ but the hardware generates a $1.10_B \times 2^1$ due to the use of an approximation algorithm, the error is 2_D ULP since the error $(1.10_B - 1.00_B = 0.10_B)$ is two times bigger than the units in the last place (0.01_B). In practice, the hardware inversion operations in some early devices introduce an error that is twice the place value of the least place of the mantissa, or 2 ULP. Thanks to the more abundant transistors in more recent generations of CUDA devices, their hardware arithmetic operations are much more accurate.

7.5 ALGORITHM CONSIDERATIONS

Numerical algorithms often need to sum up a large number of values. For example, the dot product in matrix multiplication needs to sum up pairwise products of input matrix elements. Ideally, the order of summing these values should not affect the final total since addition is an associative operation. However, with finite precision, the order of summing these values can affect the accuracy of the final result. For example, if we need to perform a sum reduction on four numbers in our 5-bit representation: $1.00_B \times 2^0 + 1.00_B \times 2^0 + 1.00_B \times 2^{-2} + 1.00_B \times 2^{-2}$.

If we add up the numbers in strict sequential order, we have the following sequence of operations:

$$1.00_B \times 2^0 + 1.00_B \times 2^0 + 1.00_B \times 2^{-2} + 1.00_B \times 2^{-2} = 1.00_B \times 2^1$$
$$+ \ 1.00_B \times 2^{-2} + 1.00_B \times 2^{-2} = 1.00_B \times 2^1 + 1.00_B \times 2^{-2} = 1.00_B \times 2^1$$

Note that in the second and third step, the smaller operand simply disappears because they are too small compared to the larger operand.

Now, let's consider a parallel algorithm where the first two values are added and the second two operands are added in parallel. The algorithm then adds up the pairwise sum:

$$(1.00_B \times 2^0 + 1.00_B \times 2^0) + (1.00_B \times 2^{-2} + 1.00_B \times 2^{-2}) = 1.00_B \times 2^1$$
$$+ \ 1.00_B \times 2^{-1} = 1.01_B \times 2^1$$

Note that the results are different from the sequential result! This is because the sum of the third and fourth values is large enough that it now affects the addition result. This discrepancy between sequential algorithms and parallel algorithms often surprises application developers who are not familiar with floating-point precision and accuracy considerations. Although we showed a scenario where a parallel algorithm produced a more accurate result than a sequential algorithm, readers should be able to come up with a slightly different scenario where the parallel algorithm produces a less accurate result than a sequential algorithm. Experienced application developers either make sure that the variation in the final result can be tolerated, or ensure that the data is sorted or grouped in a way that the parallel algorithm results in the most accurate results.

A common technique to maximize floating-point arithmetic accuracy is to presort data before a reduction computation. In our sum reduction example, if we presort the data according to ascending numerical order, we will have the following:

$$1.00_B \times 2^{-2} + 1.00_B \times 2^{-2} + 1.00_B \times 2^0 + 1.00_B \times 2^0$$

When we divide up the numbers into groups in a parallel algorithm, say the first pair in one group and the second pair in another group, numbers with numerical values close to each other are in the same group. Obviously, the sign of the numbers needs to be taken into account during the presorting process. Therefore, when we perform addition in these groups, we will likely have accurate results. Furthermore, some parallel algorithms use each thread to sequentially reduce values within each

group. Having the numbers sorted in ascending order allows a sequential addition to get higher accuracy. This is a reason why sorting is frequently used in massively parallel numerical algorithms. Interested readers should study more advanced techniques such as compensated summation algorithm, also known as Kahan's summation algorithm, for getting an even more robust approach to accurate summation of floating-point values [Kahan1965].

7.6 NUMERICAL STABILITY

While the order of operations may cause variation in the numerical outcome of reduction operations, it may have even more serious implications on some types of computation such as solvers for linear systems of equations. In these solvers, different numerical values of input may require different ordering of operations to find a solution. If an algorithm fails to follow a desired order of operations for an input, it may fail to find a solution even though the solution exists. Algorithms that can always find an appropriate operation order and thus find a solution to the problem as long as it exists for any given input values are called *numerically stable*. Algorithms that fall short are referred to as *numerically unstable*.

In some cases, numerical stability considerations can make it more difficult to find efficient parallel algorithms for a computational problem. We can illustrate this phenomenon with a solver that is based on Gaussian elimination. Consider the following system of linear equations:

$$3X + 5Y + 2Z = 19 \qquad \text{(equation 1)}$$

$$2X + 3Y + Z = 11 \qquad \text{(equation 2)}$$

$$X + 2Y + 2Z = 11 \qquad \text{(equation 3)}$$

As long as the three planes represented by these equations have an intersection point, we can use Gaussian elimination to derive the solution that gives the coordinate of the intersection point. We show the process of applying Gaussian elimination to this system in Figure 7.9, where variables are systematically eliminated from lower positioned equations.

In the first step, all equations are divided by their coefficient for the *X* variable: 3 for equation 1, 2 for equation 2, and 1 for equation 3. This makes the coefficients for *X* in all equations the same. In step two, equation 1 is

subtracted from equations 2 and 3. These subtractions eliminate variable X from equations 2 and 3, as shown in Figure 7.9.

We can now treat equations 2 and 3 as a smaller system of equations with one fewer variable than the original equation. Since they do not have variable X, they can be solved independently from equation 1. We can make more progress by eliminating variable Y from equation 3. This is done in step 3 by dividing equations 2 and 3 by the coefficients for their Y variables: $-1/6$ for equation 2 and $1/3$ for equation 3. This makes the coefficients for Y in both equations 2 and 3 the same. In step four, equation 2 is subtracted from equation 3, which eliminates variable Y from equation 3.

For systems with a larger number of equations, the process would be repeated more. However, since we have only three variables in this example, the equation 3 has only the Z variable. We simply need to divide

$$
\begin{array}{llll}
3X & +5Y & +2Z & = 19 \\
2X & +3Y & +Z & = 11 \\
X & +2Y & +2Z & = 11
\end{array}
$$

Original

$$
\begin{array}{llll}
X & +5/3Y & +2/3Z & = 19/3 \\
X & +3/2Y & +1/2Z & = 11/2 \\
X & +2Y & +2Z & = 11
\end{array}
$$

Step 1: divide equation 1 by 3, equation 2 by 2

$$
\begin{array}{llll}
X & +5/3Y & +2/3Z & = 19/3 \\
& -1/6Y & -1/6Z & = -5/6 \\
& 1/3Y & +4/3Z & = 14/3
\end{array}
$$

Step 2: subtract equation 1 from equation 2 and equation 3

$$
\begin{array}{llll}
X & +5/3Y & +2/3Z & = 19/3 \\
& Y & +Z & = 5 \\
& Y & +4Z & = 14
\end{array}
$$

Step 3: divide equation 2 by -1/6 and equation 3 by 1/3

$$
\begin{array}{llll}
X & +5/3Y & +2/3Z & = 19/3 \\
& Y & +Z & = 5 \\
& & +3Z & = 9
\end{array}
$$

Step 4: subtract equation 2 from equation 3

$$
\begin{array}{llll}
X & +5/3Y & +2/3Z & = 19/3 \\
& Y & +Z & = 5 \\
& & Z & = 3
\end{array}
$$

Step 5 : divide equation 3 by 3
Solution for Z!

$$
\begin{array}{llll}
X & +5/3Y & +2/3Z & = 19/3 \\
& Y & & = 2 \\
& & Z & = 3
\end{array}
$$

Step 6: substitute Z solution into equation 2. Solution for Y!

$$
\begin{array}{llll}
X & & & = 1 \\
& Y & & = 2 \\
& & Z & = 3
\end{array}
$$

Step 7: substitute Y and Z into equation 1. Solution for X!

FIGURE 7.9

Gaussian elimination and backward substitution for solving systems of linear equations.

equation 3 by the coefficient for variable *Z*. This conveniently gives us the solution *Z* = 3.

With the solution for the *Z* variable in hand, we can substitute the *Z* value into equation 2 to get the solution *Y* = 2. We can then substitute both *Z* = 3 and *Y* = 2 into equation 1 to get the solution *X* = 1. We now have the complete solution for the original system. It should be obvious why steps six and seven form the second phase of the method called *backward substitution*. We go backwards from the last equation to the first equation to get solutions for more and more variables.

In general, the equations are stored in matrix forms in computers. Since all calculations only involve the coefficients and the right-side values, we can just store these coefficients and right-side values in a matrix. Figure 7.10 shows the matrix view of the Gaussian elimination and back substitution process. Each row of the matrix corresponds to an original equation. Operations on equations become operations on matrix rows.

FIGURE 7.10

Gaussian elimination and backward substitution in matrix view.

After Gaussian elimination, the matrix becomes a triangular matrix. This is a very popular type of matrix for various physics and mathematics reasons. We see that the end goal is to make the coefficient part of the matrix into a diagonal form, where each row has only a value 1 on the diagonal line. This is called an *identity matrix* because the result of multiplying any matrix multiplied by an identity matrix is itself. This is also the reason why performing Gaussian elimination on a matrix is equivalent to multiplying the matrix by its inverse matrix.

In general, it is straightforward to design a parallel algorithm for the Gaussian elimination procedure that we described in Figure 7.10. For example, we can write a CUDA kernel and designate each thread to perform all calculations to be done on a row of the matrix. For systems that can fit into shared memory, we can use a thread block to perform Gaussian elimination. All threads iterate through the steps. After each division step, all threads participate in barrier synchronization. They then all perform a subtraction step, after which one thread will stop its participation since its designated row has no more work to do until the back substitution phase. After the subtraction step, all threads need to perform barrier synchronization again to ensure that the next step will be done with the updated information. With systems of equations with many variables, we can expect a reasonable amount of speedup from the parallel execution.

Unfortunately, the simple Gaussian elimination algorithm we have been using can suffer from numerical instability. This can be illustrated with the following example.

$$5Y + 2Z = 16 \qquad \text{(equation 1)}$$

$$2X + 3Y + Z = 11 \qquad \text{(equation 2)}$$

$$X + 2Y + 2Z = 11 \qquad \text{(equation 3)}$$

We will encounter a problem when we perform step one of the algorithm. The coefficient for the X variable in equation 1 is 0. We will not be able to divide equation 1 by the coefficient for variable X and eliminate the X variable from equations 2 and 3 by subtracting equation 1 from equations 2 and 3. Readers should verify that this system of equation is solvable and has the same solution $X = 1$, $Y = 2$, and $Z = 3$. Therefore, the algorithm is numerically unstable. It can fail to generate a solution for certain input values even though the solution exists.

This is a well-known problem with Gaussian elimination algorithms and can be addressed with a method commonly referred to as *pivoting*.

The idea is to find one of the remaining equations of which the coefficient for the lead variable is not 0. By swapping the current top equation with the identified equation, the algorithm can successfully eliminate the lead variable from the rest of the equations. If we apply pivoting to the three equations, we end up with the following set.

$$2X + 3Y + Z = 11 \qquad \text{(equation 1 ', original equation 2)}$$

$$5Y + 2Z = 16 \qquad \text{(equation 2 ', original equation 1)}$$

$$X + 2Y + 2Z = 11 \qquad \text{(equation 3 ', original equation 3)}$$

Note that the coefficient for X in equation 1' is no longer 0. We can proceed with Gaussian elimination, as illustrated in Figure 7.11.

Readers should follow the steps in Figure 7.11. The most important additional insight is that some equations may not have the variable that the algorithm is eliminating at the current step (see row 2 of step one in

FIGURE 7.11

Gaussian elimination with pivoting.

Figure 7.11). The designated thread does not need to do the division on the equation.

In general, the pivoting step should choose the equation with the largest absolute coefficient value among all the lead variables and swap its equation (row) with the current top equation, as well as swap the variable (column) with the current variable. While pivoting is conceptually simple, it can incur significant implementation complexity and performance overhead. In the case of our simple CUDA kernel implementation, recall that each thread is assigned a row. Pivoting requires an inspection and perhaps swapping of coefficient data spread across these threads. This is not a big problem if all coefficients are in the shared memory. We can run a parallel reduction using threads in the block as long as we control the level of control flow divergence within warps.

However, if the system of linear equations is being solved by multiple thread blocks or even multiple nodes of a compute cluster, the idea of inspecting data spread across multiple thread blocks or multiple compute cluster nodes can be an extremely expensive proposition. This is the main motivation for *communication-avoiding algorithms* that avoid a global inspection of data such as pivoting [Ballard2011]. In general, there are two approaches to this problem. Partial pivoting restricts the candidates of the swap operation to come from a localized set of equations so that the cost of global inspection is limited. This can, however, slightly reduce the numerical accuracy of the solution. Researchers have also demonstrated that randomization tends to maintain a high level of numerical accuracy for the solution.

7.7 SUMMARY

This chapter introduced the concepts of floating-point format and representable numbers that are foundational to the understanding of precision. Based on these concepts, we also explained the denormalized numbers and why they are important in many numerical applications. In early CUDA devices, denormalized numbers were not supported. However, later hardware generations support denormalized numbers. We have also explained the concept of arithmetic accuracy of floating-point operations. This is important for CUDA programmers to understand the potential lower accuracy of fast arithmetic operations implemented in the special function units. More importantly, readers should now have a good understanding of why parallel algorithms often can affect the accuracy of

calculation results and how one can potentially use sorting and other techniques to improve the accuracy of their computation.

7.8 EXERCISES

7.1. Draw the equivalent of Figure 7.5 for a 6-bit format (1-bit sign, 3-bit mantissa, 2-bit exponent). Use your result to explain what each additional mantissa bit does to the set of representable numbers on the number line.

7.2. Draw the equivalent of Figure 7.5 for another 6-bit format (1-bit sign, 2-bit mantissa, 3-bit exponent). Use your result to explain what each additional exponent bit does to the set of representable numbers on the number line.

7.3. Assume that in a new processor design, due to technical difficulty, the floating-point arithmetic unit that performs addition can only do "round to zero" (rounding by truncating the value toward 0). The hardware maintains a sufficient number of bits that the only error introduced is due to rounding. What is the maximal ulp error value for add operations on this machine?

7.4. A graduate student wrote a CUDA kernel to reduce a large floating-point array to the sum of all its elements. The array will always be sorted with the smallest values to the largest values. To avoid branch divergence, he decided to implement the algorithm of Figure 6.4. Explain why this can reduce the accuracy of his results.

7.5. Assume that in a arithmetic unit design, the hardware implements an iterative approximation algorithm that generates two additional accurate mantissa bits of the result for the sin() function in each clock cycle. The architect decided to allow the arithmetic function to iterate nine clock cycles. Assume that the hardware fill in all remaining mantissa bits with zeroes. What would be the maximal ulp error of the hardware implementation of the sin() function in this design for the IEEE single-precision numbers? Assume that the omitted 1. mantissa bit must also be generated by the arithmetic unit.

References

Ballard, G., Demmel, J., Holtz, O., & Schwartz, O. (2011). Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, *32* (3), 866−901.

IEEE Microprocessor Standards Committee. Draft standard for floating-point arithmetic P754. January 2008.

Kahan, W. (1965). Further remarks on reducing truncation errors. *Communications of the ACM*, *8*(1), 40. doi:10.1145/363707.363723.

This page intentionally left blank

Parallel Patterns: Convolution

With an Introduction to Constant Memory and Caches

In the next several chapters, we will discuss a set of important parallel computation patterns. These patterns are the basis of many parallel algorithms that appear in 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. For example, Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images. Other filters smooth out the signal values so that one can see the big-picture trend. They also form the basis of a large number of force and energy calculation algorithms used in simulation models. Convolution typically involves a significant number of arithmetic operations on each data element. For large data sets such as high-definition images and videos, the amount of computation can be very large. Each output data element can be calculated independently of each other, a desirable trait for massively parallel computing. On the other hand, there is a substantial level of input data sharing among output data

elements with somewhat challenging boundary conditions. This makes convolution an important use case of sophisticated tiling methods and input data staging methods.

8.1 BACKGROUND

Mathematically, convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements. The weights used in the weighted sum calculation are defined by an input mask 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 mask arrays as *convolution masks* to avoid confusion. The same convolution mask is typically used for all elements of the array.

In audio digital signal processing, the input data are in 1D form and represent signal volume as a function of time. Figure 8.1 shows a convolution example for 1D data where a five-element convolution mask array M is applied to a seven-element input array N. We will follow the C language convention where N and P elements are indexed from 0 to 6 and M elements are indexed from 0 to 4. The fact that we use a five-element mask M means that each P element is generated by a weighted sum of the corresponding N element, up to two elements to the left and up to two elements to the right. For example, the value of P[2] is generated as the weighted sum of N[0] (N[2-2]) through N[4] (N[2+2]). In this example, we arbitrarily assume that the values of the N elements are 1, 2, 3, ..., 7. The M elements define the weights, the values of which are 3, 4, 5, 4, and 3 in this example. Each weight value is multiplied to the corresponding N
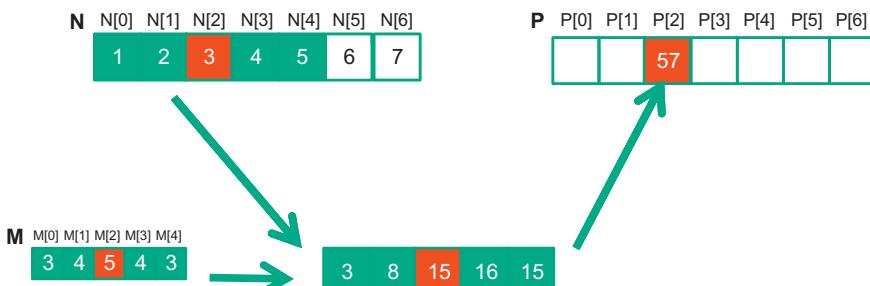
FIGURE 8.1

A 1D convolution example, inside elements.

element values before the products are summed together. As shown in Figure 8.1, the calculation for `P[2]` is as follows:

```
P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]
     = 1*3 + 2*4 + 3*5 + 4*4 + 5*3
     = 57
```

In general, the size of the mask tends to be an odd number, which makes the weighted sum calculation symmetric around the element being calculated. That is, an odd number of mask elements define the weight to be applied to the input element in the corresponding position of the output element along with the same number of input elements on each side of that position. In Figure 8.1, with a mask size of five elements, each output element is calculated as the weighted sum of the corresponding input element, two elements on the left and two elements on the right. For example, `P[2]` is calculated as the weighted sum of `N[2]` along with `N[0]` and `N[1]` on the left and `N[3]` and `N[4]` on the right.

In Figure 8.1, the calculation for `P` element `P[i]` can be viewed as an inner product between the subarray of `N` that starts at `N[i-2]` and the `M` array. Figure 8.2 shows the calculation for `P[3]`. The calculation is shifted by one `N` element from that of Figure 8.1. That is the value of `P[3]` is the weighted sum of `N[1]` (`N[3-2]`) through `N[5]` (`3 + 2`). We can think of the calculation for `P[3]` as follows:

```
P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]
     = 2*3 + 3*4 + 4*5 + 5*4 + 6*3
     = 76
```

Because convolution is defined in terms of neighboring elements, boundary conditions naturally exist for output elements that are close to the ends of an array. As shown in Figure 8.3, when we calculate `P[1]`, there is only one `N` element to the left of `N[1]`. That is, there are not
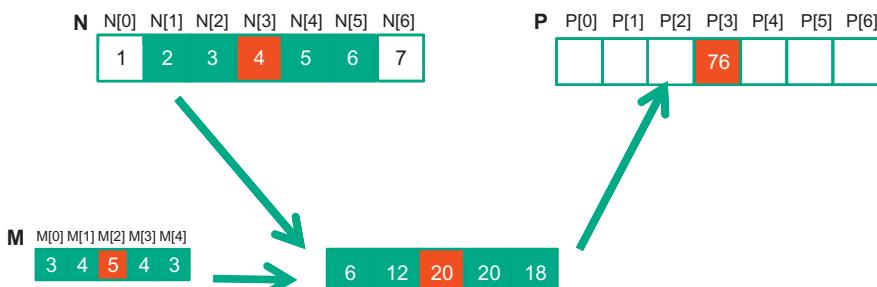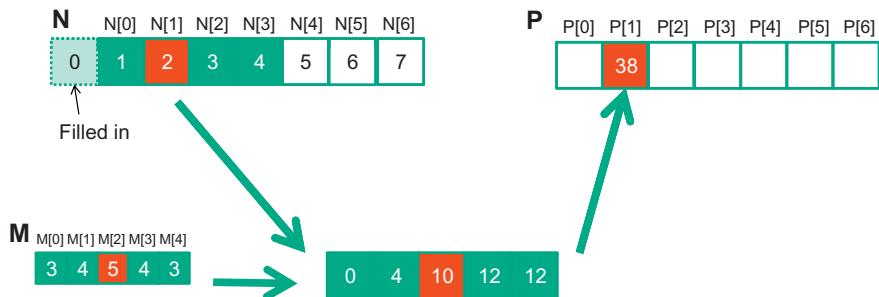
FIGURE 8.2

1D convolution, calculation of `P[3]`.

FIGURE 8.3
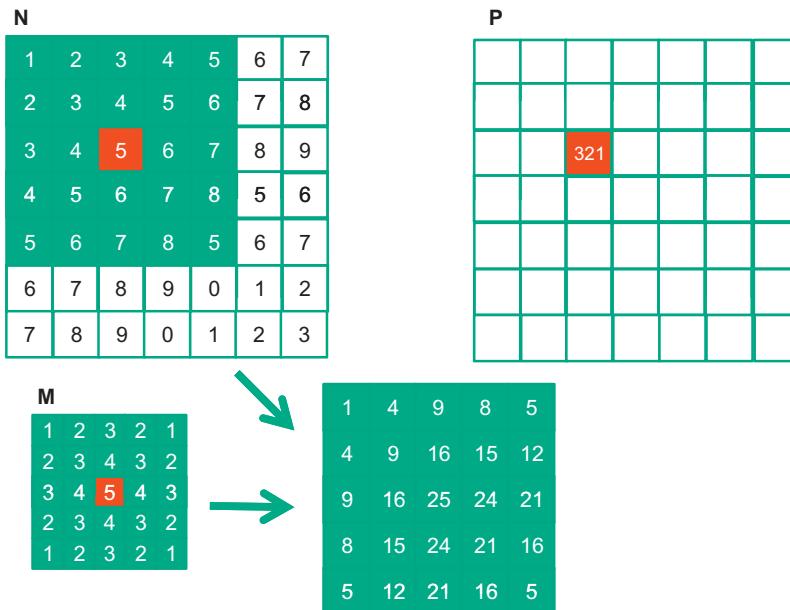
1D convolution boundary condition.

enough N elements to calculate P[1] according to our definition of convolution. A typical approach to handling such a boundary condition is to define a default value to these missing N elements. For most applications, the default value is 0, which is what we use in Figure 8.3. For example, in audio signal processing, we can assume that the signal volume is 0 before the recording starts and after it ends. In this case, the calculation of P[1] is as follows:

```
P[1] = 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4]
     = 0 * 3 + 1*4 + 2*5 + 3*4 + 4*3
     = 38
```

The N element that does not exist in this calculation is illustrated as a dashed box in Figure 8.3. It should be clear that the calculation of P[0] will involve two missing N elements, both of which will be assumed to be 0 for this example. We leave the calculation of P[0] as an exercise. These missing elements are typically referred to as *ghost elements* in literature. There are also other types of ghost elements due to the use of tiling in parallel computation. These ghost elements can have significant impact on the complexity and/or efficiency of tiling. We will come back to this point soon.

For image processing and computer vision, input data is usually in 2D form, with pixels in an *x-y* space. Image convolutions are also two dimensional, as illustrated in Figure 8.4. In a 2D convolution, the mask M is a 2D array. Its *x* and *y* dimensions determine the range of neighbors to be included in the weighted sum calculation. In Figure 8.4, we use a 5×5 mask for simplicity. In general, the mask does not have to be a square array. To generate an output element, we take the subarray of which the center is at the corresponding location in the input array N. We then

FIGURE 8.4

A 2D convolution example.
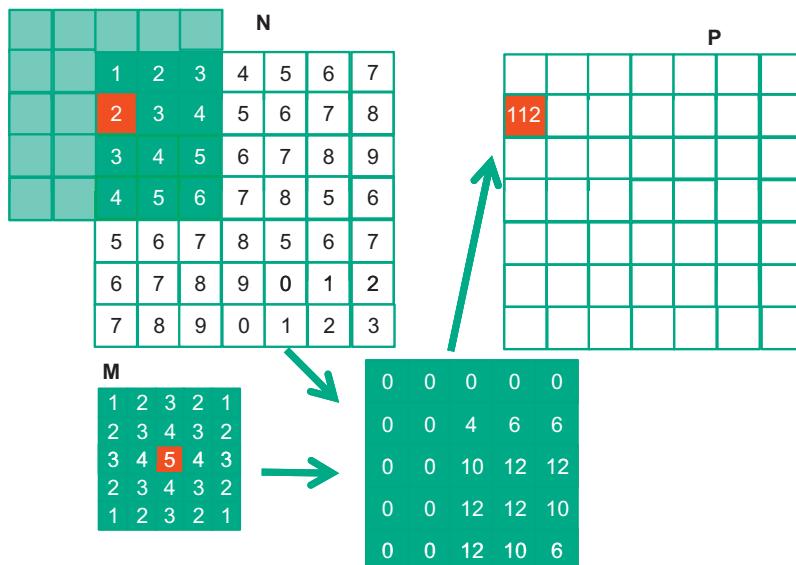
perform pairwise multiplication between elements of the mask array and those of the mask array. For our example, the result is shown as the 5×5 product array below N and P arrays in Figure 8.4. The value of the output element is the sum of all elements of the product array.

The example in Figure 8.4 shows the calculation of $P_{2,2}$. For brevity, we will use $N_{y,x}$ to denote N[y][x] in addressing a C array. Since N and P are most likely dynamically allocated arrays, we will be using linearized indices in our actual code examples. The subarray of N for calculating the value of $P_{2,2}$ span from $N_{0,0}$ to $N_{0,4}$ in the x or horizontal direction and $N_{0,0}$ to $N_{4,0}$ in the y or vertical direction. The calculation is as follows:

```
P2,2 = N0,0*M0,0 + N0,1*M0,1 + N0,2*M0,2 + N0,3*M0,3 + N0,4*M0,4
     + N1,0*M1,0 + N1,1*M1,1 + N1,2*M1,2 + N1,3*M1,3 + N1,4*M1,4
     + N2,0*M2,0 + N2,1*M2,1 + N2,2*M2,2 + N2,3*M2,3 + N2,4*M2,4
     + N3,0*M3,0 + N3,1*M3,1 + N3,2*M3,2 + N3,3*M3,3 + N3,4*M3,4
     + N4,0*M4,0 + N4,1*M4,1 + N4,2*M4,2 + N4,3*M4,3 + N4,4*M4,4
   = 1*1 + 2*2 + 3*3 + 4*2 + 5*1
     + 2*2 + 3*3 + 4*4 + 5*3 + 6*2
     + 3*3 + 4*4 + 5*5 + 6*4 + 7*3
     + 4*2 + 5*3 + 6*4 + 7*3 + 8*2
```

```
        + 5*1 + 6*2 + 7*3 + 8*2 + 5*1
    =  1 + 4 + 9 + 8 + 5
        + 4 + 9 + 16 + 15 + 12
        + 9 + 16 + 25 + 24 + 21
        + 8 + 15 + 24 + 21 + 16
        + 5 + 12 + 21 + 16 + 5
    = 321
```

Like 1D convolution, 2D convolution must also deal with boundary conditions. With boundaries in both the *x* and *y* dimensions, there are more complex boundary conditions: the calculation of an output element may involve boundary conditions along a horizontal boundary, a vertical boundary, or both. Figure 8.5 illustrates the calculation of a P element that involves both boundaries. From Figure 8.5, the calculation of $P_{1,0}$ involves two missing columns and one missing horizontal row in the subarray of N. Like in 1D convolution, different applications assume different default values for these missing N elements. In our example, we assume that the default value is 0. These boundary conditions also affect the efficiency of tiling. We will come back to this point soon.

FIGURE 8.5

A 2D convolution boundary condition.

8.2 1D PARALLEL CONVOLUTION—A BASIC ALGORITHM

As we mentioned in Section 8.1, the calculation of all output (P) elements can be done in parallel in a convolution. This makes convolution an ideal problem for parallel computing. Based on our experience in matrix−matrix multiplication, we can quickly write a simple parallel convolution kernel. For simplicity, we will work on 1D convolution.

The first step is to define the major input parameters for the kernel. We assume that the 1D convolution kernel receives five arguments: pointer to input array N, pointer to input mask M, pointer to output array P, size of the mask Mask_Width, and size of the input and output arrays Width. Thus, we have the following set up:

```
__global__ void convolution_1D_basic_kernel(float *N, float
 *M, float *P,
int Mask_Width, int Width) {
// kernel body
}
```

The second step is to determine and implement the mapping of threads to output elements. Since the output array is one dimensional, a simple and good approach is to organize the threads into a 1D grid and have each thread in the grid calculate one output element. Readers should recognize that this is the same arrangement as the vector addition example as far as output elements are concerned. Therefore, we can use the following statement to calculate an output element index from the block index, block dimension, and thread index for each thread:

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Once we determined the output element index, we can access the input N elements and the mask M elements using offsets to the output element index. For simplicity, we assume that Mask_Width is an odd number and the convolution is symmetric, that is, Mask_Width is 2*n + 1 where n is an integer. The calculation of P[i] will use N[i-n], N[i-n + 1],..., N[i-1], N[i], N[i + 1], ..., N[i + n-1], N[i + n]. We can use a simple loop to do this calculation in the kernel:

```
float Pvalue = 0;
int N_start_point = i - (Mask_Width/2);
for (int j = 0; j < Mask_Width; j++) {
  if (N_start_point + j >= 0 && N_start_point + j < Width) {
     Pvalue + = N[N_start_point + j]*M[j];
  }
}
P[i] = Pvalue;
```

The variable `Pvalue` will allow all intermediate results to be accumulated in a register to save DRAM bandwidth. The `for` loop accumulates all the contributions from the neighboring elements to the output `P` element. The `if` statement in the loop tests if any of the input `N` elements used are ghost elements, either on the left side or the right side of the `N` array. Since we assume that 0 values will be used for ghost elements, we can simply skip the multiplication and accumulation of the ghost element and its corresponding `N` element. After the end of the loop, we release the `Pvalue` into the output `P` element. We now have a simple kernel in Figure 8.6.

We can make two observations about the kernel in Figure 8.6. First, there will be control flow divergence. The threads that calculate the output `P` elements near the left end or the right end of the `P` array will handle ghost elements. As we showed in Section 8.1, each of these neighboring threads will encounter a different number of ghost elements. Therefore, they will all be somewhat different decisions in the `if` statement. The thread that calculates `P[0]` will skip the multiply-accumulate statement about half of the time, whereas the one that calculates `P[1]` will skip one fewer times, and so on. The cost of control divergence will depend on `Width`, the size of the input array, and `Mask_Width` (the size of the masks). For large input arrays and small masks, the control divergence only occurs to a small portion of the output elements, which will keep the effect of control divergence small. Since convolution is often applied to large images and spatial data, we typically expect that the effect of convergence will be modest or insignificant.

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

FIGURE 8.6

A 1D convolution kernel with boundary condition handling.

A more serious problem is memory bandwidth. The ratio of floating-point arithmetic calculation to global memory accesses is only about 1.0 in the kernel. As we have seen in the matrix−matrix multiplication example, this simple kernel can only be expected to run at a small fraction of the peak performance. We will discuss two key techniques for reducing the number of global memory accesses in the next two sections.

8.3 CONSTANT MEMORY AND CACHING

We can make three interesting observations about the way the mask array M is used in convolution. First, the size of the M array is typically small. Most convolution masks are less than 10 elements in each dimension. Even in the case of a 3D convolution, the mask typically contains only less than 1,000 elements. Second, the contents of M are not changed throughout the execution of the kernel. Third, all threads need to access the mask elements. Even better, all threads access the M elements in the same order, starting from M[0] and move by one element a time through the iterations of the for loop in Figure 8.6. These two properties make the mask array an excellent candidate for constant memory and caching.
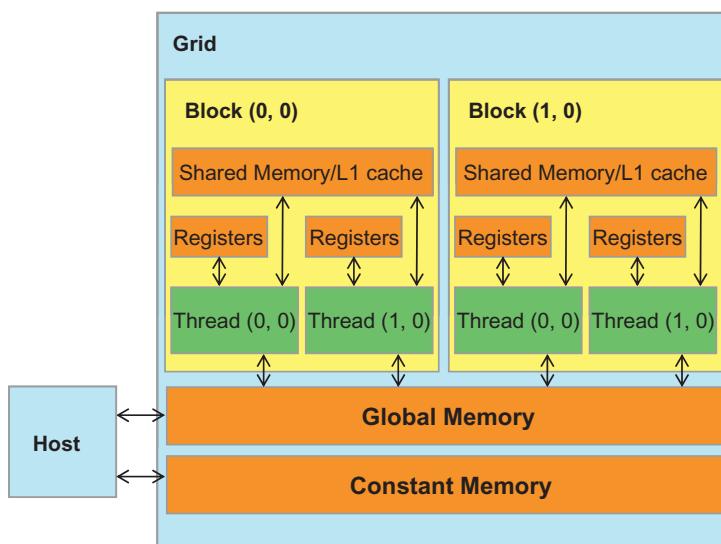
FIGURE 8.7

A review of the CUDA memory model.

The CUDA programming model allows programmers to declare a variable in the constant memory. Like global memory variables, constant memory variables are also visible to all thread blocks. The main difference is that a constant memory variable cannot be changed by threads during kernel execution. Furthermore, the size of the constant memory can vary from device to device. The amount of constant memory available on a device can be learned with a device property query. Assume that `dev_prop` is returned by `cudaGetDeviceProperties()`. The field `dev_prop.totalConstMem` indicates the amount of constant memory available on a device is in the field.

To use constant memory, the host code needs to allocate and copy constant memory variables in a different way than global memory variables. To declare an `M` array in constant memory, the host code declares it as a global variable as follows:

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

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 `M` should be placed into the device constant memory.

Assume that the host code has already allocated and initialized the mask in a mask `h_M` array in the host memory with `Mask_Width` elements. The contents of the `h_M` can be transferred to `M` in the device constant memory as follows:

```
cudaMemcpyToSymbol(M, h_M, Mask_Width*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 the `cudaMemcpyToSymbol()` function is as follows:

```
cudaMemcpyToSymbol(dest, src, size)
```

where `dest` is a pointer to the destination location in the constant memory, `src` is a pointer to the source data in the host memory, and `size` is the number of bytes to be copied.

Kernel functions access constant memory variables as global variables. Thus, their pointers do not need to be passed to the kernel as parameters. We can revise our kernel to use the constant memory as shown in Figure 8.8. Note that the kernel looks almost identical to that in Figure 8.6. The only difference is that `M` is no longer accessed through a pointer passed in as a parameter. It is now accessed as a global variable declared by the host code. Keep in mind that all the C language scoping rules for global variables apply here. If the host code and kernel code are

```
__global__ void convolution_1D_ba sic_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

FIGURE 8.8
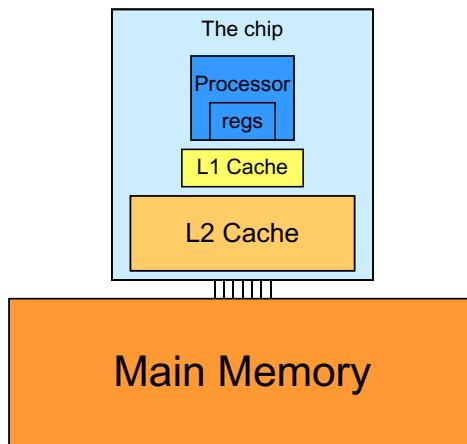
A 1D convolution kernel using constant memory for M.

in different files, the kernel code file must include the relevant external declaration information to ensure that the declaration of M 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.

In modern processors, accessing a variable from DRAM takes hundreds if not thousands of clock cycles. Also, the rate at which variables can be accessed from DRAM is typically much lower than the rate at which processors can perform an arithmetic operation. The long latency and limited bandwidth of DRAM has been a major bottleneck in virtually all modern processors commonly referred to as the memory wall. To mitigate the effect of memory bottleneck, modern processors commonly employ on-chip cache memories, or caches, to reduce the number of variables that need to be accessed from DRAM (Figure 8.9).

Unlike CUDA shared memory, or scratchpad memories in general, caches are "transparent" to programs. That is, to use CUDA shared memory, a program needs to declare variables as __shared__ and explicitly move a global memory variable into a shared memory variable. On the other hand, when using caches, the program simply accesses the original variables. The processor hardware will automatically retain some of the

FIGURE 8.9

A simplified view of the cache hierarchy of modern processors.

most recently or frequently used variables in the cache and remember their original DRAM 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 the cache. The value of the variable will then be provided from the cache, eliminating the need to access DRAM.

There is a trade-off between the size of a memory and the speed of a memory. As a result, modern processors often employ multiple levels of caches. The numbering convention for these cache levels reflects the distance to the processor. The lowest level, L1 or level 1, is the cache that is directly attached to a processor core. It runs at a speed very close to the processor in both latency and bandwidth. However, an L1 cache is small in size, typically between 16 KB and 64 KB. L2 caches are larger, in the range of 128 KB to 1 MB, but can take tens of cycles to access. They are typically shared among multiple processor cores, or streaming multiprocessors (SMs) in a CUDA device. In some high-end processors today, there are even L3 caches that can be of several MB in size.

A major design issue with using caches in a massively parallel processor is cache coherence, which arises when one or more processor cores modify cached data. Since L1 caches are typically directly attached to only one of the processor cores, changes in its contents are not easily observed by other processor cores. This causes a problem if the modified variable is shared among threads running on different processor cores.

A *cache coherence mechanism* is needed to ensure that the contents of the caches of the other processor cores are updated. Cache coherence is difficult and expensive to provide in massively parallel processors. However, their presence typically simplifies parallel software development. Therefore, modern CPUs typically support cache coherence among processor cores. While modern GPUs provide two levels of caches, they typically do without cache coherence to maximize hardware resources available to increase the arithmetic throughput of the processor.

Constant memory variables play an interesting role in using caches in massively parallel processors. Since they are not changed during kernel execution, there is no cache coherence issue during the execution of a kernel. Therefore, the hardware can aggressively cache the constant variable values in L1 caches. Furthermore, the design of caches in these processors is typically optimized to broadcast a value to a large number of threads. As a result, when all threads in a warp access the same constant memory variable, as is the case with M, the caches can provide a tremendous amount of bandwidth to satisfy the data needs of threads. Also, since the size of M is typically small, we can assume that all M elements are effectively always accessed from caches. Therefore, we can simply assume that no DRAM bandwidth is spent on M accesses. With the use of constant caching, we have effectively doubled the ratio of floating-point arithmetic to memory access to 2.

The accesses to the input N array elements can also benefit from caching in more recent devices. We will come back to this point in Section 8.5.

8.4 TILED 1D CONVOLUTION WITH HALO ELEMENTS

We now address the memory bandwidth issue in accessing the N array element with a tiled convolution algorithm. Recall that in a tiled algorithm, threads collaborate to load input elements into an on-chip memory and then access the on-chip memory for their subsequent use of these elements. For simplicity, we will continue to assume that each thread calculates one output P element. With up to 1,024 threads in a block we can process up to 1,024 data elements. We will refer to the collection of output elements processed by each block as an *output tile*. Figure 8.10 shows a small example of a 16-element, 1D convolution using four thread blocks of four threads each. In this example, there are four output tiles. The first output tile covers P[0] through P[3], the second tile P[4] through P[7],

the third tile P[8] through P[11], and the fourth tile P[12] through P[15]. Keep in mind that we use four threads per block to keep the example small. In practice, there should be at least 32 threads per block for the current generation of hardware. From this point on, we will assume that M elements are in the constant memory.

We will discuss two input data tiling strategies for reducing the total number of global memory accesses. The first one is the most intuitive and involves loading all input data elements needed for calculating all output elements of a thread block into the shared memory. The number of input elements to be loaded depends on the size of the mask. For simplicity, we will continue to assume that the mask size is an odd number equal to $2 \times n + 1$. That is, each output element P[i] is a weighted sum of the input element at the corresponding input element N[i], the n input elements to the left (N[i-n], ..., N[i-1]), and the n input elements to the right (N[i + 1], ..., N[i + n]). Figure 8.10 shows an example where $n = 2$.

Threads in block 0 calculate output elements P[0] through P[3]. This is the leftmost tile in the output data and is often referred to as the left *boundary tile*. They collectively require input elements N[0] through N[5]. Note that the calculation also requires two ghost elements to the left of N[0]. This is shown as two dashed empty elements on the left end of tile 0 of Figure 8.6. These ghost elements will be assumed have a default value of 0. Tile 3 has a similar situation at the right end of input array N. In our discussions, we will refer to tiles like tile 0 and tile 3 as boundary tiles
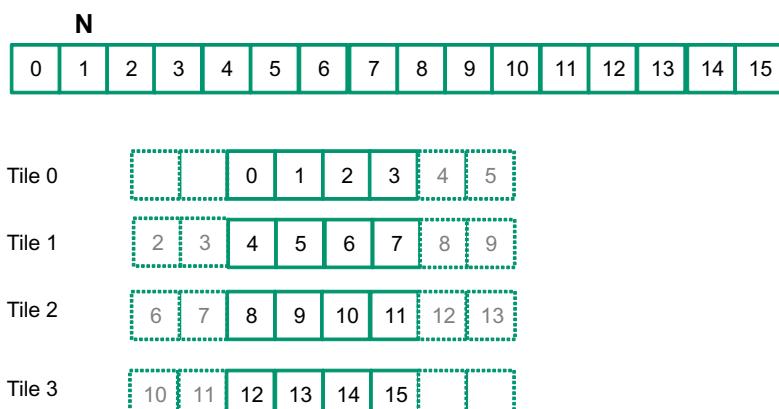
FIGURE 8.10

A 1D tiled convolution example.

since they involve elements at or outside the boundary of the input array N.

Threads in block 1 calculate output elements P[4] through P[7]. They collectively require input elements N[2] through N[9], also shown in Figure 8.10. Calculations for tiles 1 and 2 in Figure 8.10 do not involve ghost elements and are often referred to as *internal tiles*. Note that elements N[2] and N[3] belong to two tiles and are loaded into the shared memory twice, once to the shared memory of block 0 and once to the shared memory of block 1. Since the contents of shared memory of a block are only visible to the threads of the block, these elements need to be loaded into the respective shared memories for all involved threads to access them. The elements that are involved in multiple tiles and loaded by multiple blocks are commonly referred to as *halo elements* or *skirt elements* since they "hang" from the side of the part that is used solely by a single block. We will refer to the center part of an input tile that is solely used by a single block the *internal elements* of that input tile. Tiles 1 and 2 are commonly referred to as *internal tiles* since they do not involve any ghost elements at or outside the boundaries of the input array N.

We now show the kernel code that loads the input tile into shared memory. We first declare a shared memory array, N_ds, to hold the N tile for each block. The size of the shared memory array must be large enough to hold the left halo elements, the center elements, and the right halo elements of an input tile. We assume that Mask_Size is an odd number. The total is TILE_SIZE + MAX_MASK_WIDTH -1, which is used in the following declaration in the kernel:

```
__shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
```

We then load the left halo elements, which include the last n = Mask_Width/2 center elements of the previous tile. For example, in Figure 8.10, the left halo elements of tile 1 consist of the last two center elements of tile 0. In C, assuming that Mask_Width is an odd number, the expression Mask_Width/2 will result in an integer value that is the same as (Mask_Width-1)/2. We will use the last (Mask_Width/2) threads of the block to load the left halo element. This is done with the following two statements:

```
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
        (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

In the first statement, we map the thread index to the element index into the previous tile with the expression `(blockIdx.x-1)*blockDim.x + threadIdx.x`. We then pick only the last *n* threads to load the needed left halo elements using the condition in the `if` statement. For example, in Figure 8.6, `blockDim.x` equals 4 and `n` equals 2; only threads 2 and 3 will be used. Threads 0 and 1 will not load anything due to the failed condition.

For the threads used, we also need to check if their halo elements are ghost elements. This can be checked by testing if the calculated `halo_index_left` value is negative. If so, the halo elements are actually ghost elements since their `N` indices are negative, outside the valid range of the `N` indices. The conditional C assignment will choose 0 for threads in this situation. Otherwise, the conditional statement will use the `halo_index_left` to load the appropriate `N` elements into the shared memory. The shared memory index calculation is such that left halo elements will be loaded into the shared memory array starting at element 0. For example, in Figure 8.10, `blockDim.x-n` equals 2. So for block 1, thread 2 will load the leftmost halo element into `N_ds[0]` and thread 3 will load the next halo element into `N_ds[1]`. However, for block 0, both threads 2 and 3 will load value 0 into `N_ds[0]` and `N_ds[1]`.

The next step is to load the center elements of the input tile. This is done by mapping the `blockIdx.x` and `threadIdx.x` values into the appropriate `N` indices, as shown in the following statement. Readers should be familiar with the `N` index expression used:

```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
```

Since the first `n` elements of the `N_ds` array already contain the left halo elements, the center elements need to be loaded into the next section of `N_ds`. This is done by adding `n` to `threadIdx.x` as the index for each thread to write its loaded center element into `N_ds`.

We now load the right halo elements, which is quite similar to loading the left halo. We first map the `blockIdx.x` and `threadIdx.x` to the elements of next output tile. This is done by adding `(blockIdx.x + 1)*blockDim.x` to the thread index to form the `N` index for the right halo elements. In this case, we are loading the beginning `Mask_Width`:

```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
}
```

Now that all the input tile elements are in N_ds, each thread can calculate their output P element value using the N_ds elements. Each thread will use a different section of the N_ds. Thread 0 will use N_ds[0] through N_ds[Mask_Width-1]; thread 1 will use N_ds[1] through N[Mask_Width]. In general, each thread will use N_ds[threadIdx.x] through N[threadIdx.x + Mask_Width-1]. This is implemented in the following for loop to calculate the P element assigned to the thread:

```
float Pvalue = 0;
for(int j = 0; j < Mask_Width; j++) {
     Pvalue += N_ds[threadIdx.x + j]*M[j];
}
P[i] = Pvalue;
```

However, one must not forget to do a barrier synchronization using syncthreads() to make sure that all threads in the same block have completed loading their assigned N elements before anyone should start using them from the shared memory.

Note that the code for multiply and accumulate is simpler than the base algorithm. The conditional statements for loading the left and right halo elements have placed the 0 values into the appropriate N_ds elements for the first and last thread block.

The tiled 1D convolution kernel is significantly longer and more complex than the basic kernel. We introduced the additional complexity to reduce the number of DRAM accesses for the N elements. The goal is to improve the arithmetic to memory access ratio so that the achieved performance is not limited or less limited by the DRAM bandwidth. We will evaluate improvement by comparing the number of DRAM accesses performed by each thread block for the kernels in Figure 8.8 and Figure 8.11.

In Figure 8.8, there are two cases. For thread blocks that do not handle ghost elements, the number of N elements accessed by each thread is Mask_Width. Thus, the total number of N elements accessed by each thread block is blockDim.x*Mask_Width or blockDim.x*(2n + 1). For example, if Mask_Width is equal to 5 and each block contains 1,024 threads, each block accesses a total of 5,120 N elements.

For the first and last blocks, the threads that handle ghost elements, no memory access is done for the ghost elements. This reduces the number of memory accesses. We can calculate the reduced number of memory accesses by enumerating the number of threads that use each ghost element. This is illustrated with a small example in Figure 8.12. The leftmost ghost element is used by one thread. The second left ghost element is used by two threads. In general, the number of ghost elements is *n* and the

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH -1];

  int n = Mask_Width/2;

  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }

  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }

  __syncthreads();

  float Pvalue = 0;
  for(intj = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;

}
```
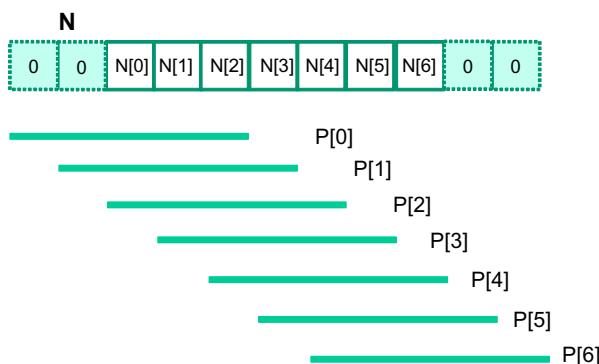
FIGURE 8.11

A tiled 1D convolution kernel using constant memory for M.

FIGURE 8.12

A small example of accessing N elements and ghost elements.

number of threads that use each of these ghost eleme*n*ts, from left to right is 1, 2, . . ., *n*. This is a simple series with sum $n(n + 1)/2$, which is the total number of accesses that were avoided due to ghost elements. For our simple example where `Mask_Width` is equal to 5 and `n` is equal to 2, the number of accesses avoided due to ghost elements is $2 \times 3/2 = 3$. A similar analysis gives the same results for the right ghost elements. It should be clear that for large thread blocks, the effect of ghost elements for small mask sizes will be insignificant.

We now calculate the total number of memory accesses for `N` elements by the tiled kernel in Figure 8.11. All the memory accesses have been shifted to the code that loads the `N` elements into the shared memory. In the tiled kernel, each `N` element is only loaded by one thread. However, $2n$ halo elements will also be loaded, *n* from the left and *n* from the right, for blocks that do not handle ghost elements. Therefore, we have the `blockDim.x + 2n` elements in for the internal thread blocks and `blockDim + n` for boundary thread blocks.

For internal thread blocks, the ratio of memory accesses between the basic and the tiled 1D convolution kernel is

`(blockDim.x*(2n + 1)) / (blockDim.x + 2n)`

whereas the ratio for boundary blocks is

`(blockDim.x*(2n + 1) − n(n + 1)/2) / (blockDim.x + n)`

For most situations, `blockDim.x` is much larger than `n`. Both ratios can be approximated by eliminating the small terms $n(n + 1)/2$ and *n*:

`(blockDim.x*(2n + 1)/ blockDim.x = 2n + 1 = Mask_Width`

This should be quite an intuitive result. In the original algorithm, each `N` element is redundantly loaded by approximately `Mask_Width` threads. For example, in Figure 8.12, `N[2]` is loaded by the five threads that calculate `P[2]`, `P[3]`, `P[4]`, `P[5]`, and `P[6]`. That is, the ratio of memory access reduction is approximately proportional to the mask size.

However, in practice, the effect of the smaller terms may be significant and cannot be ignored. For example, if `blockDim.x` is 128 and `n` is 5, the ratio for the internal blocks is

`(128*11-10) / (128 + 10) = 1398 / 138 = 10.13`

whereas the approximate ratio would be 11. It should be clear that as `blockDim.x` becomes smaller, the ratio also becomes smaller. For example, if `blockDim` is 32 and `n` is 5, the ratio for the internal blocks becomes

`(32*11-10) / (32 + 10) = 8.14`

Readers should always be careful when using smaller block and tile sizes. They may result in significantly less reduction in memory accesses than expected. In practice, smaller tile sizes are often used due to an

insufficient amount of on-chip memory, especially for 2D and 3D convolution where the amount of on-chip memory needed grows quickly with the dimension of the tile.

8.5 A SIMPLER TILED 1D CONVOLUTION—GENERAL CACHING

In Figure 8.11, much of the complexity of the code has to do with loading the left and right halo elements in addition to the internal elements into the shared memory. More recent GPUs such as Fermi provide general L1 and L2 caches, where L1 is private to each SM and L2 is shared among all SMs. This leads to an opportunity for the blocks to take advantage of the fact that their halo elements may be available in the L2 cache.

Recall that the halo elements of a block are also internal elements of a neighboring block. For example, in Figure 8.10, the halo elements N[2] and N[3] of tile 1 are also internal elements of tile 0. There is a significant probability that by the time block 1 needs to use these halo elements, they are already in the L2 cache due to the accesses by block 0. As a result, the memory accesses to these halo elements may be naturally served from the L2 cache without causing additional DRAM traffic. That is, we can leave the accesses to these halo elements in the original N elements rather than loading them into the N_ds. We now present a simpler tiled 1D convolution algorithm that only loads the internal elements of each tile into the shared memory.

In the simpler tiled kernel, the shared memory N_ds array only needs to hold the internal elements of the tile. Thus, it is declared with the TILE_SIZE, rather than TILE_SIZE + Mask_Width-1:

```
__shared__ float N_ds[TILE_SIZE];
i = blockIdx.x*blockDim.x + threadIdx.x;
```

Loading the tile becomes very simple with only one line of code:

```
N_ds[threadIdx.x] = N[i];
```

We still need a barrier synchronization before using the elements in N_ds. The loop that calculates P elements, however, becomes more complex. It needs to add conditions to check for use of both halo elements and ghost elements. The ghost elements are handled with the same conditional statement as that in Figure 8.6. The multiply–accumulate statement becomes more complex:

```
__syncthreads();
int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);
```

```
float Pvalue = 0;
for (int j = 0; j < Mask_Width; j ++) {
    int N_index = N_start_point + j;
    if (N_index > = 0 && N_index < Width) {
        if ((N_index > = This_tile_start_point)
          && (N_index < Next_tile_start_point)) {
          Pvalue + = N_ds[threadIdx.x + j-(Mask_Width/2)]*M[j];
        } else {
          Pvalue + = N[N_index] * M[j];
        }
    }
}
P[i] = Pvalue;
```

The variables `This_tile_start_point` and `Next_tile_start_point` hold the starting position index of the tile processed by the current block and that of the tile processed by the next in the next block. For example, in Figure 8.10, the value of `This_tile_start_point` for block 1 is 4 and the value of `Next_tile_start_point` is 8.

The new `if` statement tests if the current access to the `N` element falls within a tile by testing it against `This_tile_start_point` and `Next_tile_start_point`. If the element falls within the tile—that is, it is an internal element for the current block—it is accessed from the `N_ds` array in the shared memory. Otherwise, it is accessed from the `N` array, which is hopefully in the L2 cache. The final kernel code is shown is Figure 8.13.

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

8.6 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 other parallel algorithms. For example, one can view the stencil algorithms in partial differential equation (PDE) solvers as a special case of convolution. For another example, one can also view

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int Mask_Width,
int Width) {

int i = blockIdx.x*blockDim.x + threadIdx.x;
__shared__ float N_ds[TILE_SIZE];

N_ds[threadIdx.x] = N[i];

__syncthreads();

int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);
float Pvalue = 0;
for (intj = 0; j < Mask_Width; j ++) {
   int N_index = N_start_point + j;
   if (N_index >= 0  && N_index < Width) {
     if ((N_index >= This_tile_start_point)
        && (
        Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
     } else {
        Pvalue += N[N_index] * M[j];
     }
   }
}
P[i] = Pvalue;

}
```

FIGURE 8.13

A simpler tiled 1D convolution kernel using constant memory and general caching.

the calculation of grid point force or the potential value as a special case of convolution.

We have presented a basic parallel convolution algorithm of which the implementations will be limited by DRAM bandwidth for accessing both the input N and mask M elements. We then introduced the 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 mask elements. We further introduced a tiled parallel convolution algorithm that reduces DRAM bandwidth consumption by introducing more control flow divergence and programming complexity. Finally, we presented a simpler tiled parallel convolution algorithm that takes advantage of the L2 caches.

8.7 EXERCISES

8.1. Calculate the P[0] value in Figure 8.3.

8.2. Consider performing a 1D convolution on array N = {4,1,3,2,3} with mask M = {2,1,4}. What is the resulting output array?

8.3. What do you think the following 1D convolution masks 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]

8.4. Consider performing a 1D convolution on an array of size n with a mask of size m:

    a. How many halo cells are there in total?

    b. How many multiplications are performed if halo cells are treated as multiplications (by 0)?

    c. How many multiplications are performed if halo cells are not treated as multiplications?

8.5. Consider performing a 2D convolution on a square matrix of size $n \times n$ with a square mask of size $m \times m$:

    a. How many halo cells are there in total?

    b. How many multiplications are performed if halo cells are treated as multiplications (by 0)?

    c. How many multiplications are performed if halo cells are not treated as multiplications?

8.6. Consider performing a 2D convolution on a rectangular matrix of size $n1 \times n2$ with a rectangular mask of size $m1 \times m2$:

    a. How many halo cells are there in total?

    b. How many multiplications are performed if halo cells are treated as multiplications (by 0)?

    c. How many multiplications are performed if halo cells are not treated as multiplications?

8.7. Consider performing a 1D tiled convolution with the kernel shown in Figure 8.11 on an array of size n with a mask of size m using a tile of size t.

    a. How many blocks are needed?

      b. How many threads per block are needed?

      c. How much shared memory is needed in total?

      d. Repeat the same questions if you were using the kernel in Figure 8.13.

8.8. Revise the 1D kernel in Figure 8.6 to perform 2D convolution. Add more width parameters to the kernel declaration as needed.

8.9. Revise the tiled 1D kernel in Figure 8.8 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.

8.10. Revise the tiled 1D kernel in Figure 8.11 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.

Parallel Patterns: Prefix Sum

9

An Introduction to Work Efficiency in Parallel Algorithms

CHAPTER OUTLINE

Our next parallel pattern is prefix sum, which is also commonly known as *scan*. Parallel scan is frequently used to convert seemingly sequential operations, such as resource allocation, work assignment, and polynomial evaluation, into parallel operations. In general, if a computation is naturally described as a mathematical recursion, 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. Another reason why parallel scan is an important parallel pattern is that sequential scan algorithms are linear algorithms and are extremely work-efficient, which makes it also very important to control the work efficiency of parallel scan algorithms. As we will show, a slight increase in algorithm complexity can make parallel scan run slower than sequential scan for large data sets. Therefore, work-efficient parallel scan also represents an important class of parallel algorithms that can run effectively on parallel systems with a wide range of available computing resources.

9.1 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 output array

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

For example, if $\oplus$ is addition, then an inclusive scan operation on the input array `[3 1 7 0 4 1 6 3]` would return `[3 4 11 11 15 16 22 25]`.

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, 3. That is, person number 0 wants 3 inches of sausage, person number 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 number 0. The sausage is now 37 inches long. We then cut a 1-inch section for person number 1. The sausage becomes 36 inches long. We can continue to cut more sections until we serve the 3-inch section to person number 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 all the cutting points based on the amount each person orders. 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 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 point so the first section will be 3 inches, as ordered by person number 0. The second cut point is 4, therefore the second section will be 1-inch long, as ordered by person number 1. The final cut will be at the 25-inch point, which will produce a 3-inch long section since the previous cut point is at the 22-inch point. This gives person number 7 what she ordered. Note that since all the cutting points are known from the scan operation, all cuts can be done in parallel.

In summary, an intuitive way of thinking about inclusive scan is that the operation takes an order from a group of people and identifies all the cutting 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 cutting points, all orders can be served in parallel.

An exclusive scan operation is similar to an inclusive operation with the exception that it returns the output array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})]$$

That is, the first output element is 0 while the last output element only reflects the contribution of up to x_{n-2}.

The applications of an exclusive scan operation are pretty much the same as those for an inclusive scan. The inclusive scan provides slightly different information. In the sausage example, the 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 number 0 starts at the 0-inch point. For another example, the section for person number 7 starts at the 22-inch point. The beginning point information is important in applications such as memory allocation, where the allocated memory is returned to the requester via a pointer to its beginning point.

Note that it is fairly easy to convert between the inclusive scan output and the exclusive scan output. One simply needs to do a shift and fill in an element. When converting from inclusive to exclusive, one can simply shift all elements to the right and fill in value 0 for the 0 element. When converting from exclusive to inclusive, we need to shift all elements to the left and fill in the last element with the previous last element plus the last input element. It is just a matter of convenience that we can directly generate an inclusive or exclusive scan, whether we care about the cutting points or the beginning points for the sections.

In practice, 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 histograms.

Before we present parallel scan algorithms and their implementations, we would like to first show a work-efficient sequential inclusive scan algorithm and its implementation. We will assume that the operation is addition. The algorithm assumes that the input elements are in the x array and the output elements are to be written into the y array.

```
void sequential_scan(float *x, float *y, int Max_i) {
y[0] = x[0];
for (int i = 1; i < Max_i; i++) {
   y[i] = y [i-1] + x[i];
}
}
```

The algorithm is work-efficient. With a reasonably good compiler, only one addition, one memory load, and one memory store are used in processing each input x element. This is pretty much the minimal that we will

ever be able to do. As we will see, when the sequential algorithm of a computation is so "lean and mean," it is extremely challenging to develop a parallel algorithm that will consistently beat the sequential algorithm when the data set size becomes large.

9.2 A SIMPLE PARALLEL SCAN

We start with a simple parallel inclusive scan algorithm by doing a reduction operation for all output elements. The main idea is to create each element quickly by calculating a reduction tree of the relevant input elements for each output element. There are multiple ways to design the reduction tree for each output element. We will present a simple one that is shown in Figure 9.1.

The algorithm is an in-place scan algorithm that operates on an array XY that originally contains input elements. It then iteratively evolves the
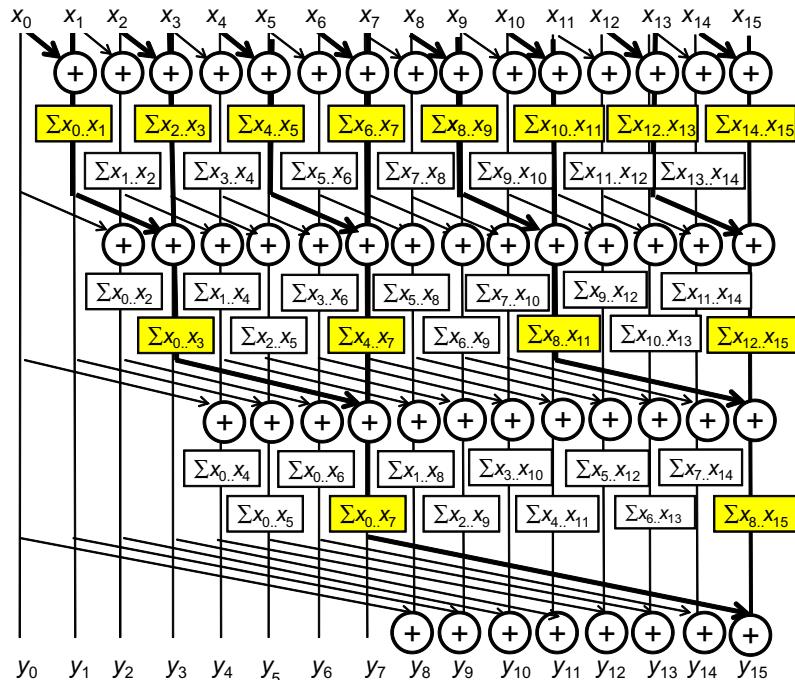
FIGURE 9.1

A simple but work-inefficient parallel inclusive scan.

contents of the array into output elements. Before the algorithm begins, we assume XY[i] contains input element x_i. At the end of iteration n, XY[i] will contain the sum of 2^n input elements at and before the location. That is, at the end of iteration 1, 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.

Figure 9.1 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 the 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 Figure 9.1. 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 Figure 9.1. For example, after the first iteration, XY[3] contains $x_2 + x_3$, shown as $\sum x_2..x_3$. 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] receive the sum of its current content and that of the position that is two elements away. This is illustrated in the labeling boxes below the second row of addition operators. As a result, XY[i] now contains $x_{i-3} + x_{i-2} + x_{i-1} + x_i$. For example, after the first iteration, XY[3] contains $x_0 + x_1 + x_2 + x_3$, shown as $\sum x_0..x_3$. Note that after the second iteration, XY[2] and XY[3] contain their final answers and will not need to be changed in subsequent iterations.

Readers are encouraged to work through the rest of the iterations. We now work on the implementation of the algorithm illustrated in Figure 9.1. We assign each thread to evolve the contents of one XY element. We will write a kernel that performs a scan on a section of the input that is small enough for a block to handle. The size of a section is defined as a compile-time constant SECTION_SIZE. We assume that the kernel launch will use SECTION_SIZE as the block size so there will be an equal number of threads and section elements. All results will be calculated as if the array only has the elements in the section. Later, we will make final adjustments to these sectional scan results for large input arrays. We also assume that input values were originally in a global memory array X, the address of which is passed into the kernel as an argument. We will have all the threads in the block to collaboratively load the X array elements

into a shared memory array `XY`. At the end of the kernel, each thread will write its result into the assigned output array `Y`.

```
__global__ void work_inefficient_scan_kernel(float *X, float * Y,
    int InputSize) {
    __shared__ float XY[SECTION_SIZE];
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (i < InputSize) {
       XY[threadIdx.x] = X[i];
    }
    // the code below performs iterative scan on XY
    ...
    Y[i] = XY[threadIdx.x];
}
```

We now focus on the implementation of the iterative calculations for each `XY` element in Figure 9.1 as a `for` loop:

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

The loop iterates through the reduction tree for the `XY` array position that is assigned to a thread. Note that we use a barrier synchronization to make sure that all threads have finished their current iteration of additions in the reduction tree before any of them starts the next iteration. This is the same use of `__syncthreads();` as in the reduction discussion in Chapter 6. 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. Thus, the thread can exit the `while` loop. The smaller the `threadIdx.x` value, the earlier the thread will exit the `while` loop. This is consistent with the example shown in Figure 9.1. The actions of the smaller positions of `XY` end earlier than the larger positions. This will cause some level of control divergence in the first warp when stride values are small. The effect should be quite modest for large block sizes since it only impacts the first loop for smaller stride values. The detailed analysis is left as an exercise. The final kernel is shown in Figure 9.2.

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 value 0. This is illustrated in Figure 9.3. Note that the only real

```
__global__ void work_inefficient_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }

  // the code below performs iterative scan on XY
  for (unsigned int stride = 1; stride <= threadIdx.x; stride *= 2) {
    __syncthreads();
    XY[threadIdx.x] += XY[threadIdx.x-stride];
  }

  Y[i] = XY[threadIdx.x];

}
```
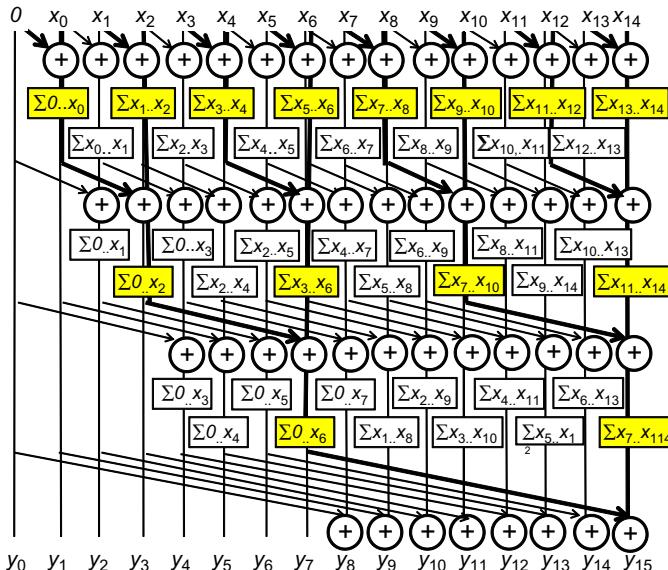
FIGURE 9.2

A kernel for the inclusive scan algorithm in Figure 9.1.

difference is the alignment of elements on top of the picture. All labeling

FIGURE 9.3

Work-inefficient parallel exclusive scan.

boxes are updated to reflect the new alignment. All iterative operations remain the same.

We can now easily convert the kernel in Figure 9.2 into an exclusive scan kernel. The only modification we need to do is to load 0 into XY[0] and X[i-1] into XY[threadIdx.x], as shown in the following code:

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

Note that the XY positions of which the associated input elements are outside the range are now also filled with 0. This causes no harm and yet it simplifies the code slightly. We leave the work to finish the exclusive scan kernel as an exercise.

9.3 WORK EFFICIENCY CONSIDERATIONS

We now analyze the work efficiency of the kernel in Figure 9.2. All threads will iterate up to $\log(N)$ steps, where N is the SECTION_SIZE. In each iteration, the number of threads that do not need to do any addition is equal to the stride size. Therefore, we can calculate the amount of work done for the algorithm as

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

The first part of each term is independent of the stride, so they add up to $N \times \log_2(N)$. The second part is a familiar geometric series and sums up to $(N - 1)$. So the total number of add operations is

$$N \times \log_2(N) - (N - 1)$$

Recall that the number of add operations for a sequential scan algorithm is $N - 1$. We can put this into perspective by comparing the number of add operations for different N values, as shown in Figure 9.4. Note that even for modest-size sections, the kernel in Figure 9.2 does much more work than the sequential algorithm. In the case of 1,024 elements, the kernel does nine times more work than the sequential code. The ratio will continue to grow as N becomes larger. Such additional work is problematic in two ways. First, the use of hardware for executing the parallel kernel needs to be much less efficient. In fact, just to break even one needs to have at least nine times more execution units in a parallel machine than

N	16	32	64	128	256	512	1024
N − 1	15	31	63	127	255	511	1023
N·log₂(N) − (N −1)	49	129	321	769	1793	4097	9217

FIGURE 9.4

Work efficiency calculation for the kernel in Figure 9.2.
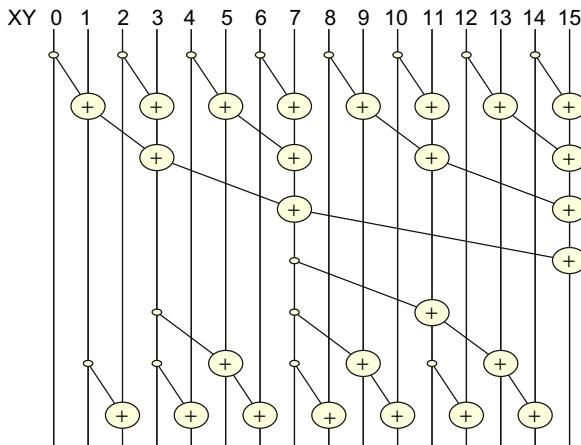
the sequential machine. For example, if we execute the kernel on a parallel machine with four times the execution resources as a sequential machine, the parallel machine executing the parallel kernel can end up with only half the performance of the sequential machine executing the sequential code. Second, all the extra work consumes additional energy. This makes the kernel inappropriate for power-constrained environments such as mobile applications.

9.4 A WORK-EFFICIENT PARALLEL SCAN

While the kernel in Figure 9.2 is conceptually simple, its work efficiency is too low for many practical applications. Just by inspecting Figures 9.1 and 9.3, we can see that there are potential opportunities for sharing some intermediate results to streamline the operations performed. However, to allow more sharing across multiple threads, we need to quickly calculate the intermediate results to be shared and then quickly distribute them to different threads.

As we know, the fastest parallel way to produce sum values for a set of values is a reduction tree. A reduction tree can generate the sum for N values in $\log_2(N)$ steps. Furthermore, the tree can also generate a number of subsums that can be in the calculation of some of the scan output values.

In Figure 9.5, 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 changed to $x_{i-1} + x_i$. During the second step, only the XY elements of which the indices are of the form of $4 \times n - 1$, which are 3, 7, 11, and 15 in Figure 9.5, will be updated. During the third step, only the XY elements of which the indices are of the form $8 \times n - 1$, which are 7 and 15, will be updated. Finally, during the fourth step, only XY[15] is updated. The total number of

FIGURE 9.5

Basic idea of a work-efficient parallel scan algorithm.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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}	$x_0.x_{15}$
											$x_{0..}x_{11}$				
					$x_{0..}x_5$				$x_{0..}x_9$				$x_{0..}x_{13}$		

FIGURE 9.6

Partial sums available in each XY element after the reduction tree phase.

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 these values as quickly as possible. This is illustrated in the bottom half of Figure 9.5. At the end of the reduction phase, we have quite a few usable partial sums. For our example, the first row of Figure 9.6 shows all the partial sums in XY right after the top reduction tree. An important observation is that XY[0], XY[7], and X[15] contain their final answers. Therefore, all remaining XY elements can obtain the partial sums they need from no farther than four positions away. For example, XY[14] can obtain all the partial sums it needs

from XY[7], XY[11], and XY[13]. 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 away, then one position way. By inspection, XY[7] contains a critical value needed by many positions in the right half. A good way is to add XY[7] to XY[11], which brings XY[11] to the final answer. More importantly, XY[7] also becomes a good partial sum for XY[12], XY[13], and XY[14]. No other partial sums have so many uses. Therefore, there is only one addition, XY[11] = XY[7] + XY[11], that needs to occur in the four-position level in Figure 9.5. We show the updated partial sum in the second row of Figure 9.6.

We now identify all additions for getting partial sums that are two positions away. We see that XY[2] only needs the partial sum that is next to it in XY[1]. This is the same with XY[4]—it needs the partial sum next to it to be complete. The first XY element that can need a partial sum two positions away is XY[5]. Once we calculate XY[5] = XY[3] + XY[5], XY[5] contains the final answer. The same analysis shows that XY[6] and XY[8] can become complete with the partial sums next to them in XY[5] and XY[7].

The next two-position addition is XY[9] = XY[7] + XY[9], which makes XY[9] complete. XY[10] can wait for the next round to catch XY[9]. XY[12] only needs the XY[11], which contains its final answer after the four-position addition. The final two-position addition is XY[13] = XY[11] + XY[13]. The third row shows all the updated partial sums in XY[5], XY[9], and XY[13]. It is clear that now every position is either complete or can be completed when added by its left neighbor. This leads to the final row of additions in Figure 9.5, which completes the contents for all the incomplete positions XY[2], XY[4], XY[6], XY[8], XY[10], and XY[12].

We could implement the reduction tree phase of the parallel scan using the following loop:

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

Note that this loop is very similar to the reduction in Figure 6.2. The only difference is that we want the thread that has a thread index that is in the form of $2^n - 1$, rather than 2^n to perform addition in each iteration. This is why we added 1 to the threadIdx.x when we select the threads for performing addition in each iteration. However, this style of reduction

is known to have control divergence problems. A better way to do this is to use a decreasing number of contiguous threads to perform the additions as the loop advances:

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

In our example in Figure 9.5, there are 16 threads in the block. In the first iteration, the stride is equal to 1. The first 8 consecutive threads in the block will satisfy the if condition. The 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 Figure 9.5. In the second iteration, the stride is equal to 2. Only the first 4 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 Figure 9.5. Note that since we will always be using consecutive threads in each iteration, the control divergence problem does not arise until the number of active threads drops below the warp size.

The distribution tree is a little more complex to implement. We make an observation that the stride value decreases from SECTION_SIZE/2 to 1. In each iteration, we need to "push" the value of the XY element from a position that is a multiple of the stride value minus 1 to a position that is a stride away. For example, in Figure 9.5, the stride value decreases from 8 to 1. In the first iteration in Figure 9.5, we would like to push the value of XY[7] to XY[11], where 7 is $8 - 1$. In the second iteration, we would like to push the values of XY[3], XY[7], and XY[11] to XY[5], XY[9], and XY[13]. This 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 < BLOCK_SIZE) {
      XY[index + stride] + = XY[index];
    }
 }
```

The calculation of index is similar to that in the reduction tree phase. The final kernel for a work-efficient parallel scan is shown in Figure 9.7.

```
__global__ void work_efficient_scan_kernel (float *X, float *Y, int InputSize) {

  __shared__ float XY[SECTION_SIZE];

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
    XY[threadIdx.x] = X[i];
  }

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

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

  __syncthreads();

  Y[i] = XY[threadIdx.x];

}
```

FIGURE 9.7

A work-efficient kernel for an inclusive scan.

Readers should notice that we never need to have more than SECTION_SIZE/2 threads for either the reduction phase or the distribution phase. So, we could simply launch a kernel with SECTION_SIZE/2 threads in a block. Since we can have up to 1,024 threads in a block, each scan section can have up to 2,048 elements. However, we will need to have each thread to load two X elements at the beginning and store two Y elements at the end. This will be left as an exercise.

As was the case of the work-inefficient scan kernel, one can easily adapt the work-efficient 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 2007] for an interesting natively exclusive scan kernel that is based on a different way of designing the distribution tree phase of the scan kernel.

We now analyze the number of operations in the distribution tree stage. The number of operations are $(16/8) - 1 + (16/4) + (16/2)$. In general, for N input elements, the total number of operations would be $(N/2) + (N/4)$

$+ \ldots + 4 + 2 - 1$, which is less than $N - 2$. This makes the total number of operations in the parallel scan $2 \times N - 3$. Note that the number of operations is now a proportional to N, rather than $N \times \log_2(N)$. We compare the number of operations performed by the two algorithms for N from 16 to 1,024 in Figure 9.8.

The advantage of a work-efficient algorithm is quite clear in the comparison. As the input section becomes bigger, the work-efficient algorithm never performs more than two times the number of operations performed by the sequential algorithm. As long as we have at least two times more hardware execution resources, the parallel algorithm will achieve better performance than the sequential algorithm. This is not true, however, for the work-inefficient algorithm. For 102 elements, the parallel algorithm needs at least nine times the hardware execution resources just to break even.

9.5 PARALLEL SCAN FOR ARBITRARY-LENGTH INPUTS
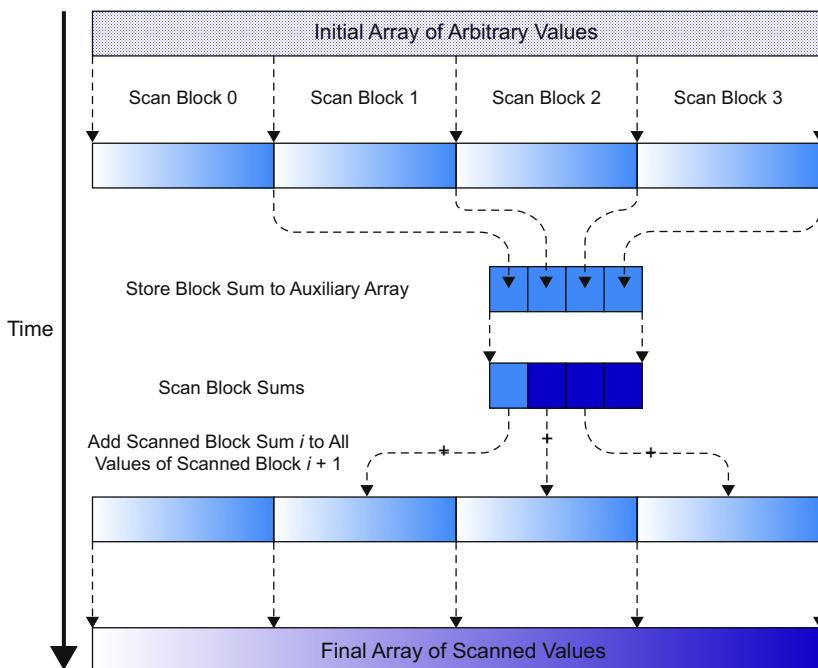
For many applications, the number of elements to be processed by a scan operation can be in the millions. Obviously, we cannot expect that all input elements can fit into the shared memory. Furthermore, it would be a loss of parallelism opportunity if we used only one thread block to process these large data sets. Fortunately, there is a hierarchical approach to extending the scan kernels that we have generated so far to handle inputs of arbitrary size. The approach is illustrated in Figure 9.9.

For a large data set, we first partition the input into sections that can fit into the shared memory and processed by a single block. For the current generation of CUDA devices, the work-efficient kernel in Figure 9.8 can process up to 2,048 elements in each section using 1,024 threads in each block. For example, if the input data consists of 2,000,000 elements, we can use ceil(2,000,000/2,048.0) = 977 thread blocks. With up to 65,536 thread blocks in the x dimension of a grid, the approach can process up to

N	16	32	64	128	256	512	1024
$N - 1$	15	31	63	127	255	511	1023
$N \cdot \log_2(N) - (N - 1)$	49	129	321	769	1793	4097	9217
$2 \cdot N - 3$	29	61	125	253	509	1021	2045

FIGURE 9.8

Work efficiency of the kernels.

FIGURE 9.9

A hierarchical scan for arbitrary-length inputs.
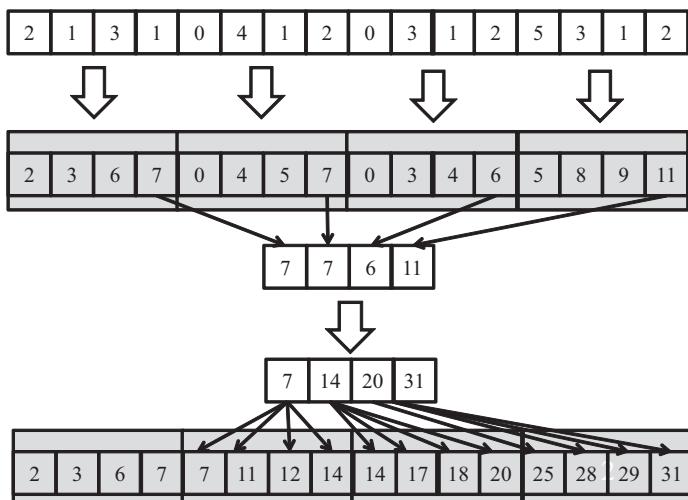
134,217,728 elements in the input set. If the input is even bigger than this, we can use additional levels of hierarchy to handle a truly arbitrary number of input elements. However, for this chapter, we will restrict our discussion to a two-level hierarchy that can process up to 134,217,728 elements.

Assume that the host code launches the kernel in Figure 9.7 on the input. Note that the kernel uses the familiar `i = blockIdx*blockDim.x + trheadIdx.x` statement to direct threads in each block to load their input values from the appropriate section. At the end of the grid execution, the threads write their results into the Y array. That is, after the kernel in Figure 9.7 completes, the Y array contains the scan results for individual sections, called scan blocks in Figure 9.9. Each result 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 write and launch another kernel that adds the sum of all elements in preceding scan blocks to each element of a scan block.

Figure 9.10 shows a small operational example of the hierarchical scan approach of Figure 9.9. In this example, there are 16 input elements that are divided into four scan blocks. The kernel treats the four scan blocks as independent input data sets. After the scan kernel terminates, each Y element contains the scan result with its scan block. For example, scan block 1 has inputs 0, 4, 1, 2. The scan kernel produces the scan result for this section (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 $(2 + 1 + 3 + 1 = 7)$ should be added to every result element of scan block 1.

For another example, the inputs in scan block 2 are 0, 3, 1, and 2. The kernel produces the scan result for this scan block (0, 3, 4, 6). To produce the final results for this scan block, the sum of all elements in both scan blocks 0 and 1 $(2 + 1 + 3 + 1 + 0 + 4 + 1 + 2 = 14)$ should be added to every result element of scan block 2.

It is important to note that the last scan 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 Figure 9.10. This brings us to the second step of the hierarchical scan algorithm in Figure 9.9, which gathers the last result elements from each scan block into an array and performs a scan on these

FIGURE 9.10

An example of a hierarchical scan.

output elements. This step is also illustrated in Figure 9.10, where the last scan output elements are all collected into a new array S. This 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 index. A scan operation is then performed on S to produce output values 7, 14, 20, and 31. Note that each of these second-level scan output values are the accumulated sum from the beginning location X[0] to the end of each scan block. That is, the output value in S[0] = 7 is the accumulated sum from X[0] to the end of scan block 0, which is X[3]. The output 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. That is, in Figure 9.10, 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]. 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 hierarchical scan algorithm in Figure 9.9. The second-level scan output values are added to the values of their corresponding scan blocks.

For example, in Figure 9.10, the value of S[0] (7) will be added to Y[0], Y[1], Y[2], and Y[3] of thread block 1, which completes the results in these positions. The final results in these positions are 7, 11, 12, and 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], and Y[11], which completes the results in these positions. The value of S[2] will be added to S[2] (20), which will be added to Y[12], Y[13], Y[14], and 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 hierarchical scan algorithm is quite similar to the carry look-ahead in hardware adders of modern processors.

We can implement the hierarchical scan with three kernels. The first kernel is largely the same as the kernel in Figure 9.7. We need to add one more parameter S, which has the dimension of InputSize/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:

```
__synchtrheads();
if (threadIdx.x = = 0) {
```

```
        S[blockIdx.x] = XY[SECTION_SIZE − 1];
    }
```

The second kernel is simply the same kernel as Figure 9.7, which takes S as input and writes S as output.

The third kernel takes the S and Y arrays as inputs and writes the output back into Y. The body of the kernel adds one of the S elements to all Y elements:

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

We leave it as an exercise for readers to complete the details of each kernel and complete the host code.

9.6 SUMMARY

In this chapter, we studied scan as an important parallel computation pattern. Scan is used to enable parallel allocation of resources parties of which the needs are not uniform. It converts seemingly sequential recursive computation into parallel computation, which helps to reduce sequential bottlenecks in many applications. We showed that a simple sequential scan algorithm performs only N additions for an input of N elements.

We first introduced a parallel scan algorithm that is conceptually simple but not work-efficient. As the data set size increases, the number of execution units needed for a parallel algorithm to break even with the simple sequential algorithm also increases. For an input of 1,024 elements, the parallel algorithm performs over nine times more additions than the sequential algorithm and requires at least nine times more executions to break even with the sequential algorithm. This makes the work-inefficient parallel algorithm inappropriate for power-limited environments such as mobile applications.

We then presented a work-efficient parallel scan algorithm that is conceptually more complicated. Using a reduction tree phase and a distribution tree phase, the algorithm performs only $2 \times N - 3$ additions no matter how large the input data sets are. Such work-efficient algorithms of which the number of operations grows linearly with the size of the input set are often also referred to as data-scalable algorithms. We also presented a hierarchical approach to extending the work-efficient parallel scan algorithm to handle the input sets of arbitrary sizes.

9.7 EXERCISES

9.1. Analyze the parallel scan kernel in Figure 9.2. Show that control divergence only occurs in the first warp of each block for stride values up to half of the warp size. That is, for warp size 32, control divergence will occur to iterations for stride values 1, 2, 4, 8, and 16.

9.2. For the work-efficient scan kernel, assume that we have 2,048 elements. How many add operations will be performed in both the reduction tree phase and the inverse reduction tree phase?

   a. $(2{,}048 - 1) \times 2$

   b. $(1{,}024 - 1) \times 2$

   c. $1{,}024 \times 1{,}024$

   d. $10 \times 1{,}024$

9.3. For the work-inefficient scan kernel based on reduction trees, assume that we have 2,048 elements. Which of the following gives the closest approximation on how many add operations will be performed?

   a. $(2{,}048\text{-}1) \times 2$

   b. $(1{,}024\text{-}1) \times 2$

   c. $1{,}024 \times 1{,}024$

   d. $10 \times 1{,}024$

9.4. Use the algorithm in Figure 9.3 to complete an exclusive scan kernel.

9.5. Complete the host code and all the three kernels for the hierarchical parallel scan algorithm in Figure 9.9.

9.6. Analyze the hierarchical parallel scan algorithm and show that it is work-efficient and the total number of additions is no more than $4 \times N - 3$.

9.7. Consider the following array: `[4 6 7 1 2 8 5 2]`. Perform a parallel inclusive prefix scan on the array using the work-inefficient algorithm. Report the intermediate states of the array after each step.

9.8. Repeat Exercise 9.7 using the work-efficient algorithm.

9.9. Using the two-level hierarchical scan discussed in Section 9.5, what is the largest possible data set that can be handled if computing on a:

   a. GeForce GTX280?

   b. Tesla C2050?

   c. GeForce GTX690?

Reference

Harris, M. Parallel Prefix Sum with CUDA, Available at: 2007 <http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf>.

Parallel Patterns: Sparse Matrix−Vector Multiplication 10

An Introduction to Compaction and Regularization in Parallel Algorithms

Our next parallel pattern is sparse matrix computation. In a sparse matrix, the vast majority of the elements are zeros. Storing and processing these zero elements are wasteful in terms of memory, time, and energy. Many important real-world problems involve sparse matrix computations that are highly parallel in nature. Due to the importance of these problems, several sparse matrix storage formats and their corresponding processing methods have been proposed and widely used in the field. All of them employ some type of compaction techniques to avoid storing or processing zero elements at the cost of introducing some level of irregularity into the data representation. Unfortunately, such irregularity can lead to underutilization of memory bandwidth, control flow divergence, and load imbalance in parallel computing. It is therefore important to strike a good balance between compaction and regularization. Some storage formats achieve a higher level of compaction at a high level of irregularity. Others achieve a more modest level of compaction while keeping the representation more regular. The parallel computation performance of their corresponding

methods is known to be heavily dependent on the distribution of nonzero elements in the sparse matrices. Understanding the wealth of work in sparse matrix storage formats and their corresponding parallel algorithms gives a parallel programmer an important background for addressing compaction and regularization challenges in solving related problems.

10.1 BACKGROUND

A sparse matrix is a matrix where the majority of the elements are zero. Sparse matrices arise in many science, engineering, and financial modeling problems. For example, as we saw in Chapter 7, matrices are often used to represent the coefficients in a linear system of equations. Each row of the matrix represents one equation of the linear system. In many science and engineering problems, there are a large number of variables and the equations involved are loosely coupled. That is, each equation only involves a small number of variables. This is illustrated in Figure 10.1, where variables x_0 and x_2 are involved in equation 0, none of the variables in equation 1, variables x_1, x_2, and x_3 in equation 2, and finally variables x_0 and x_3 in equation 3.

Sparse matrices are stored in a format that avoids storing zero elements. We will start with the compressed sparse row (CSR) storage format, which is illustrated in Figure 10.2. CSR stores only nonzero values in a 1D data storage, shown as data[] in Figure 10.2. Array data[] stores all the nonzero values in the sparse matrix in Figure 10.1. This is done by storing nonzero elements of row 0 (3 and 1) first, followed by nonzero elements of row 1 (none), followed by nonzero elements of row 2 (2, 4, 1), and then nonzero elements of row 3 (1, 1). The format compresses away all zero elements.

With the compressed format, we need to put in two sets of markers to preserve the structure of the original sparse matrix. The first set of markers form a column index array, col_index[] in Figure 10.2, that gives the

Row 0	3	0	1	0
Row 1	0	0	0	0
Row 2	0	2	4	1
Row 3	1	0	0	1

FIGURE 10.1

A simple sparse matrix example.

column index of every nonzero value in the original sparse matrix. Since we have squeezed away nonzero elements of each row, we need to use these markers to remember where the remaining elements were in the original row of the sparse matrix. For example, values 3 and 1 came from columns 0 and 2 of row 0 in the original sparse matrix. The `col_index[0]` and `col_index[1]` elements are assigned to store the column indices for these two elements. For another example, values 2, 4, and 1 came from columns 1, 2, and 3 in the original sparse matrix. Therefore, `col_index[0]`, `col_index[1]`, and `col_index[2]` store indices 1, 2, and 3.

The second set of markers give the starting location of every row in the compressed storage. This is because the size of each row becomes variable after zero elements are removed. It is no longer possible to use indexing based on the row size to find the starting location of each row in the compressed storage. In Figure 10.2, we show a `row_ptr[]` array of which the elements are the pointers or indices of the beginning locations of each row. That is, `row_ptr[0]` indicates that row 0 starts at location 0 of the `data[]` array, `row_ptr[1]` indicates that row 1 starts at location 2, etc. Note that `row_ptr[1]` and `row_ptr[2]` are both 2. This means that none of the elements of the row 1 was stored in the compressed format. This makes sense since row 1 in Figure 10.1 consists entirely of zero values. Note also that `row_ptr[5]` stores the starting location of a nonexisting row 4. This is for convenience, as some algorithms need to use the starting location of the next row to delineate the end of the current row. This extra marker gives a convenient way to locate the ending location of row 3.

As we discussed in Chapter 7, matrices are often used in solving a linear system of N equations of N variables in the form of $A \times X + Y = 0$, where A is an $N \times N$ matrix, X is a vector of N variables, and Y is a vector of N constant values. The objective is to solve for the X variable values that will satisfy all the questions. An intuitive approach is to Invert the matrix so that $X = A^{-1} \times (-Y)$. This can be done through methods such as Gaussian elimination for moderate-size arrays. While it is theoretically possible to solve equations represented in sparse matrices, the sheer size

	Row 0	Row 2	Row 3
Nonzero values data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers row_ptr[5]	{ 0, 2,	2, 5,	7 }

FIGURE 10.2

Example of CSR format.

and the number of zero elements of many sparse linear systems of equations can simply overwhelm this intuitive approach.

Instead, sparse linear systems can often be better solved with an iterative approach. When the sparse matrix A is positive-definite (i.e., $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all nonzero vectors $\mathbf{x}$ in $\mathbf{R}^n$), one can use conjugate gradient methods to iteratively solve the corresponding linear system with guaranteed convergence to a solution [Hest1952]. This is done by guessing a solution X, perform $A \times X + Y$, and see if the result is close to a 0 vector. If not, we can use a gradient vector formula to refine the guessed X and perform another iteration of $A \times X + Y$ using the refined X. The most time-consuming part of such an iterative approach is in the evaluation of $A \times X + Y$, which is a sparse matrix—vector multiplication and accumulation. Figure 10.3 shows a small example of matrix-vector multiplication, where A is a sparse matrix. The dark squares in A represent non-zero elements. In contrast, both X and Y are typically dense vectors. That is, most of the elements of X and Y hold non-zero values. Due to its importance, standardized library function interfaces have been created to perform this operation under the name SpMV (sparse matrix—vector multiplication). We will use SpMV to illustrate the important trade-offs between different storage formats of sparse computation.

A sequential implementation of SpMV based on CSR is quite straightforward, which is shown in Figure 10.4. We assume that the code has access to (1) num_rows, a function argument that specifies the number of rows in the sparse matrix, and (2) a floating-point data[] array and two integer row_ptr[] and x[] arrays as in Figure 10.3. There are only seven lines of code. Line 1 is a loop that iterates through all rows of the matrix, with each iteration calculating a dot product of the current row and the vector x.

In each row, line 2 first initializes the dot product to zero. It then sets up the range of data[] array elements that belong to the current row. The
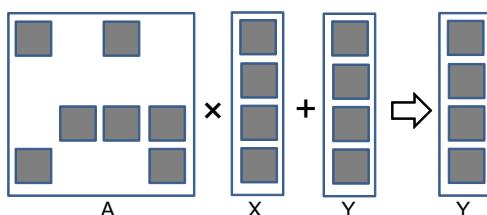
FIGURE 10.3

A small example of matrix—vector multiplication and accumulation.

starting and ending locations can be loaded from the row_ptr[] array. Figure 10.5 illustrates this for the small sparse matrix in Figure 10.1. For row = 0, row_ptr[row] is 0 and row_ptr[row + 1] is 2. Note that the two elements from row 0 reside in data[0] and data[1]. That is, row_ptr [row] gives the starting position of the current row and row_ptr[row + 1] gives the starting position of the next row, which is one after the ending position of the current row. This is reflected in the loop in line 5, where the loop index iterates from the position given by row_ptr[row] to row_ptr[row + 1] − 1.

The loop body in line 6 calculates the dot product for the current row. For each element, it uses the loop index elem to access the matrix element in data[elem]. It also uses elem to retrieve the column index for the element from col_index[elem]. This column index is then used to access the appropriate x element for multiplication. For example, the element in data[0] and data[1] are from column 0 (col_index[0] = 0) and column 2 (col_index[1] = 2). So the inner loop will perform the dot product for

```
1.    for (introw = 0; row < num_rows; row++) {
2.       float dot = 0;
3.       int row_start = row_ptr[row];
4.       int row_end =   row_ptr[row+1];

5.       for (intelem = row_start; elem < row_end; elem++) {
6.           dot += data[elem] * x[col_index[elem]];
         }

7.       y[row] += dot;

      }
```

FIGURE 10.4
A sequential loop that implements SpMV.
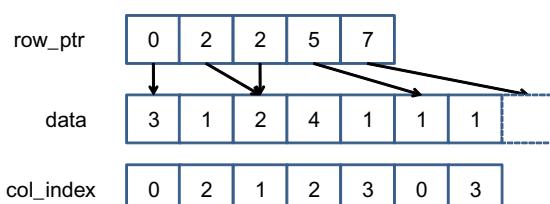
FIGURE 10.5
SpMV loop operating on the sparse matrix in Figure 10.1.

row 0 as `data[0]*x[0] + data[1]*x[2]`. Readers are encouraged to work out the dot product for other rows as an exercise.

CSR completely removes all zero elements from the storage. It does incur an overhead by introducing the `col_index` and `row_ptr` arrays. In our small example, where the number of zero elements are not much more than the non-zero elements, the overhead is slightly more than the space needed for the nonzero elements.

It should be obvious that any SpMV code will reflect the storage format assumed. Therefore, we will add the storage format to the name of a code to clarify the combination used. We will refer to the SpMV code in Figure 10.4 as sequential SpMV/CSR. With a good understanding of sequential SpMV/CSR, we are now ready to discuss parallel sparse computation.

10.2 PARALLEL SPMV USING CSR

Note that the dot product calculation for each row of the sparse matrix is independent of those of other rows. This is reflected in the fact that all iterations of the outer loop (line 1) in Figure 10.4 are logically independent of each other. We can easily convert this sequential SpMV/CSR into a parallel CUDA kernel by assigning each iteration of the outer loop to a thread, which is illustrated in Figure 10.6, where thread 0 calculates the dot product for row 0, thread 1 for row 1, and so on.

In a real sparse matrix computation, there are usually thousands to millions of rows, each of which contain tens to hundreds of nonzero elements. This makes the mapping shown in Figure 10.6 seem very appropriate: there are many threads and each thread has a substantial amount of work. We show a parallel SpMV/CSR in Figure 10.7.

It should be clear that the kernel looks almost identical to the sequential SpMV/CSR loop. The loop construct has been removed since it is replaced by the thread grid. All the other changes are very similar to the case of the vector addition kernel in Chapter 3. In line 2, the row index is calculated as the familiar expression `blockIdx.x * blockDim.x +`

Thread 0	3	0	1	0
Thread 1	0	0	0	0
Thread 2	0	2	4	1
Thread 3	1	0	0	1

FIGURE 10.6

Example of mapping threads to rows in parallel SpMV/CSR.

threadIdx.x. Also, due to the need to handle an arbitrary number of rows, line 3 checks if the row index of a thread exceeds the number of rows. This handles the situation where the number of rows is not a multiple of thread block size.

While the parallel SpMV/CSR kernel is quite simple, it has two major shortcomings. First the kernel does not make coalesced memory accesses. If readers examine Figure 10.5, it should be obvious that adjacent threads will be making simultaneous nonadjacent memory accesses. In our small example, threads 0, 1, 2, and 3 will access data[0], none, data[2], and data[5] in the first iteration of their dot product loop. They will then access data[1], none, data[3], and data[6] in the second iterations, and so on. It is obvious that these simultaneous accesses made by adjacent threads are not to adjacent locations. As a result, the parallel SpMV/CSR kernel does not make efficient use of memory bandwidth.

The second shortcoming of the SpMV/CSR kernel is that it can potentially have significant control flow divergence in all warps. The number of iterations taken by a thread in the dot product loop depends on the number of nonzero elements in the row assigned to the thread. Since the distribution of nonzero elements among rows can be random, adjacent rows can have a very different number of nonzero elements. As a result, there can be widespread control flow divergence in most or even all warps.

It should be clear that both the execution efficiency and memory bandwidth efficiency of the parallel SpMV kernel depends on the distribution of the input data matrix. This is quite different from most of the kernels we have presented so far. However, such data-dependent performance behavior is quite common in real-world applications. This is one of the

```
1.  __global__ void SpMV_CSR(int num_rows, float *data, int *col_index,
        int *row_ptr, float *x, float *y) {

2.      int row = blockIdx.x * blockDim.x + threadIdx.x;

3.      if (row < num_rows) {
4.        float dot = 0;
5.        int row_start = row_ptr[row];
6.        int row_end =   row_ptr[row+1];
7.        for (int elem = row_start; elem < row_end; elem++) {
8.          dot += data[elem] * x[col_index[elem]];
        }
9.      y[row] = dot;
      }

    }
```

FIGURE 10.7

A parallel SpMV/CSR kernel.

reasons why parallel SpMV is such an important parallel pattern—it is simple and yet it illustrates an important behavior in many complex parallel applications. We will discuss the important techniques in the next sections to address the two shortcomings of the parallel SpMV/CSR kernel.

10.3 PADDING AND TRANSPOSITION

The problems of noncoalesced memory accesses and control divergence can be addressed with data padding and transposition of matrix layout. The ideas were used in the ELL storage format, the name of which came from the sparse matrix package ELLPACK. A simple way to understand the ELL format is to start with the CSR format, as illustrated in Figure 10.8.

From a CSR representation, we first determine the rows with the maximal number of nonzero elements. We then add dummy (zero) elements to all other rows after the nonzero elements to make them the same length as the maximal rows. This makes the matrix a rectangular matrix. For our small sparse matrix example, we determine that row 2 has the maximal number of elements. We then add one zero element to row 0, three zero elements to row 1, and one zero element to row 3 to make all them the same length. These additional zero elements are shown as squares with an * in Figure 10.8. Now the matrix has become a rectangular matrix. Note that the col_index array also needs to be padded the same way to preserve their correspondence to the data values.

We can now lay out the padded matrix in column-major order. That is, we will place all elements of column 0 in consecutive locations, followed by all elements of column 1, and so on. This is equivalent to transposing the rectangular matrix and layout out the matrix in the row-major order of

CSR with Padding   Transposed
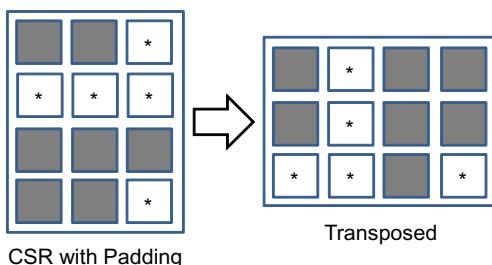
FIGURE 10.8

ELL storage format.

C. In terms of our small example, after the transposition, data[0] through data[3] now contain 3, *, 2, 1, the zero elements of all rows. This is illustrated in the bottom portion of Figure 10.9. col_index[0] through col_index[3] contain the first elements of all rows. Note that we no longer need the row_ptr since the beginning of row *i* is now simply data[i]. With the padded elements, it is also very easy to move from the current element of row *i* to the next element by simply adding the number of rows to the index. For example, the zero element of row 2 is in data[2] and the next element is in data[2 + 4] = data[6], where 4 is the number of rows in our small example.

Using the ELL format, we show a parallel SpMV/ELL kernel in Figure 10.10. The kernel receives a slightly different argument. It no longer needs the row_ptr. Instead, it needs an argument num_elem to know the number of elements in each row after padding.

A first observation is that the SpMV/ELL kernel code is simpler than SpMV/CSR. With padding, all rows are now of the same length. In the dot product loop in line 5, all threads can simply loop through the number of elements given by num_elem. As a result, there is no longer control flow divergence in warps: all threads now iterate exactly the same number of times in the dot product loop. In the case where a dummy element is used in the multiplication, since its value is zero, it will not affect the final result.

A second observation is that in the dot product loop body, each thread accesses its zero element in data[row] and then access its *i* element in data[row + i*num_rows]. As we have seen in Figure 10.10, all adjacent threads are now accessing adjacent memory locations, enabling memory coalescing and thus making more efficient use of memory bandwidth.
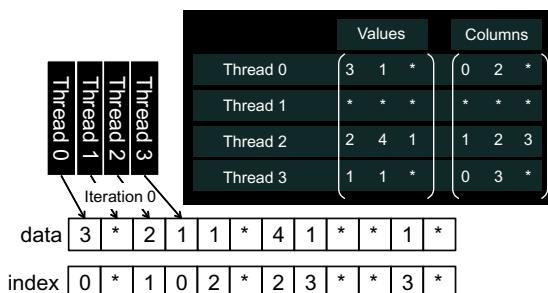
FIGURE 10.9

More details of our small example in ELL.

```
1.   __global__ void SpMV_ELL(intnum_rows, float *data, int *col_index,
     int num_elem, float *x, float *y) {

2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.       float dot = 0;
5.       for (int i = 0; i < num_elem; i++) {
6.         dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];
       }
7.       y[row] = dot;
     }
   }
```

FIGURE 10.10

A parallel SpMV/ELL kernel.

By eliminating control flow divergence and enabling memory coalescing, SpMV/ELL should run faster than SpMV/CSR. Furthermore, SpMV/ELL is simpler. This seems to make SpMV/ELL an all-winning approach. However, it is not quite so. Unfortunately, it does have its potential downside. In situations where one or a small number of rows have an exceedingly large number of nonzero elements, the ELL format will result in an excessive number of padded elements. These padded elements will take up storage, need to be fetched, and take part in calculations even though they do not contribute to the final result. With enough padded elements, an SpMV/ELL kernel can actually run more slowly than an SpMV/CSR kernel. This calls for a method to control the number of padded elements in an ELL representation.

10.4 USING HYBRID TO CONTROL PADDING

The root of the problem with excessive padding in ELL is that one or a small number of rows have an exceedingly large number of nonzero elements. If we have a mechanism to "take away" some elements from these rows, we can reduce the number of padded elements in ELL. The coordinate (COO) format provides such a venue.

The COO format is illustrated in Figure 10.11, where each nonzero element is stored with both its column index and row index. We have both col_index and row_index arrays to accompany the data array. For example A[0,0] of our small example is now stored with both its column index (0 in col_index[0]) and its row index (0 in row_index[0]). With COO format, one can look at any element in the storage and know where the

		Row 0		Row 2			Row 3	
Nonzero values data[7]	{	3,	1,	2,	4,	1,	1,	1 }
Column indices col_index[7]	{	0,	2,	1,	2,	3,	0,	3 }
Row indices row_index[7]	{	0,	0,	2,	2,	2,	3,	3 }

FIGURE 10.11

Example of COO format.

Nonzero values data[7]	{ 1 1, 2, 4, 3, 1 1 }
Column indices col_index[7]	{ 0 2, 1, 2, 0, 3, 3 }
Row indices row_index[7]	{ 3 0, 2, 2, 0, 2, 3 }

FIGURE 10.12

Reordering COO format.

nonzero element came from in the original sparse matrix. Like the ELL format, there is no need for row_ptr since each element self-identifies its column and row index.

While the COO format does come with the cost of additional storage for the row_index array, it also comes with the additional benefit of flexibility. We can arbitrarily reorder the elements in a COO format without losing any information as long as we reorder the data, col_index, and row_index the same way. This is illustrated using our small example in Figure 10.12.

In Figure 10.12, we have reordered the elements of data, col_index, and row_index. Now data[0] actually contains an element from row 0 and column 3 of the small sparse matrix. Because we have also shifted the row index and column index values along with the data value, we can correctly identify this element in the original sparse matrix. Readers may ask why we would want to reorder these elements. Such reordering would disturb the locality and sequential patterns that are important for efficient use of memory bandwidth.

The answer lies in an important use case for the COO format. It can be used to curb the length of the CSR or ELL format. First, we will make an important observation. In the COO format, we can process the elements in any order we want. For each element in data[i], we can simply perform a y[row_index[i]] += data[i] * x[col_index[i]] operation. If we make sure somehow we perform this operation for all elements of data, we will calculate the correct final answer.

More importantly, we can take away some of the elements from the rows with an exceedingly large number of nonzero elements and place them into a separate COO format. We can use either CSR or ELL to perform SpMV on the remaining elements. With excess elements removed from the extra-long rows, the number of padded elements for other rows can be significantly reduced. We can then use a SpMV/COO to finish the job. This approach of employing two formats to collaboratively complete a computation is often referred to as a *hybrid method*.

Let's illustrate a hybrid ELL and COO method for SpMV using our small sparse matrix, as shown in Figure 10.13. We see that row 2 has the most number of nonzero elements. We remove the last nonzero element of row 2 from the ELL representation and move it into a separate COO representation. By removing the last element of row 2, we reduce the maximal number of nonzero elements among all rows in the small sparse matrix from 3 to 2. As shown in Figure 10.13, we reduce the number of padded elements from 5 to 2. More importantly, all threads now only need to take two iterations rather than three. This can give a 50% acceleration to the parallel execution of the SpMV/ELL kernel.

A typical way of using an ELL-COO hybrid method is for the host to convert the format from something like a CSR format into ELL. During the conversion, the host removes some nonzero elements from the rows with an exceedingly large number of nonzero elements. The host places these elements into a COO representation. The host then transfers the ELL representation of the data to a device. When the device completes the SpMV/ELL kernel, it transfers the y values back to the host. These values are missing the contributions from the elements in the COO representation.
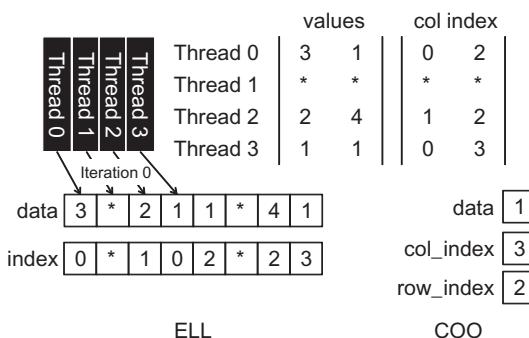
FIGURE 10.13

Our small example in ELL and COO hybrid.

The host performs a sequential SpMV/COO on the COO elements and finishes their contributions to the y element values.

The user may question whether the additional work done by the host to separate COO elements from an ELL format could incur too much overhead. The answer is it depends. In situations where a sparse matrix is only used in one SpMV calculation, this extra work can indeed incur significant overhead. However, in many real-world applications, the SpMV is performed on the same sparse kernel repeated in an iterative solver. In each iteration of the solver, the x and y vectors vary but the sparse matrix remains the same since its elements correspond to the coefficients of the linear system of equations being solved and these coefficients do not change from iteration to iteration. So, the work done to produce both the ELL and COO representation can be amortized across many iterations. We will come back to this point in the next section.

In our small example, the device finishes the SpMV/ELL kernel on the ELL portion of the data. The y values are then transferred back to the host. The host then adds the contribution of the COO element with the operation $y[2] += data[0]*x[col\_index[0]] = 1*x[3]$. Note that there are in general multiple nonzero elements in the COO format. So, we expect that the host code to be a loop as shown in Figure 10.14.

The loop is extremely simple. It iterates through all the data elements and performs the multiply and accumulate operations on the appropriate x and y elements using the accompanying col_index and row_index elements. We will not present a parallel SpMV/COO kernel. It can be easily constructed using each thread to process a portion of the data elements and use an atomic operation to accumulate the results into y elements. This is because the threads are no longer mapped to a particular row. In fact, many rows will likely be missing from the COO representation; only the rows that have an exceedingly large number of nonzero elements will have elements in the COO representation. Therefore, it is better just to have each thread to take a portion of the data element and use an atomic operation to make sure that none of the threads will trample the contribution of other threads.

```
1.    for (int i = 0; i < num_elem; row++)
2.       y[row_index[i]] += data[i] * x[col_index[i]];
```

FIGURE 10.14

A sequential loop that implements SpMV/COO.

The hybrid SpMV/ELL-COO method is a good illustration of productive use of both CPUs and GPUs in a heterogeneous computing system. The CPU can perform SpMV/COO fast using its large cache memory. The GPU can perform SpMV/ELL fast using its coalesced memory accesses and large number of hardware execution units. The removal of some elements from the ELL format is a form of regularization technique: it reduces the disparity between long and short rows and makes the workload of all threads more uniform. Such improved uniformity results in benefits such as less control divergence in a SpMV/CSR kernel or less padding in a SpMV/ELL kernel.

10.5 SORTING AND PARTITIONING FOR REGULARIZATION

While COO helps to regulate the amount of padding in an ELL representation, we can further reduce the padding overhead by sorting and partitioning the rows of a sparse matrix. The idea is to sort the rows according to their length, say from the longest to the shortest. This is illustrated with our small sparse matrix in Figure 10.15. Since the sorted matrix looks largely like a triangular matrix, the format is often referred to as jagged diagonal storage (JDS). As we sort the rows, we typically maintain an additional jds_row_index array that preserves the original index of the row. For CSR, this is similar to the row_ptr array in that there is one element per row. Whenever we exchange two rows in the sorting process, we also exchange the corresponding elements of the jds_row_index array. This way, we can always keep track of the original position of all rows.

Once a sparse matrix is in JDS format, we can partition the matrix into sections of rows. Since the rows have been sorted, all rows in a section will likely have a more or less uniform number of nonzero elements. In

CSR   JDS
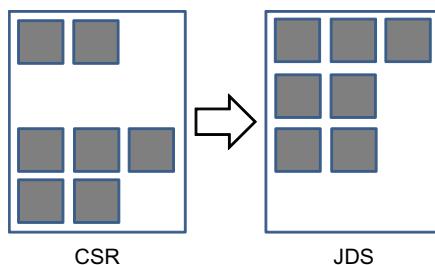
FIGURE 10.15

Sorting rows according to their length.

Figure 10.15, we can divide the small matrix into three sections: the first section consists of the one row that has three elements, the second section consists of the two rows with two elements each, and the third section consists of one row without any element. We can then generate ELL representation for each section. Within each section, we only need to pad the rows to match the row with the maximal number of elements in that section. This would reduce the number of padded elements. In our example, we do not even need to pad within any of the three sections. We can then transpose each section independently and launch a separate kernel on each section. In fact, we do not even need to launch a kernel for the section of rows with no nonzero elements.

Figure 10.16 shows a JDS-ELL representation of our small sparse matrix. It assumed the same sorting and partitioning results shown in Figure 10.15. Out of the three sections, the first section has only one row so the transposed layout is the same as the original. The second section is a 2×2 matrix and has been transposed. The third section consists of row 1, which does not have any nonzero element. This is reflected in the fact that its starting location and the next section's starting position are identical.

We will not show a SpMV/JDS kernel. The reason is that we would be just using either an SpMV/CSR kernel on each section of the CSR, or a SpMV/ELL kernel on each section of the ELL after padding. The host code required to create a JDS representation and to launch SpMV kernels on each section of the JDS representation is left as an exercise.

Note that we want each section to have a large number of rows so that its kernel launch will be worthwhile. In the extreme cases where a very small number of rows have an extremely large number of nonzero elements, we can still use the COO hybrid with JDS to allow us to have more rows in each section.
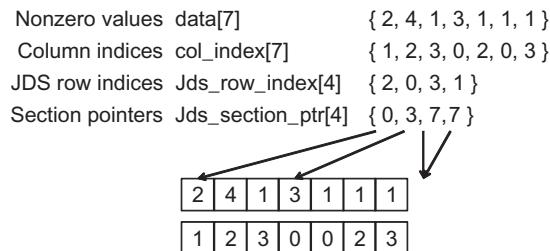
FIGURE 10.16

JDS format and sectioned ELL.

Once again readers should ask whether sorting rows will result into incorrect solutions to the linear system of equations. Recall that we can freely reorder equations of a linear system without changing the solution. As long as we reorder the y elements along with the rows, we are effectively reordering the equations. Therefore, we will end up with the correct solution. The only extra step is to reorder the final solution back to the original order using the `jds_row_index` array.

The other question is whether sorting will incur significant overhead. The answer is similar to what we saw in the hybrid method. As long as the SpMV/JDS kernel is used in an iterative solver, one can afford to perform such sorting as well as the reordering of the final solution x elements and amortize the cost among many iterations of the solver.

In more recent devices, the memory coalescing hardware has relaxed the address alignment requirement. This allows one to simply transpose a JDS-CSR representation. Note that we do need to adjust the `jds_section_ptr` array after transposition. This further eliminates the need to pad rows in each section. As memory bandwidth becomes increasingly the limiting factor of performance, eliminating the need to store and fetch padded elements can be a significant advantage. Indeed, we have observed that while sectioned JDS-ELL tends to give the best performance on older CUDA devices, transposed JDS-CSR tends to give the best performance on Fermi and Kepler.

We would like to make an additional remark on the performance of sparse matrix computation as compared to dense matrix computation. In general, the FLOPS rating achieved by either CPUs or GPUs are much lower for sparse matrix computation than for dense matrix computation. This is especially true for SpMV, where there is no data reuse in the sparse matrix. The CGMA value (see Chapter 5) is essentially 1, limiting the achievable FLOPS rate to a small fraction of the peak performance. The various formats are important for both CPUs and GPUs since both are limited by memory bandwidth when performing SpMV. Many folks have been surprised by the low FLOPS rating of this type of computation on both CPUs and GPUs in the past. After reading this chapter, one should no longer be surprised.

10.6 SUMMARY

In this chapter, we presented sparse matrix computation as an important parallel pattern. Sparse matrices are important in many real-world

applications that involve modeling complex phenomenon. Furthermore, sparse matrix computation is a simple example of data-dependent performance behavior of many large real-world applications. Due to the large amount of zero elements, compaction techniques are used to reduce the amount of storage, memory accesses, and computation performed on these zero elements. Unlike most other kernels presented in this book so far, the SpMV kernels are sensitive to the distribution of nonzero elements in the sparse matrices. Not only can the performance of each kernel vary significantly across matrices, their relative merit can also change significantly. Using this pattern, we introduce the concept of regularization using hybrid methods and sorting/partitioning. These regularization methods are used in many real-world applications. Interestingly, some of the regularization techniques reintroduce zero elements into the compacted representations. We use hybrid methods to mitigate the pathological cases where we could introduce too many zero elements. Readers are referred to [Bell2009] and encouraged to experiment with different sparse data sets to gain more insight into the data-dependent performance behavior of the various SpMV kernels presented in this chapter.

10.7 EXERCISES

10.1. Complete the host code to produce the hybrid ELL-COO format, launch the ELL kernel on the device, and complete the contributions of the COO elements.

10.2. Complete the host code for producing JDS-ELL and launch one kernel for each section of the representation.

10.3. Consider the following sparse matrix:

```
1 0 7 0
0 0 8 0
0 4 3 0
2 0 0 1
```

Represent it in each of the following formats: (a) COO, (b) CSR, and (c) ELL.

10.4. Given a sparse matrix of integers with m rows, n columns, and z nonzeros, how many integers are needed to represent the matrix in (a) COO, (b) CSR, and (c) ELL. If the information provided is not enough, indicate what information is missing.

References

Hestenes, M., & Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, *49*(6).

Bell, N., & Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors, Proceedings of the ACM Conference on High-Performance Computing Networking Storage and Analysis (SC'09), 2009.

Application Case Study: Advanced MRI Reconstruction

11

Application case studies teach computational thinking and practical programming techniques in a concrete manner. They also help demonstrate how the individual techniques fit into a top-to-bottom development process. Most importantly, they help us to visualize the practical use of these techniques in solving problems. In this chapter, we start with the background and problem formulation of a relatively simple application. We show that parallel execution not only speeds up the existing approaches, but also allows applications experts to pursue approaches that are known to provide benefit but were previously ignored due to their excessive computational requirements. We then use an example algorithm and its implementation source code from such an approach to illustrate how a developer can systematically determine the kernel parallelism structure, assign variables into CUDA memories, steer around limitations of the hardware, validate results, and assess the impact of performance improvements.

235

11.1 APPLICATION BACKGROUND

Magnetic resonance imaging (MRI) is commonly used by the medical community to safely and noninvasively probe the structure and function of biological tissues in all regions of the body. Images that are generated using MRI have made profound impact in both clinical and research settings. MRI consists of two phases: acquisition (scan) and reconstruction. During the acquisition phase, the scanner samples data in the *k*-space domain (i.e., the spatial-frequency domain or Fourier transform domain) along a predefined trajectory. These samples are then transformed into the desired image during the reconstruction phase.

The application of MRI is often limited by high noise levels, significant imaging artifacts, and/or long data acquisition times. In clinical settings, short scan times not only increase scanner throughput but also reduce patient discomfort, which tends to mitigate motion-related artifacts. High image resolution and fidelity are important because they enable earlier detection of pathology, leading to improved prognoses for patients. However, the goals of short scan time, high resolution, and high signal-to-noise ratio (SNR) often conflict; improvements in one metric tend to come at the expense of one or both of the others. One needs new technological breakthroughs to be able to simultaneously improve on all of three dimensions. This study presents a case where massively parallel computing provides such a breakthrough.

Readers are referred to MRI textbooks such as Liang and Lauterbur [LL1999] for the physics principles behind MRI. For this case study, we will focus on the computational complexity in the reconstruction phase and how the complexity is affected by the *k*-space sampling trajectory. The *k*-space sampling trajectory used by the MRI scanner can significantly affect the quality of the reconstructed image, the time complexity of the reconstruction algorithm, and the time required for the scanner to acquire the samples. Equation (11.1) below shows a formulation that relates the *k*-space samples to the reconstructed image for a class of reconstruction methods.

$$\hat{m}(\mathrm{r}) = \sum_j W(\mathrm{k}_j)s(\mathrm{k}_j)e^{i2\pi\mathrm{k}_j\cdot\mathrm{r}} \qquad (11.1)$$

In Eq. (11.1), $m(\mathbf{r})$ is the reconstructed image, $s(\mathbf{k})$ is the measured *k*-space data, and $W(\mathbf{k})$ is the weighting function that accounts for nonuniform sampling. That is, $W(\mathbf{k})$ decreases the influence of data from *k*-space regions where a higher density of sample points are taken. For this class of reconstructions, $W(\mathbf{k})$ can also serve as an apodization function that reduces the influence of noise and reduces artifacts due to finite sampling.

If data is acquired at uniformly spaced Cartesian grid points in the k-space under ideal conditions, then the $W(\mathbf{k})$ weighting function is a constant and can thus be factored out of the summation in Eq. (11.1). As a result, the reconstruction of $m(\mathbf{r})$ becomes an inverse fast Fourier transform (FFT) on $s(\mathbf{k})$, an extremely efficient computation method. A collection of data measured at such uniformed spaced Cartesian grid points is referred to as a *Cartesian scan trajectory*, depicted in Figure 11.1(a). In practice, Cartesian scan trajectories allow straightforward implementation on scanners and are widely used in clinical settings today.

Although the inverse FFT reconstruction of Cartesian scan data is computationally efficient, non-Cartesian scan trajectories often have advantages in reduced sensitivity to patient motion, better ability to provide self-calibrating field inhomogeneity information, and reduced requirements on scanner hardware performance. As a result, non-Cartesian scan trajectories like spirals (shown in Figure 11.1c), radial lines (projection imaging), and rosettes have been proposed to reduce motion-related artifacts and address scanner hardware performance limitations. These improvements have recently allowed the reconstructed image pixel values to be used for measuring subtle phenomenon such as tissue chemical anomalies before they become anatomical pathology. Figure 11.2 shows such a measurement that generates a map of sodium, a heavily regulated substance in normal human tissues. The information can be used to track tissue health in stroke and cancer treatment processes. The variation or shifting of sodium concentration gives early signs of disease development or tissue death. For example, the sodium map of a human brain shown in Figure 11.2 can be
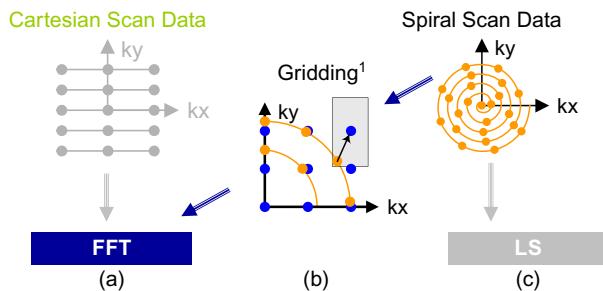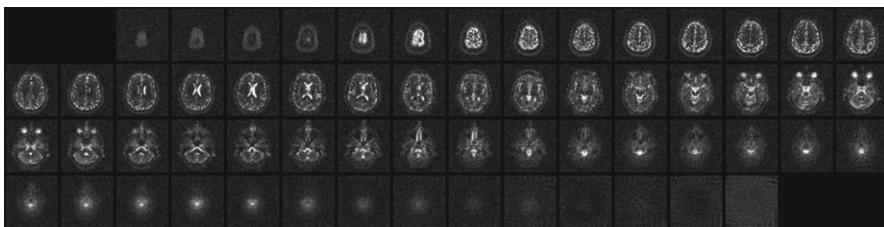
FIGURE 11.1

Scanner k-space trajectories and their associated reconstruction strategies: (a) Cartesian trajectory with FFT reconstruction, (b) spiral (or non-Cartesian trajectory in general) followed by gridding to enable FFT reconstruction, and (c) spiral (non-Cartesian) trajectory with linear solver–based reconstruction.

used to give an early indication of brain tumor tissue responsiveness to chemotherapy protocols, enabling individualized medicine. Because sodium is much less abundant than water molecules in human tissues, a reliable measure of sodium levels requires a higher SNR through a higher number of samples and needs to control the extra scan time with non-Cartesian scan trajectories.

Image reconstruction from non-Cartesian trajectory data presents both challenges and opportunities. The main challenge arises from the fact that the exponential terms are no longer uniformly spaced; the summation does not have the form of an FFT anymore. Therefore, one can no longer perform reconstruction by directly applying an inverse FFT to the k-space samples. In a commonly used approach called *gridding*, the samples are first interpolated onto a uniform Cartesian grid and then reconstructed using the FFT (see Figure 11.1b). For example, a convolution approach to gridding takes a k-space data point, convolves it with a gridding kernel, and accumulates the results on a Cartesian grid. Convolution is quite computationally intensive. Accelerating gridding computation on many-core processors facilitates the application of the current FFT approach to non-Cartesian trajectory data. Since we have already studied the convolution pattern in Chapter 8 and will be examining a convolution-style computation in Chapter 12, we will not cover it here.

In this chapter, we will cover an iterative, statistically optimal image reconstruction method that can accurately model imaging physics and

Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

FIGURE 11.2

The use of non-Cartesian k-space sample trajectory and accurate linear solver−based reconstruction enables new MRI modalities with exciting medial applications. The improved SNR enables reliable collection of in-vivo concentration data on a chemical substance such as sodium in human tissues. The variation or shifting of sodium concentration gives early signs of disease development or tissue death. For example, the sodium map of a human brain shown in this Figure can be used to give early indication of brain tumor tissue responsiveness to chemo-therapy protocols, enabling individualized medicine.

bound the noise error in each image pixel value. However, such iterative reconstructions have been impractical for large-scale 3D problems due to their excessive computational requirements compared to gridding. Recently, these reconstructions have become viable in clinical settings when accelerated on GPUs. In particular, we will show that an iterative reconstruction algorithm that used to take hours using a high-end sequential CPU now takes only minutes using both CPUs and GPUs for an image of moderate resolution, a delay acceptable in clinical settings.

11.2 ITERATIVE RECONSTRUCTION

Haldar, et al [HHB 2007] proposed a linear solver−based iterative reconstruction algorithm for non-Cartesian scan data, as shown in Figure 11.1(c). The algorithm allows for explicit modeling and compensation for the physics of the scanner data acquisition process, and can thus reduce the artifacts in the reconstructed image. It is, however, computationally expensive. The reconstruction time on high-end sequential CPUs has been hours for moderate-resolution images and thus impractical in clinical use. We use this as an example of innovative methods that have required too much computation time to be considered practical. We will show that massive parallelism can reduce the reconstruction time to the order of a minute so that one can deploy the new MRI modalities such as sodium imaging in clinical settings.

Figure 11.3 shows a solution of the quasi-Bayesian estimation problem formulation of the iterative linear solver−based reconstruction approach,
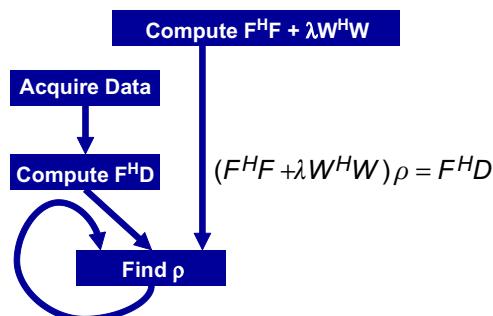
FIGURE 11.3

An iterative linear solver−based approach to reconstruction of no-Cartesian *k*-space sample data.

where ρ is a vector containing voxel values for the reconstructed image, F is a matrix that models the physics of imaging process, D is a vector of data samples from the scanner, and W is a matrix that can incorporate prior information such as anatomical constraints. In clinical settings, the anatomical constraints represented in W are derived from one or more high-resolution, high-SNR water molecule scans of the patient. These water molecule scans reveal features such as the location of anatomical structures. The matrix W is derived from these reference images. The problem is to solve for ρ given all the other matrices and vectors.

On the surface, the computational solution to the problem formulation in Figure 11.3 should be very straightforward. It involves matrix−matrix multiplications and addition ($F^H F + \lambda W^H W$), matrix−vector multiplication ($F^H D$), matrix inversion ($F^H F + \lambda W^H W)^{-1}$, and finally matrix−matrix multiplication (($F^H F + \lambda W^H W)^{-1} \times F^H D$). However, the sizes of the matrices make this straightforward approach extremely time consuming. F^H and F are 3D matrices of which the dimensions are determined by the resolution of the reconstructed image ρ. Even in a modest resolution 128^3-voxel reconstruction, there are 128^3 columns in F with N elements in each column where N is the number of k-space samples used. Obviously, F is extremely large.

The sizes of the matrices involved are so large that the matrix operations involved in a direct solution of the equation in Figure 11.3 are practically intractable. An iterative method for matrix inversion, such as the conjugate gradient (CG) algorithm, is therefore preferred. The conjugate gradient algorithm reconstructs the image by iteratively solving the equation in Figure 11.3 for ρ. During each iteration, the CG algorithm updates the current image estimate ρ to improve the value of the quasi-Bayesian cost function. The computational efficiency of the CG technique is largely determined by the efficiency of matrix−vector multiplication operations involving $F^H F + \lambda W^H W$ and ρ, as these operations are required during each iteration of the CG algorithm.

Fortunately, matrix W often has a sparse structure that permits efficient multiplication by $W^H W$, and matrix $F^H F$ is Toeplitz that enables efficient matrix−vector multiplication via the FFT. Stone et al. [SHT2008] present a GPU-accelerated method for calculating Q, a data structure that allows us to quickly calculate matrix−vector multiplication involving $F^H F$ without actually calculating $F^H F$ itself. The calculation of Q can take days on a high-end CPU core. It only needs to be done once for a given trajectory and can be used for multiple scans.

The matrix−vector multiply to calculate $F^H D$ takes about one order of magnitude less time than Q but can still take about three hours for a

128^3-voxel reconstruction on a high-end sequential CPU. Since F^{H}D needs to be computed for every image acquisition, it is desirable to reduce the computation time of F^{H}D to minutes.[1] We will show the details of this process. As it turns out, the core computational structure of Q is identical to that of F^{H}D. As a result, the same methodology can be used to accelerate the computation of both.

The "find ρ" step in Figure 11.3 performs the actual CG based on F^{H}D. As we explained earlier, precalculation of Q makes this step much less computationally intensive than F^{H}D, and accounts for only less than 1% of the execution of the reconstruction of each image on a sequential CPU. As a result, we will leave it out of the parallelization scope and focus on F^{H}D in this chapter. We will, however, revisit its status at the end of the chapter.

11.3 COMPUTING F^{H}D

Figure 11.4 shows a sequential C implementation of the computations for the core step of computing a data structure for multiplications with F^{H}F (referred to as Q computation in Figure 11.4a) without explicitly calculating F^{H}F, and that for F^{H}D (Figure 11.4b). It should be clear from a quick glance at Figures 11.4(a) and (b) that the core steps of Q and F^{H}D have identical structures. Both computations start with an outer loop that encloses an inner loop. The only differences are the particular calculation done in each loop body and the fact that the core step of Q involves a much larger m, since it implements a matrix−matrix multiplication as opposed to a matrix−vector multiplication, thus it incurs a much longer execution time. Thus, it suffices to discuss one of them. We will focus on F^{H}D, since this is the one that will need to be run for each image being reconstructed.

A quick glance at Figure 11.4(b) shows that the C implementation of F^{H}D is an excellent candidate for acceleration on the GPU because it exhibits substantial data parallelism. The algorithm first computes the real and imaginary components of Mu (rMu and iMu) at each sample point in the k-space. It then computes the real and imaginary components of F^{H}D at each voxel in the image space. The value of F^{H}D at any voxel depends on the values of all k-space sample points. However, no voxel elements of

---

[1]Note that the F^{H}D computation can be approximated with gridding and can run in a few seconds, with perhaps reduced quality of the final reconstructed image.

$F^H D$ depend on any other elements of $F^H D$. Therefore, all elements of $F^H D$ can be computed in parallel. Specifically, all iterations of the outer loop can be done in parallel and all iterations of the inner loop can be done in parallel. The calculations of the inner loop, however, have a dependence on the calculation done by the preceding statements in the same iteration of the outer loop.

Despite the algorithm's abundant inherent parallelism, potential performance bottlenecks are evident. First, in the loop that computes the elements of $F^H D$, the ratio of floating-point operations to memory accesses is at best 3:1 and at worst 1:1. The best case assumes that the `sin` and `cos` trigonometry operations are computed using the five-element Taylor series that requires 13 and 12 floating-point operations, respectively. The worst case assumes that each trigonometric operation is computed as a single operation in hardware. As we have seen in Chapter 5, a floating-point to memory access ratio of 16:1 or more is needed for the kernel to not be limited by memory bandwidth. Thus, the memory accesses will clearly limit the performance of the kernel unless the ratio is drastically increased.

Second, the ratio of floating-point arithmetic to floating-point trigonometry functions is only 13:2. Thus, a GPU-based implementation must tolerate or avoid stalls due to long-latency `sin` and `cos` operations.

```
for (m = 0; m < M; m++) {

  phiMag[m] = rPhi[m]*rPhi[m] +
              iPhi[m]*iPhi[m];

  for (n = 0; n < N; n++) {
    expQ = 2*PI*(kx[m]*x[n] +
                 ky[m]*y[n] +
                 kz[m]*z[n]);

    rQ[n] +=phiMag[m]*cos(expQ);
    iQ[n] +=phiMag[m]*sin(expQ);
  }
}
        (a) Q computation
```

```
for (m = 0; m < M; m++) {

  rMu[m] = rPhi[m]*rD[m] +
           iPhi[m]*iD[m];
  iMu[m] = rPhi[m]*iD[m] –
           iPhi[m]*rD[m];

  for (n = 0; n < N; n++) {
    expFhD = 2*PI*(kx[m]*x[n] +
                   ky[m]*y[n] +
                   kz[m]*z[n]);

    cArg = cos(expFhD);
    sArg = sin(expFhD);

    rFhD[n] +=  rMu[m]*cArg –
                iMu[m]*sArg;
    iFhD[n] +=  iMu[m]*cArg +
                rMu[m]*sArg;
  }
}     (b) F^H d computation
```

FIGURE 11.4

Computation of (a) Q and (b) $F^H D$.

Without a good way to reduce the cost of trigonometry functions, the performance will likely be dominated by the time spent in these functions.

We are now ready to take the steps in converting $F^H D$ from sequential C code to a CUDA kernel.

Step 1: Determine the Kernel Parallelism Structure

The conversion of a loop into a CUDA kernel is conceptually straightforward. Since all iterations of the outer loop of Figure 11.4(b) can be executed in parallel, we can simply convert the outer loop into a CUDA kernel by mapping its iterations to CUDA threads. Figure 11.5 shows a kernel from such a straightforward conversion. Each thread implements an iteration of the original outer loop. That is, we use each thread to calculate the contribution of one k-space sample to all $F^H D$ elements. The original outer loop has *M* iterations, and *M* can be in the millions. We obviously need to have multiple thread blocks to generate enough threads to implement all these iterations.

To make performance tuning easy, we declare a constant FHD_THREADS_PER_BLOCK that defines the number of threads in each thread block when we invoke the cmpFHd kernel. Thus, we will use M/FHD_THREADS_PER_BLOCK for the grid size (in terms of number of blocks) and FHD_THREADS_PER_BLOCK for block size (in terms of number of threads) for kernel invocation. Within the kernel, each thread calculates

```
__global__ void cmpFhD(float* rPhi, iPhi, rD, iD,
    kx, ky, kz, x, y, z, rMu, iMu, rFhD, iFhD, int N) {

  int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
  iMu[m] = rPhi[m]*iD[m] – iPhi[m]*rD[m];

  for (n = 0; n < N; n++) {
    floatexpFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);

    floatcArg = cos(expFhD);  floatsArg = sin(expFhD);

    rFhD[n] +=  rMu[m]*cArg – iMu[m]*sArg;
    iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
  }
}
```

FIGURE 11.5

First version of the $F^H D$ kernel. The kernel will not execute correctly due to conflicts between threads in writing into rFhD and iFhD arrays.

the original iteration of the outer loop that it is assigned to cover using the formula `blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x`. For example, assume that there are 65,536 *k*-space samples and we decided to use 512 threads per block. The grid size at kernel innovation would be $65,536 \div 512 = 128$ blocks. The block size would be 512. The calculation of `m` for each thread would be equivalent to `blockIdx.x*512 + threadIdx`.

While the kernel of Figure 11.5 exploits ample parallelism, it suffers from a major problem: all threads write into all `rFhD` and `iFhD` voxel elements. This means that the kernel must use atomic operations in the global memory in the inner loop to keep threads from trashing each other's contributions to the voxel value. This can seriously affect the performance of the kernel. Note that as is, the code will not even execute correctly since no atomic operation is used. We need to explore other options.

The other option is to use each thread to calculate one `FhD` value from all *k*-space samples. To do so, we need to first swap the inner loop and the outer loop so that each of the new outer loop iterations processes one `FhD` element. That is, each of the new outer loop iterations will execute the new inner loop that accumulates the contribution of all *k*-space samples to the `FhD` element handled by the outer loop iteration. This transformation to the loop structure is called *loop interchange*. It requires a perfectly nested loop, meaning that there is no statement between the outer `for` loop statement and the inner `for` loop statement. This is, however, not true for the F^HD code in Figure 11.4(b). We need to find a way to move the calculation of `rMu` and `iMu` elements out of the way.

From a quick inspection of Figure 11.6(a), which is a replicate of Figure 11.4(b), we see that the F^HD calculation can be split into two separate loops, as shown in Figure 11.6(b), using a technique called *loop fission* or *loop splitting*. This transformation takes the body of a loop and splits it into two loops. In the case of F^HD, the outer loop consists of two parts: the statements before the inner loop and the inner loop. As shown in Figure 11.6(b), we can perform loop fission on the outer loop by placing the statements before the inner loop into a loop and the inner loop into a second loop. The transformation changes the relative execution order of the two parts of the original outer loop. In the original outer loop, both parts of the first iteration execute before the second iteration. After fission, the first part of all iterations will execute; they are then followed by the second part of all iterations. Readers should be able to verify that this change of execution order does not affect the execution results for F^HD. This is because the execution of the first part of each iteration does not depend on the result of the second part of any preceding iterations of the

```
for (m = 0; m < M; m++) {              for (m = 0; m < M; m++) {

  rMu[m] = rPhi[m]*rD[m] +               rMu[m] = rPhi[m]*rD[m] +
           iPhi[m]*iD[m];                         iPhi[m]*iD[m];
  iMu[m] = rPhi[m]*iD[m] -               iMu[m] = rPhi[m]*iD[m] -
           iPhi[m]*rD[m];                         iPhi[m]*rD[m];
                                       }
  for (n = 0; n < N; n++) {            for (m = 0; m < M; m++) {
    expFhD = 2*PI*(kx[m]*x[n] +          for (n = 0; n < N; n++) {
                   ky[m]*y[n] +            expFhD = 2*PI*(kx[m]*x[n] +
                   kz[m]*z[n]);                          ky[m]*y[n] +
                                                         kz[m]*z[n]);
    cArg = cos(expFhD);
    sArg = sin(expFhD);                   cArg = cos(expFhD);
                                          sArg = sin(expFhD);
    rFhD[n] +=  rMu[m]*cArg -
                iMu[m]*sArg;              rFhD[n] +=  rMu[m]*cArg -
    iFhD[n] +=  iMu[m]*cArg +                         iMu[m]*sArg;
                rMu[m]*sArg;              iFhD[n] +=  iMu[m]*cArg +
  }                                                   rMu[m]*sArg;
}                                         }
    (a) F H D computa    tion         }    (b) after loop fission
```

FIGURE 11.6

(a) Loop fission on the F^{H}D computation and (b) after loop fission.

original outer loop. Loop fission is a transformation often done by advanced compilers that are capable of analyzing the (lack of) dependence between statements across loop iterations.

With loop fission, the F^{H}D computation is now done in two steps. The first step is a single-level loop that calculates the rMu and iMu elements for use in the second loop. The second step corresponds to the loop that calculates the FhD elements based on the rMu and iMu elements calculated in the first step. Each step can now be converted into a CUDA kernel. The two CUDA kernels will execute sequentially with respect to each other. Since the second loop needs to use the results from the first loop, separating these two loops into two kernels that execute in sequence does not sacrifice any parallelism.

The cmpMu() kernel in Figure 11.7 implements the first loop. The conversion of the first loop from sequential C code to a CUDA kernel is straightforward: each thread implements one iteration of the original C code. Since the *M* value can be very big, reflecting the large number of *k*-space samples, such a mapping can result in a large number of threads. With 512 threads in each block, we will need to use multiple blocks to

allow the large number of threads. This can be accomplished by having a number of threads in each block, specified by MU_THREADS_PER_BLOCK in Figure 11.4(c), and by employing M/MU_THREADS_PER_BLOCK blocks needed to cover all *M* iterations of the original loop. For example, if there are 65,536 *k*-space samples, the kernel could be invoked with a configuration of 512 threads per block and $65,536 \div 512 = 128$ blocks. This is done by assigning 512 to MU_THREADS_PER_BLOCK and using MU_THREADS_PER_BLOCK as the block size and M/MU_THREADS_PER_BLOCK as the grid size during kernel innovation.

Within the kernel, each thread can identify the iteration assigned to it using its blockIdx and threadIdx values. Since the threading structure is one dimensional, only blockIdx.x and threadIdx.x need to be used. Because each block covers a section of the original iterations, the iteration covered by a thread is blockIdx.x*MU_THREADS_PER_BLOCK + threadIdx. For example, assume that MU_THREADS_PER_BLOCK = 512. The thread with blockIdx.x = 0 and threadIdx.x = 37 covers the 37th iteration of the original loop, whereas the thread with blockIdx.x = 5 and threadIdx. x = 2 covers the 2,562nd ($5 \times 512 + 2$) iteration of the original loop. Using this iteration number to access the Mu, Phi, and D arrays ensures that the arrays are covered by the threads in the same way they were covered by the iterations of the original loop. Because every thread writes into its own Mu element, there is no potential conflict between any of these threads.

Determining the structure of the second kernel requires a little more work. An inspection of the second loop in Figure 11.6(b) shows that there are at least three options in designing the second kernel. In the first option, each thread corresponds to one iteration of the inner loop. This option creates the most number of threads and thus exploits the largest amount of parallelism. However, the number of threads would be $N \times M$, with both N in the range of millions and M in the range of hundreds of thousands. Their product would result in too many threads in the grid.

```
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)
{
  int m = blockIdx.x * MU_THREAEDS_PER_BLOCK + threadIdx.x;

  rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
  iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
```

FIGURE 11.7

cmpMu kernel.

A second option is to use each thread to implement an iteration of the outer loop. This option employs fewer threads than the first option. Instead of generating $N \times M$ threads, this option generates M threads. Since M corresponds to the number of k-space samples and a large number of samples (on the order of a hundred thousand) are typically used to calculate F^{H}D, this option still exploits a large amount of parallelism. However, this kernel suffers the same problem as the kernel in Figure 11.5. That is, each thread will write into all `rFhD` and `iFhD` elements, thus creating an extremely large number of conflicts between threads. As is the case of Figure 11.5, the code in Figure 11.8 will not execute correctly without adding atomic operations that will significantly slow down the execution. Thus, this option does not work well.

A third option is to use each thread to compute one pair of `rFhD` and `iFhD` elements. This option requires us to interchange the inner and outer loops and then use each thread to implement an iteration of the new outer loop. The transformation is shown in Figure 11.9. Loop interchange is necessary because the loop being implemented by the CUDA threads must be the outer loop. Loop interchange makes each of the new outer loop iterations process a pair of `rFhD` and `iFhD` elements. Loop interchange is permissible here because all iterations of both levels of loops are independent of each other. They can be executed in any order relative to one another. Loop interchange, which changes the order of the iterations, is allowed when these iterations can be executed in any order. This option generates N threads. Since N corresponds to the number of voxels in the reconstructed image, the N value can be very large for higher-resolution images.

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
    kx, ky, kz, x, y, z, rMu, iMu, int N) {

  int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  for (n = 0; n < N; n++) {
    float expFhD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]+kz[m]*z[n]);

    float cArg = cos(expFhD);
    float sArg = sin(expFhD);

    rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
    iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
  }
}
```

FIGURE 11.8

Second option of the F^{H}D kernel.

For a 128^3 images, there are $128^3 = 2,097,152$ threads, resulting in a large amount of parallelism. For higher resolutions, such as 512^3, we may need to invoke multiple kernels, and each kernel generates the value of a subset of the voxels. Note these threads now all accumulate into their own rFhD and iFhd elements since every thread has a unique n value. There is no conflict between threads. These threads can run totally in parallel. This makes the third option the best choice among the three options.

The kernel derived from the interchanged loops is shown in Figure 11.10. The threads are organized as a two-level structure. The outer loop has been stripped away; each thread covers an iteration of the outer (n) loop, where n is equal to blockIdx.x*FHD_THREADS_PER_BLOCK + threadIdx.x. Once this iteration (n) value is identified, the thread executes the inner loop based on that n value. This kernel can be invoked with a number of threads in each block, specified by a global constant FHD_THREADS_PER_BLOCK. Assuming that *N* is the variable that gives the number of voxels in the reconstructed image, N/FHD_THREADS_PER_BLOCK blocks cover all *N* iterations of the original loop. For example, if there are 65,536 *k*-space samples, the kernel could be invoked with a configuration of 512 threads per block and $65,536 \div 512 = 128$ blocks. This is done by assigning 512 to FHD_THREADS_PER_BLOCK and using FHD_THREADS_PER_BLOCK as the block size and N/FHD_THREADS_PER_BLOCK as the grid size during kernel innovation.

```
for (m = 0; m < M; m++) {          for (n = 0; n < N; n++) {
  for (n = 0; n < N; n++) {          for (m = 0; m < M; m++) {
    expFhD = 2*PI*(kx[m]*x[n] +        expFhD = 2*PI*(kx[m]*x[n] +
                   ky[m]*y[n] +                      ky[m]*y[n] +
                   kz[m]*z[n]);                      kz[m]*z[n]);

    cArg = cos(expFhD);               cArg = cos(expFhD);
    sArg = sin(expFhD);               sArg = sin(expFhD);

    rFhD[n] +=  rMu[m]*cArg -         rFhD[n] +=  rMu[m]*cArg -
                iMu[m]*sArg;                      iMu[m]*sArg;
    iFhD[n] +=  iMu[m]*cArg +         iFhD[n] +=  iMu[m]*cArg +
                rMu[m]*sArg;                      rMu[m]*sArg;
  }                                 }
} (a) before loop interchange      } (b) after loop interchange
```

FIGURE 11.9

Loop interchange of the $F^H D$ computation.

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
      kx, ky, kz, x, y, z, rMu, iMu, int M) {

  int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  for (m = 0; m < M; m++) {
    float expFhD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]+kz[m]*z[n]);

    float cArg = cos(expFhD);
    float sArg = sin(expFhD);

    rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
    iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
  }
}
```

FIGURE 11.10

Third option of the F^{H}D kernel.

Step 2: Getting Around the Memory Bandwidth Limitation

The simple cmpFhD kernel in Figure 11.10 will provide limited speedup
due to memory bandwidth limitations. A quick analysis shows that the
execution is limited by the low compute to memory access ratio of each
thread. In the original loop, each iteration performs at least 14 memory
accesses: kx[m], ky[m], kz[m], x[n], y[n], z[n], rMu[m] twice, iMu[m]
twice, rFhD[n] read and write, and iFhD[n] read and write. Meanwhile,
about 13 floating-point multiply, add, or trigonometry operations are per-
formed in each iteration. Therefore, the compute to memory access ratio is
approximately 1, which is too low according to our analysis in Chapter 5.

We can immediately improve the compute to memory access ratio by
assigning some of the array elements to automatic variables. As we dis-
cussed in Chapter 5, the automatic variables will reside in registers, thus
converting reads and writes to the global memory into reads and writes to
on-chip registers. A quick review of the kernel in Figure 11.10 shows that
for each thread, the same x[n], y[n], and z[n] elements are used across
all iterations of the for loop. This means that we can load these elements
into automatic variables before the execution enters the loop.[2] The kernel
can then use the automatic variables inside the loop, thus converting

---

[2]Note that declaring x[], y[], z[], rFhD[], and iFhD[] as automatic arrays will not
work for our purpose here. Such declaration would have created private copies of all
these five arrays in the local memory of every thread! All we want is to have a private
copy of one element of each array in the registers of each thread.

global memory accesses to register accesses. Furthermore, the loop repeatedly reads from and writes into rFhD[n] and iFhD[n]. We can have the iterations read from and write into two automatic variables and only write the contents of these automatic variables into rFhD[n] and iFhD[n] after the execution exits the loop. The resulting code is shown in Figure 11.11. By increasing the number of registers used by 5 for each thread, we have reduced the memory access done in each iteration from 14 to 7. Thus, we have increased the compute to memory access ratio from 13:14 to 13:7. This is a very good improvement and a good use of the precious register resource.

Recall that the register usage can limit the number of blocks that can run in a streaming multiprocessor (SM). By increasing the register usage by 5 in the kernel code, we increase the register usage of each thread block by 5*FHD_THREADS_PER_BLOCK. Assuming that we have 128 threads per block, we just increased the block register usage by 640. Since each SM can accommodate a combined register usage of 65,536 registers among all blocks assigned to it (at least in SM version 3.5), we need to be careful, as any further increase of register usage can begin to limit the number of blocks that can be assigned to an SM. Fortunately, the register usage is not a limiting factor to parallelism for this kernel.

We want to further improve the compute to memory access ratio to something closer to 10:1 by eliminating more global memory accesses in

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
        kx, ky, kz, x, y, z, rMu, iMu, int M) {

    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

    for (m = 0; m < M; m++) {
      float expFhD = 2*PI*(kx[m]*xn_r+ky[m]*yn_r+kz[m]*zn_r);

      float cArg = cos(expFhD);
      float sArg = sin(expFhD);

      rFhDn_r +=  rMu[m]*cArg – iMu[m]*sArg;
      iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 11.11

Using registers to reduce memory accesses in the F^HD kernel.

the `cmpFHD` kernel. The next candidates to consider are the *k*-space samples `kx[m]`, `ky[m]`, and `kz[m]`. These array elements are accessed differently than the `x[n]`, `y[n]`, and `z[n]` elements: different elements of `kx`, `ky`, and `kz` are accessed in each iteration of the loop in Figure 11.11. This means that we cannot load each *k*-space element into an automatic variable register and access that automatic variable off a register through all the iterations. So, registers will not help here. However, we should notice that the *k*-space elements are not modified by the kernel. This means that we might be able to place the *k*-space elements into the constant memory. Perhaps the cache for the constant memory can eliminate most of the memory accesses.

An analysis of the loop in Figure 11.11 reveals that the *k*-space elements are indeed excellent candidates for constant memory. The index used for accessing `kx`, `ky`, and `kz` is `m`. `m` is independent of `threadIdx`, which implies that all threads in a warp will be accessing the same element of `kx`, `ky`, and `kz`. This is an ideal access pattern for cached constant memory: every time an element is brought into the cache, it will be used at least by all 32 threads in a warp for a current generation device. This means that for every 32 accesses to the constant memory, at least 31 of them will be served by the cache. This allows the cache to effectively eliminate 96% or more of the accesses to the constant memory. Better yet, each time when a constant is accessed from the cache, it can be broadcast to all the threads in a warp. This means that no delays are incurred due to any bank conflicts in the access to the cache. This makes constant memory almost as efficient as registers for accessing *k*-space elements.[3]

There is, however, a technical issue involved in placing the *k*-space elements into the constant memory. Recall that constant memory has a capacity of 64 KB. However, the size of the *k*-space samples can be much larger, in the order of hundreds of thousands or even millions. A typical way of working around the limitation of constant memory capacity is to break down a large data set into chunks or 64 KB or smaller. The developer must reorganize the kernel so that the kernel will be invoked multiple times, with each invocation of the kernel consuming only a chunk of the large data set. This turns out to be quite easy for the `cmpFHD` kernel.

A careful examination of the loop in Figure 11.11 reveals that all threads will sequentially march through the *k*-space arrays. That is, all threads in the grid access the same *k*-space element during each iteration.

---

[3] The reason why a constant memory access is not exactly as efficient as a register access is that a memory load instruction is still needed for access to the constant memory.

For large data sets, the loop in the kernel simply iterates more times. This means that we can divide up the loop into sections, with each section processing a chunk of the *k*-space elements that fit into the 64 KB capacity of the constant memory.[4] The host code now invokes the kernel multiple times. Each time the host invokes the kernel, it places a new chunk into the constant memory before calling the kernel function. This is illustrated in Figure 11.12. (For more recent devices and CUDA versions, a const __restrict__ declaration of kernel parameters makes the corresponding input data available in the "read-only data" cache, which is a simpler way of getting the same effect as using constant memory.)

In Figure 11.12, the cmpFHd kernel is called from a loop. The code assumes that kx, ky, and kz arrays are in the host memory. The dimensions of kx, ky, and kz are given by *M*. At each iteration, the host code calls the cudaMemcpy() function to transfer a chunk of the *k*-space data into the device constant memory. The kernel is then invoked to process the chunk.

```
__constant__ float  kx_c[CHUNK_SIZE],
                    ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
…
__ void main() {

  int i;
  for (i = 0; i < M/CHUNK_SIZE; i++);
    cudaMemcpyToSymbol(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE,
                        cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
                        cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
                        cudaMemCpyHostToDevice);

    …
    cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>
              (rPhi, iPhi, phiMag, x, y, z, rMu, iMu, CHUNK_SIZE);
  }
  /* Need to call kernel one more time if M is not */
  /* perfect multiple of CHUNK SIZE */
}
```

FIGURE 11.12

Chunking *k*-space data to fit into constant memory.

---

[4]Note not all accesses to read-only data are as favorable for constant memory as what we have here. In Chapter 12 we present a case where threads in different blocks access different elements in the same iteration. This more diverged access pattern makes it much harder to fit enough of the data into the constant memory for a kernel launch.

Note that when *M* is not a perfect multiple of CHUNK_SIZE, the host code will need to have an additional round of cudaMemcpy() and one more kernel invocation to finish the remaining *k*-space data.

Figure 11.13 shows the revised kernel that accesses the *k*-space data from constant memory. Note that pointers to kx, ky, and kz are no longer in the parameter list of the kernel function. Since we cannot use pointers to access variables in the constant memory, the kx_c, ky_c, and kz_c arrays are accessed as global variables declared under the __constant__ keyword as shown Figure 11.12. By accessing these elements from the constant cache, the kernel now has effectively only four global memory accesses to the rMu and iMu arrays. The compiler will typically recognize that the four array accesses are made to only two locations. It will only perform two global accesses, one to rMu[m] and one to iMu[m]. The values will be stored in temporary register variables for use in the other two. This makes the final number of memory accesses two. The compute to memory access ratio is up to 13:2. This is still not quite the desired 10:1 ratio but is sufficiently high that the memory bandwidth limitation is no longer the only factor that limits performance. As we will see, we can perform a few other optimizations that make computation more efficient and further improve performance.

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
      x, y, z, rMu, iMu, int M) {

  int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
  float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

  for (m = 0; m < M; m++) {
    float expFhD =
      2*PI*(kx_c[m]*xn_r+ky_c[m]*yn_r+kz_c[m]*zn_r);

    float cArg = cos(expFhD);
    float sArg = sin(expFhD);

    rFhDn_r +=  rMu[m]*cArg – iMu[m]*sArg;
    iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
  }
  rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 11.13

Revised F^{H}D kernel to use constant memory.

If we ran the code in Figures 11.12 and 11.13, we would have found out that the performance enhancement was not as high as we expected for some devices. As it turns out, the code shown in these figures does not result in as much memory bandwidth reduction as we expected. The reason is that the constant cache does not perform very well for the code. This has to do with the design of the constant cache and the memory layout of the *k*-space data. As shown in Figure 11.14, each constant cache entry is designed to store multiple consecutive words. This design reduces the cost of constant cache hardware. If multiple data elements that are used by each thread are not in consecutive words, as illustrated in Figure 11.14(a), they will end up taking multiple cache entries. Due to cost constraints, the constant cache has only a very small number of entries. As shown in Figures 11.12 and 11.13, the *k*-space data is stored in three arrays: kx_c, ky_c, and kz_c. During each iteration of the loop, three entries of the constant cache are needed to hold the three *k*-space elements being processed. Since different warps can be at very different iterations, they may require many entries altogether. As it turns out, the constant cache capacity in some devices may not be sufficient to provide a sufficient number of entries for all the warps active in an SM.

The problem of inefficient use of cache entries has been well studied in the literature and can be solved by adjusting the memory layout of the *k*-space data. The solution is illustrated in Figure 11.14(b) and the code based on this solution is shown in Figure 11.15. Rather than having the *x*, *y*, and *z* components of the *k*-space data stored in three separate arrays, the solution stores these components in an array of which the elements comprise a struct. In the literature, this style of declaration is often referred to as *array of structs*. The declaration of the array is shown at the top of

(a) k-space data stored in separate arrays.   (b) k-space data stored in an array whose elements are structs.
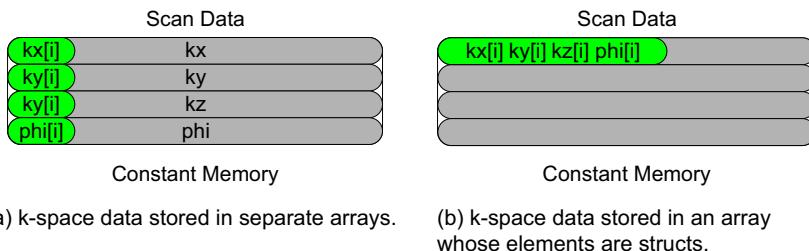
FIGURE 11.14

Effect of *k*-space data layout on constant cache efficiency: (a) *k*-space data stored in separate arrays, and (b) *k*-space data stored in an array whose elements are structs.

Figure 11.15. By storing the *x*, *y*, and *z* components in the three fields of an array element, the developer forces these components to be stored in consecutive locations of the constant memory. Therefore, all three components used by an iteration can now fit into one cache entry, reducing the number of entries needed to support the execution of all the active warps. Note that since we have only one array to hold all *k*-space data, we can just use one `cudaMemcpyToSymbol` to copy the entire chunk to the device constant memory. The size of the transfer is adjusted from `4*CHUNK_SIZE` to `12*CHUNK_SIZE` to reflect the transfer of all the three components in one `cudaMemcpy` call.

   With the new data structure layout, we also need to revise the kernel so that the access is done according to the new layout. The new kernel is shown in Figure 11.16. Note that `kx[m]` has become `k[m].x`, `ky[m]` has become `k[m].y`, and so on. As we will see later, this small change to the code can result in significant enhancement of its execution speed.

Step 3: Using Hardware Trigonometry Functions
CUDA offers hardware implementations of mathematic functions that provide much higher throughput than their software counterparts. These functions are implemented as hardware instructions executed by the SFUs (special function units). The procedure for using these functions is quite easy. In the case of the `cmpFHd` kernel, what we need to do is change the

```
struct kdata {
   float x, float y, float z;
} k;

__constant__ struct kdata k_c[CHUNK_SIZE];
…

__ void main() {

  int i;

  for (i = 0; i < M/CHUNK_SIZE; i++);
    cudaMemcpyToSymbol(k_c,k,12*CHUNK_SIZE,
        cudaMemCpyHostToDevice);

    cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>
              (…);

}
```

FIGURE 11.15

Adjusting *k*-space data layout to improve cache efficiency.

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
    x, y, z, rMu, iMu, int M) {

  int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
  float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

  for (m = 0; m < M; m++) {
    float expFhD = 2*PI*(k[m].x*xn_r+k[m].y*yn_r+k[m].z*zn_r);

    float cArg = cos(expFhD);
    float sArg = sin(expFhD);

    rFhDn_r +=  rMu[m]*cArg - iMu[m]*sArg;
    iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
  }
  rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 11.16

Adjusting the k-space data memory layout in the F^HD kernel.

calls to sin() and cos() functions into their hardware versions: __sin() and __cos(). These are intrinsic functions that are recognized by the compiler and translated into SFU instructions. Because these functions are called in a heavily executed loop body, we expect that the change will result in a very significant performance improvement. The resulting cmpFHd kernel is shown in Figure 11.17.

However, we need to be careful about the reduced accuracy when switching from software functions to hardware functions. As we discussed in Chapter 7, hardware implementations currently have slightly less accuracy than software libraries (the details are available in the *CUDA Programming Guide*). In the case of MRI, we need to make sure that the hardware implementation passes provide enough accuracy, as shown in Figure 11.18. The testing process involves a "perfect" image (I_0). We use a reverse process to generate a corresponding "scanned" k-space data that is synthesized. The synthesized scanned data is then processed by the proposed reconstruction system to generate a reconstructed image (I). The values of the voxels in the perfect and reconstructed images are then fed into the peak signal-to-noise ratio (PSNR) formula shown in Figure 11.18.

The criteria for passing the test depend on the application that the image is intended for. In our case, we worked with experts in clinical MRI to ensure that the PSNR changes due to hardware functions are well within

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
      x, y, z, rMu, iMu, int M) {

  int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

  float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
  float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

  for (m = 0; m < M; m++) {
    float expFhD = 2*PI*(k[m].x*xn_r+k[m].y*yn_r+k[m].z*zn_r);

    float cArg = __cos(expFhD);
    float sArg = __sin(expFhD);

    rFhDn_r +=  rMu[m]*cArg - iMu[m]*sArg;
    iFhDn_r +=  iMu[m]*cArg + rMu[m]*sArg;
  }
  rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 11.17

Using hardware __sin() and __cos() functions.

$$MSE = \frac{1}{mn} \sum_i \sum_j (l(i,\, j) - l_0(i,\, j))^2$$
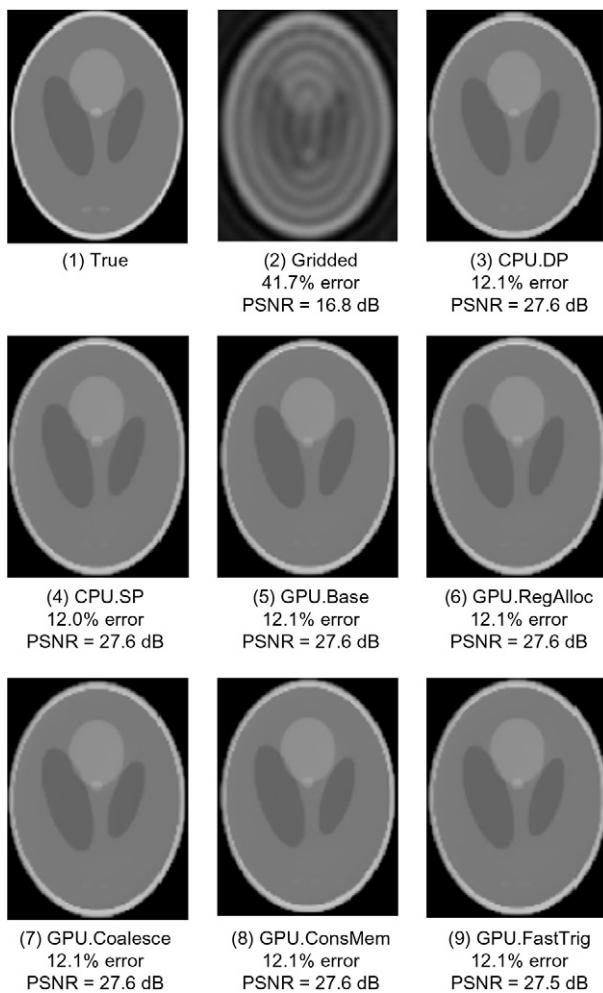
$$PSNR = 20 \log_{10}(\frac{\max(l_0(i,\, j))}{\sqrt{MSE}})$$

FIGURE 11.18

Metrics used to validate the accuracy of hardware functions. l_0 is perfect image. l is reconstructed image. PSNR is Peak signal-to-noise ratio.

the accepted limits for their applications. In applications where the images are used by physicians to form an impression of injury or evaluate a disease, one also needs to have visual inspection of the image quality. Figure 11.19 shows the visual comparison of the original "true" image. It then shows that the PSNR achieved by CPU double-precision and single-precision implementations are both 27.6 dB, an acceptable level for the application. A visual inspection also shows that the reconstructed image indeed corresponds well with the original image.

The advantage of iterative reconstruction compared to a simple bilinear interpolation gridding/iFFT is also obvious in Figure 11.19. The image reconstructed with the simple gridding/iFFT has a PSNR of only 16.8 dB, substantially lower than the PSNR of 27.6 dB achieved by the iterative

FIGURE 11.19

Validation of floating-point precision and accuracy of the different F^HD implementations.

reconstruction method. A visual inspection of the gridding/iFFT image in Figure 11.19(2) shows that there are severe artifacts that can significantly impact the usability of the image for diagnostic purposes. These artifacts do not occur in the images from the iterative reconstruction method.

When we moved from double-precision to single-precision arithmetic on the CPU, there was no measurable degradation of PSNR, which

remains at 27.6 dB. When we moved the trigonometry function from the software library to the hardware units, we observed a negligible degradation of PSNR, from 27.6 dB to 27.5 dB. The slight loss of PSNR is within an acceptable range for the application. A visual inspection confirms that the reconstructed image does not have significant artifacts compared to the original image.

Step 4: Experimental Performance Tuning

Up to this point, we have not determined the appropriate values for the configuration parameters for the kernel. For example, we need to determine the optimal number of threads for each block. On one hand, using a large number of threads in a block is needed to fully utilize the thread capacity of each SM (given that 16 blocks can be assigned to each SM at maximum). On the other hand, having more threads in each block increases the register usage of each block and can reduce the number of blocks that can fit into an SM. Some possible values of number of threads per block are 32, 64, 128, 256, and 512. One could also consider non-power-of-two numbers.

Another kernel configuration parameter is the number of times one should unroll the body of the `for` loop. This can be set using a `#pragma unroll` followed by the number of unrolls we want the compiler to perform on a loop. On one hand, unrolling the loop can reduce the number of overhead instructions, and potentially reduce the number of clock cycles to process each k-space sample data. On the other hand, too much unrolling can potentially increase the usage of registers and reduce the number of blocks that can fit into an SM.

Note that the effects of these configurations are not isolated from each other. Increasing one parameter value can potentially use the resource that could be used to increase another parameter value. As a result, one needs to evaluate these parameters jointly in an experimental manner. That is, one may need to change the source code for each joint configuration and measure the runtime. There can be a large number of source code versions to try. In the case of F^{H}D, the performance improves about 20% by systematically searching all the combinations and choosing the one with the best measured runtime, as compared to a heuristic tuning search effort that only explores some promising trends. Ryoo et al. present a pareto optimal curve−based method to screen away most of the inferior combinations [RRS2008].

11.4 **FINAL EVALUATION**

To obtain a reasonable baseline, we implemented two versions of F^HD on the CPU. Version CPU.DP uses double precision for all floating-point values and operations, while version CPU.SP uses single precision. Both CPU versions are compiled with Intel's ICC (version 10.1) using flags -O3 -msse3 -axT -vec-report3 -fp-model fast = 2, which (1) vectorizes the algorithm's dominant loops using instructions tuned for the Core 2 architecture, and (2) links the trigonometric operations to fast, approximate functions in the math library. Based on experimental tuning with a smaller data set, the inner loops are unrolled by a factor of four and the scan data is tiled to improve locality in the L1 cache.

Each GPU version of F^HD is compiled using NVCC-O3 (CUDA version 1.1) and executed on a 1.35 GHz Quadro FX5600. The Quadro card is housed in a system with a 2.4 GHz dual-socket, dual-core Opteron 2216 CPU. Each core has a 1 MB L2 cache. The CPU versions use p-threads to execute on all four cores of a 2.66 GHz Core 2 Extreme quad-core CPU, which has peak theoretical capacity of 21.2 GFLOPS per core and a 4 MB L2 cache. The CPU versions perform substantially better on the Core 2 Extreme quad-core than on the dual-socket, dual-core Opteron. Therefore, we will use the Core 2 Extreme quad-core results for the CPU.

All reconstructions use the CPU version of the linear solver, which executes 60 iterations on the Quadro FX5600. Two versions of Q were computed on the Core 2 Extreme, one using double precision and the other using single precision. The single-precision Q was used for all GPU-based reconstructions and for the reconstruction involving CPU.SP, while the double-precision Q was used only for the reconstruction involving CPU. DP. As the computation of Q is not on the reconstruction's critical path, we give Q no further consideration.

To facilitate comparison of the iterative reconstruction with a conventional reconstruction, we also evaluated a reconstruction based on bilinear interpolation gridding and inverse FFT. Our version of the gridded reconstruction is not optimized for performance, but it is already quite fast.

All reconstructions are performed on sample data obtained from a simulated, 3D, non-Cartesian scan of a phantom image. There are 284,592 sample points in the scan data set, and the image is reconstructed at 128^3 resolution, for a total of 2^{21} voxels. In the first set of experiments, the simulated data contains no noise. In the second set of experiments, we added complex white Gaussian noise to the simulated data. When determining the quality of the reconstructed images, the percent error and

PSNR metrics are used. The percent error is the root-mean-square (RMS) of the voxel error divided by the RMS voxel value in the true image (after the true image has been sampled at 1,283 resolution).

The data (runtime, GFLOPS, and images) was obtained by reconstructing each image once with each of the implementations of the F^HD algorithm described before. There are two exceptions to this policy. For GPU. Tune and GPU.Multi, the time required to compute F^HD is so small that runtime variations in performance became non-negligible. Therefore, for these configurations we computed F^HD three times and reported the average performance.

As shown in Figure 11.20, the total reconstruction time for the test image using bilinear interpolation gridding followed by inverse FFT takes less than one minute on a high-end sequential CPU. This confirms that there is little value in parallelizing this traditional reconstruction strategy. It is, however, obvious from Figure 11.19(2) that the resulting image exhibits an unacceptable level of artifacts.

The LS (CPU, DP) row shows the execution timing of reconstructing the test image using double-precision, floating-point arithmetic on the CPU. The timing shows the core step (Q) of calculating $F^HF + \lambda W^HW$. The first observation is that the Q computation for a moderate resolution image based on a moderate-size data sample takes an unacceptable amount of time (more than 65 hours) on the CPU for setting up the system for a patient. Note that this time is eventually reduced to 6.5 minutes on the GPU with all the optimizations described in Section 11.3.

	Q		F^HD		Total	
Reconstruction	Run Time (m)	GFLOP	Run Time (m)	GFLOP	Linear Solver (m)	Recon. Time (m)
Gridding + FFT (CPU, DP)	N/A	N/A	N/A	N/A	N/A	0.39
LS (CPU, DP)	4009.0	0.3	518.0	0.4	1.59	519.59
LS (CPU, SP)	2678.7	0.5	342.3	0.7	1.61	343.91
LS (GPU, Naïve)	260.2	5.1	41.0	5.4	1.65	42.65
LS (GPU, CMem)	72.0	18.6	9.8	22.8	1.57	11.37
LS (GPU, CMem, SFU, Exp)	7.5 357X	178.9	1.5 228X	144.5	1.69	3.19 108X

FIGURE 11.20

Summary of performance improvements.

The second observation is that the total reconstruction time of each image requires more than 8 hours, with only 1.59 minutes spent in the linear solver. This validates our decision to focus our parallelization effort on F^HD.

The LS (CPU, SP) row shows that we can reduce the execution time significantly when we convert the computation from double-precision, floating-point arithmetic to single precision on the CPU. This is because the streaming SIMD instruction (SSE) instructions have higher throughput, that is, they calculate more data elements per clock cycle when executing in single-precision mode. The execution times, however, are still unacceptable for practical use.

The LS (GPU, Naïve) row shows that a straightforward CUDA implementation can achieve a speedup about $10\times$ for Q and $8\times$ for the reconstruction of each image. This is a good speedup, but the resulting execution times are still unacceptable for practical use.

The LS (GPU, CMem) row shows that significant further speedup is achieved by using registers and constant cache to get around the global memory bandwidth limitations. These enhancements achieve about $4\times$ speedup over the naïve CUDA code! This shows the importance of achieving optimal compute to memory ratios in CUDA kernels. These enhancements bring the CUDA code to about $40\times$ speedup over the single-precision CPU code.

The LS (GPU, CMem, SPU, Exp) row shows the use of hardware trigonometry functions and experimental tuning together, and results in a dramatic speedup. A separate experiment, not shown in the figure, shows that most of the speedup comes from hardware trigonometry functions. The total speedup over CPU single-precision code is very impressive: $357\times$ for Q and $108\times$ for the reconstruction of each image.

An interesting observation is that in the end, the linear solver actually takes more time than F^HD. This is because we have accelerated F^HD dramatically ($228\times$). What used to be close to 100% of the per-image reconstruction time now accounts for less than 50%. Any further acceleration will now require acceleration of the linear solver, a much more difficult type of computation for massively parallel execution.

11.5 EXERCISES

11.1. Loop fission splits a loop into two loops. Use the F^HD code in Figure 11.4(b) and enumerate the execution order of the two parts

of the outer loop body: (1) the statements before the inner loop and (2) the inner loop.

(a) List the execution order of these parts from different iterations of the outer loop before fission.

(b) List the execution order of these parts from the two loops after fission. Determine if the execution results will be identical. The execution results are identical if all data required by a part is properly generated and preserved for its consumption before that part executes, and the execution result of the part is not overwritten by other parts that should come after the part in the original execution order.

11.2. Loop interchange swaps the inner loop into the outer loop and vice versa. Use the loops from Figure 11.9 and enumerate the execution order of the instances of the loop body before and after the loop exchange.

(a) List the execution order of the loop body from different iterations before the loop interchange. Identify these iterations with the values of m and n.

(b) List the execution order of the loop body from different iterations after the loop interchange. Identify these iterations with the values of m and n.

(c) Determine if the (a) and (b) execution results will be identical. The execution results are identical if all data required by a part is properly generated and preserved for its consumption before that part executes and the execution result of the part is not overwritten by other parts that should come after the part in the original execution order.

11.3. In Figure 11.11, identify the difference between the access to x[] and kx[] in the nature of indices used. Use the difference to explain why it does not make sense to try to load kx[n] into a register for the kernel shown in Figure 11.11.

11.4. During a meeting, a new graduate student told his advisor that he improved his kernel performance by using cudaMalloc() to allocate constant memory and by using cudaMemcpy() to transfer read-only data from the CPU memory to the constant memory. If you were his advisor, what would be your response?

References

Liang, Z. P., & Lauterbur, P. (1999). *Principles of Magnetic Resonance Imaging: A Signal Processing Perspective* New York: John Wiley and Sons.

Haldar, J. P., Hernando, D., Budde, M. D., Wang, Q., Song, S. -K., & Liang., Z. -P. (2007). High-resolution MR metabolic imaging. *In Proc. IEEE EMBS*, 4324−4326.

Ryoo, S., Ridrigues, C. I., & Stone, S. S., et al. (2008). Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, . doi:10.1016/j.jpdc.2008.05.011.

S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, B. P. Sutton, and Z. P. Liang, Accelerating advanced MRI reconstruction on GPUs, Journal of Parallel and Distributed Computing, 2008, doi:10.1016/j.jpdc.2008.05.013.

Application Case Study: Molecular Visualization and Analysis

12

With special contributions from John Stone

CHAPTER OUTLINE

The previous case study illustrated the process of selecting an appropriate level of a loop nest for parallel execution, the use of constant memory for magnifying the memory bandwidth for read-only data, the use of registers to reduce the consumption of memory bandwidth, and the use of special hardware functional units to accelerate trigonometry functions. In this case study, we use an application based on regular grid data structures to illustrate the use of additional practical techniques that achieve global memory access coalescing and improved computation throughput. We present a series of implementations of an electrostatic potential map calculation kernel, with each version improving upon the previous one. Each version adopts one or more practical techniques. This computation pattern of this application is one of the best matches for massively parallel computing. This application case study shows that the effective use of these practical techniques can significantly improve the execution throughput and is critical for the application to achieve its potential performance.
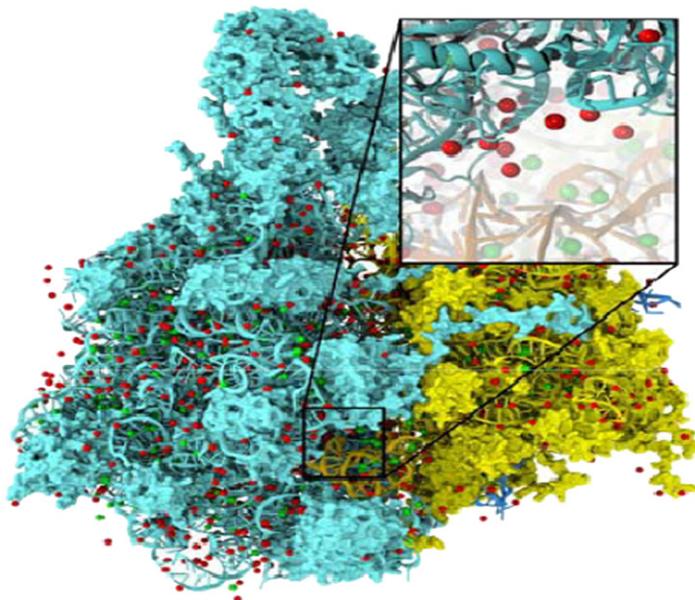
12.1 APPLICATION BACKGROUND

This case study is based on VMD (Visual Molecular Dynamics) [HDS1996], a popular software system designed for displaying, animating, and analyzing biomolecular systems. As of 2012, VMD has more than 200,000 registered users. While it has strong built-in support for analyzing biomolecular systems, such as calculating electrostatic potential values at spatial grid points of a molecular system, it has also been a popular tool for displaying other large data sets, such as sequencing data, quantum chemistry simulation data, and volumetric data, due to its versatility and user extensibility.
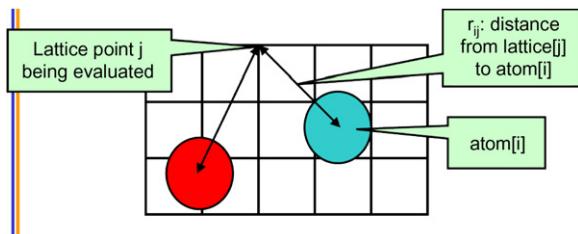
While VMD is designed to run on a diverse range of hardware—laptops, desktops, clusters, and supercomputers—most users use VMD as a desktop science application for interactive 3D visualization and analysis. For computation that runs too long for interactive use, VMD can also be used in a batch mode to render movies for later use. A motivation for accelerating VMD is to make batch mode jobs fast enough for interactive use. This can drastically improve the productivity of scientific investigations. With CUDA devices widely available in desktop PCs, such acceleration can have broad impact on the VMD user community. To date, multiple aspects of VMD have been accelerated with CUDA, including electrostatic potential calculation, ion placement, molecular orbital calculation and display, and imaging of gas migration pathways in proteins.

The particular calculation used in this case study is the calculation of electrostatic potential maps in a grid space. This calculation is often used in placement of ions into a structure for molecular dynamics simulation. Figure 12.1 shows the placement of ions into a protein structure in preparation for molecular dynamics simulation. In this application, the electrostatic potential map is used to identify spatial locations where ions (round dots around the large molecules) can fit in according to physical laws. The function can also be used to calculate time-averaged potentials during molecular dynamics simulation, which is useful for the simulation process as well as the visualization/analysis of simulation results.

There are several methods for calculating electrostatic potential maps. Among them, direct coulomb summation (DCS) is a highly accurate method that is particularly suitable for GPUs [SPF2007]. The DCS method calculates the electrostatic potential value of each grid point as the sum of contributions from all atoms in the system. This is illustrated in Figure 12.2. The contribution of atom i to a lattice point j is the charge of that atom divided by the distance from lattice point j to atom i. Since this

FIGURE 12.1

Electrostatic potential map is used in building stable structures for molecular dynamics simulation.

FIGURE 12.2

The contribution of `atom[i]` to the electrostatic potential at lattice point j (`potential[j]`) is `atom[i].charge`/r_{ij}. In the DCS method, the total potential at lattice point j is the sum of contributions from all atoms in the system.

needs to be done for all grid points and all atoms, the number of calculations is proportional to the product of the total number of atoms in the system and the total number of grid points. For a realistic molecular system, this product can be very large. Therefore, the calculation of the electrostatic potential map has been traditionally done as a batch job in VMD.

12.2 A SIMPLE KERNEL IMPLEMENTATION

Figure 12.3 shows the base C code of the DCS code. The function is written to process a 2D slice of a 3D grid. The function will be called repeatedly for all the slices of the modeled space. The structure of the function is quite simple with three levels of `for` loops. The outer two levels iterate over the y dimension and the x dimension of the grid point space. For each grid point, the innermost `for` loop iterates over all atoms, calculating the contribution of electrostatic potential energy from all atoms to the grid point. Note that each atom is represented by four consecutive elements of the `atoms[]` array. The first three elements store the x, y, and z coordinates of the atom and the fourth element the electrical charge of the atom. At the end of the innermost loop, the accumulated value of the grid point is written out to the grid data structure. The outer loops then iterate and take the execution to the next grid point.

Note that the DCS function in Figure 12.3 calculates the x and y coordinates of each grid point on-the-fly by multiplying the grid point index values by the spacing between grid points. This is a uniform grid method where all grid points are spaced at the same distance in all three

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms,
             int numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {    // calculate potential contribution of each atom
        float dx = x - atoms[n   ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
  }
}
```

FIGURE 12.3

Base coulomb potential calculation code for a 2D slice.

dimensions. The function does take advantage of the fact that all the grid points in the same slice have the same z coordinate. This value is precalculated by the caller of the function and passed in as a function parameter (z).

Based on what we learned from the MRI case study, two attributes of the DCS method should be apparent. First, the computation is massively parallel: the computation of electrostatic potential for each grid point is independent of that of other grid points. There are two alternative approaches to organizing parallel execution. In the first option, we can use each thread to calculate the contribution of one atom to all grid points. This would be a poor choice since each thread would be writing to all grid points, requiring extensive use of atomic memory operations to coordinate the updates done by different threads to each grid point. The second option uses each thread to calculate the accumulated contributions of all atoms to one grid point. This is a preferred approach since each thread will be writing into its own grid point and there is no need to use atomic operations.

We will form a 2D thread grid that matches the 2D energy grid point organization. To do so, we need to modify the two outer loops into perfectly nested loops so that we can use each thread to execute one iteration of the two-level loop. We can either perform a loop fission, or we move the calculation of the y coordinate into the inner loop. The former would require us to create a new array to hold all y values and result in two kernels communicating data through global memory. The latter increases the number of times that the y coordinate will be calculated. In this case, we choose to perform the latter since there is only a small amount of calculation that can be easily accommodated into the inner loop without a significant increase in execution time of the inner loop. The former would have added a kernel launch overhead for a kernel where threads do very little work. The selected transformation allows all i and j iterations to be executed in parallel. This is a trade-off between the amount of calculation done and the level of parallelism achieved.

The second experience that we can apply from the MRI case study is that the electrical charge of every atom will be read by all threads. This is because every atom contributes to every grid point in the DCS method. Furthermore, the values of the atomic electrical charges are not modified during the computation. This means that the atomic charge values can be efficiently stored in the constant memory (in the GPU box in Figure 12.4).

Figure 12.4 shows an overview of the DCS kernel design. The host program (host box in Figure 12.4) inputs and maintains the atomic charges and their coordinates in the system memory. It also maintains the grid

point data structure in the system memory (left side in the host box). The DCS kernel is designed to process a 2D slice of the energy grid point structure (not to be confused with thread grids). The right side grid in the host box shows an example of a 2D slice. For each 2D slice, the CPU transfers its grid data to the device global memory. The atom information is divided into chunks to fit into the constant memory. For each chunk of the atom information, the CPU transfers the chunk into the device constant memory, invokes the DCS kernel to calculate the contribution of the current chunk to the current slice, and prepares to transfer the next chunk. After all chunks of the atom information have been processed for the current slice, the slice is transferred back to update the grid point data structure in the CPU system memory. The system moves on to the next slice.

Within each kernel invocation, the thread blocks are organized to calculate the electrostatic potential of tiles of the grid structure. In the simplest kernel, each thread calculates the value at one grid point. In more sophisticated kernels, each thread calculates multiple grid points and exploits the redundancy between the calculations of the grid points to improve execution speed. This is illustrated in the left side portion labeled as "thread blocks" in Figure 12.4 and is an example of the granularity adjustment optimization discussed in Chapter 6.
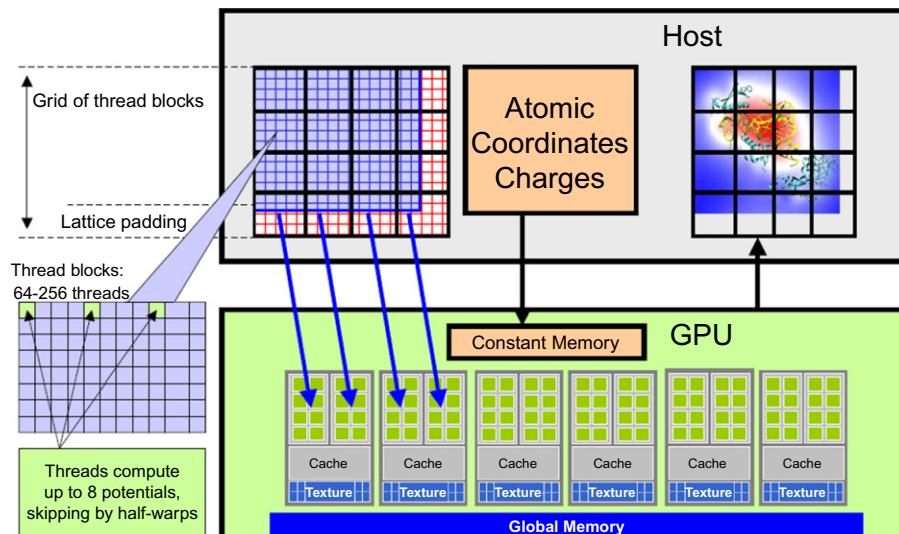
FIGURE 12.4

Overview of the DCS kernel design.

Figure 12.5 shows the resulting CUDA kernel code. We omitted some of the declarations. As was the case in the MRI case study, the `atominfo[]` array is declared in the constant memory by the host code. The host code also needs to divide up the atom information into chunks that fit into the constant memory for each kernel invocation. This means that the kernel will be invoked multiple times when there are multiple chunks of atoms. Since this is similar to the MRI case study, we will not show the details.

The outer two levels of the loop in Figure 12.3 have been removed from the kernel code and are replaced by the execution configuration parameters in the kernel invocation. Since this is also similar to one of the steps we took in the MRI case study, we will not show the kernel invocation but leave it as an exercise for readers. The rest of the kernel code is straightforward and corresponds directly to the original loop body of the innermost loop.

One particular aspect of the kernel is somewhat subtle and worth mentioning. The kernel code calculates the contribution of a chunk of atoms to a grid point. The grid point must be preserved in the global memory and updated by each kernel invocation. This means that the kernel needs to read the current grid point value, add the contributions by the current chunk of atoms, and write the updated value to global memory. The code attempts to hide the global memory latency by loading the grid value at the beginning of the kernel and using it at the end of the kernel. This helps

```
…
float curenergy = energygrid[outaddr];          ← Start global memory reads
float coorx = gridspacing * xindex;               early. Kernel hides some of
float coory = gridspacing * yindex;                      its own latency.
int atomid;
float energyval=0.0f;
for (atomid=0; atomid<numatoms; atomid++) {
    float dx = coorx - atominfo[atomid].x;
    float dy = coory - atominfo[atomid].y;
    energyval += atominfo[atomid].w *
                  rsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
}                                               Only dependency on global
energygrid[outaddr] = curenergy + energyval;    memory read is at the end of
                                                       the kernel…
```

FIGURE 12.5

DCS kernel version 1.

to reduce the number of warps needed by the streaming multiprocessor (SM) scheduler to hide the global memory latency.
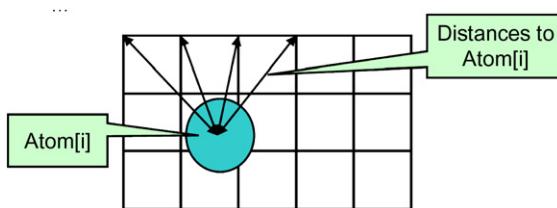
The performance of the kernel in Figure 12.5 is quite good, measured at 186 GFLOPS on a G80, a first-generation CUDA device. In terms of application-level performance, the implementation can process 18.6 billion atom evaluations per second. A quick glance over the code shows that each thread does nine floating-point operations for every four memory elements accessed. On the surface, this is not a very good ratio. We need at least a ratio of 8 to avoid global memory congestion. However, all four memory accesses are done to the `atominfo[]` array. These `atominfo[]` array elements for each atom are cached in a hardware cache memory in each SM and are broadcast to a large number of threads. A similar calculation to that in the MRI case study shows that the massive reuse of memory elements across threads make the constant cache extremely effective, boosting the effective ratio of floating operations per global memory access much higher than 10:1. As a result, global memory bandwidth is not a limiting factor for this kernel.

12.3 THREAD GRANULARITY ADJUSTMENT

Although the kernel in Figure 12.5 avoids global memory bottleneck through constant caching, it still needs to execute four constant memory access instructions for every nine floating-point operations performed. These memory access instructions consume hardware resources that could be otherwise used to increase the execution throughput of floating-point instructions. This section shows that we can fuse several threads together so that the `atominfo[]` data can be fetched once from the constant memory, stored into registers, and used for multiple grid points. This idea is illustrated in Figure 12.6.

Furthermore, all grid points along the same row have the same y coordinate. Therefore, the difference between the y coordinate of an atom and the y coordinate of any grid point along a row has the same value. In the DCS kernel version 1 in Figure 12.5, this calculation is redundantly done by all threads for all grid points in a row when calculating the distance between the atom and the grid points. We can eliminate this redundancy and improve the execution efficiency.

The idea is to have each thread to calculate the electrostatic potential for multiple grid points. The kernel in Figure 12.7 has each thread calculate four grid points. For each atom, the code calculates dy, the difference

FIGURE 12.6

Reusing information among multiple grid points.


```
...for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float x = atominfo[atomid].x;
    float dx1 = coorx1 - x;
    float dx2 = coorx2 - x;
    float dx3 = coorx3 - x;
    float dx4 = coorx4 - x;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * rsqrtf(dx1*dx1 + dysqpdzsq);
    energyvalx2 += charge * rsqrtf(dx2*dx2 + dysqpdzsq);
    energyvalx3 += charge * rsqrtf(dx3*dx3 + dysqpdzsq);
    energyvalx4 += charge * rsqrtf(dx4*dx4 + dysqpdzsq);
    }
```

Compared to non-unrolled kernel: memory loads are decreased by 4x, and FLOPS per evaluation are reduced, but register use is increased...

FIGURE 12.7

DCS kernel version 2.

of the y coordinate in line 2. It then calculates the expression dy*dy plus the precalculated dz*dz information and saves it to the auto variable dysqpdzsq, which is assigned to a register by default. This value is the same for all four grid points. Therefore, the calculation of energyvalx1 through energyvalx4 can all just use the value stored in the register. Furthermore, the electrical charge information is also accessed from constant memory and stored in the automatic variable charge. Similarly, the *x* coordinate of the atom is also read from constant memory into auto variable x. Altogether, this kernel eliminates three accesses to constant memory for atominfo[atomid].y, three accesses to constant memory for atominfo[atomid].x, three accesses to constant memory for atominfo[atomid].w, three floating-point subtraction operations, five floating-point multiply operations, and nine floating-point add operations when processing an atom for four grid points. A quick inspection of the kernel code in

Figure 12.7 shows that each iteration of the loop performs four constant memory accesses, five floating-point subtractions, nine floating-point additions, and five floating-point multiplications for four grid points.
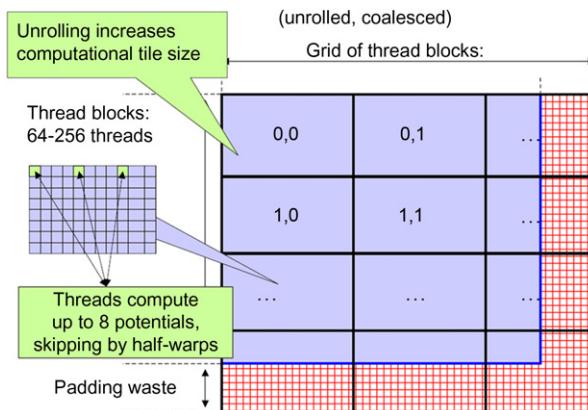
Readers should also verify that the version of DCS kernel in Figure 12.5 performs 16 constant memory accesses, 8 floating-point subtractions, 12 floating-point additions, and 12 floating-point multiplications, for a total of 48 operations for the same four grid points. Going from Figure 12.5 to Figure 12.7, there is a total reduction from 48 operations down to 25 operations, a sizable reduction. This is translated into an increased execution speed from 186 GFLOPS to 259 GFLOPS on a G80. In terms of application-level throughput, the performance increases from 18.6 billion atom evaluations per second to 33.4 billion atom evaluations per second. The reason that the application-level performance improvement is higher than the FLOPS improvement is that some of the floating-point operations have been eliminated.

The cost of the optimization is that more registers are used by each thread. This reduces the number of threads that can be assigned to each SM. However, as the results show, this is a good trade-off with an excellent performance improvement.

12.4 MEMORY COALESCING

While the performance of the DCS kernel version 2 in Figure 12.7 is quite high, a quick profiling run reveals that the threads perform memory writes inefficiently. As shown in Figures 12.6 and 12.7, each thread calculates four neighboring grid points. This seems to be a reasonable choice. However, as we illustrate in Figure 12.8, the access pattern of threads will result in uncoalesced global memory writes.

There are two problems that cause the uncoalesced writes in DCS kernel version 2. First, each thread calculates four adjacent neighboring grid points. Thus, for each statement that accesses the energygrid[] array, the threads in a warp are not accessing adjacent locations. Note that two adjacent threads access memory locations that are three elements apart. Thus, the 16 locations to be written by all the threads in warp write are spread out, with three elements in between the loaded/written locations. This problem can be solved by assigning adjacent grid points to adjacent threads in each half-warp. Assuming that we still want to have each thread calculate four grid points, we first assign 16 consecutive grid points to the 16 threads in a half-warp. We then assign the next 16 consecutive grid

FIGURE 12.8

Organizing threads and memory layout for coalesced writes.

points to the same 16 threads. We repeat the assignment until each thread has the number of grid points desired. This assignment is illustrated in Figure 12.8. With some experimentation, the best number of grid points per thread turns out to be 8 for G80.

The kernel code with a warp-aware assignment of grid points to threads is shown in Figure 12.9. Note that the x coordinates used to calculate the distances are offset by the variable gridspacing_coalescing, which is the original grid spacing times the constant BLOCKSIZE (16). This reflects the fact that the x coordinates of the 8 grid points are 16 grid points away from each other. Also, after the end of the loop, memory writes to the energygrid[] array are indexed by outaddr, outaddr + BLOCKSIZE, ..., outaddr + 7*BLOCKSIZE. Each of these indices is one BLOCKSIZE (16) away from the previous one. The detailed thread block organization for this kernel is left as an exercise. Readers should keep in mind that by setting the x dimension size of the thread block to be equal to the half-warp size (16), we can simplify the indexing in the kernel.

The other cause of uncoalesced memory writes is the layout of the energygrid[] array, which is a 3D array. If the x dimension of the array is not a multiple of half-warp size, the beginning location of the second row, as well as those of the subsequent rows, will no longer be at the 16-word boundaries. In older devices, this means that the half-warp accesses will not be coalesced, even though they write to consecutive locations. This problem can be corrected by padding each row with additional

```
...float coory = gridspacing * yindex;
  float coorx = gridspacing * xindex;
  float gridspacing_coalesce = gridspacing * BLOCKSIZEX;        Points spaced for
  int atomid;                                                   memory coalescing
  for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;          Reuse partial distance
    float dx1 = coorx - atominfo[atomid].x;               components dy^2 + dz^2
[...]
    float dx8 = dx7 + gridspacing_coalesce;
    energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
[...]
    energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
  }
  energygrid[outaddr                 ] += energyvalx1;       Global memory ops
[...]                                                        occur only at the end
  energygrid[outaddr+7*BLOCKSIZEX] += energyvalx7;              of the kernel,
                                                            decreases register use
```

FIGURE 12.9

DCS kernel version 3.

elements so that the total length of the x dimension is a multiple of 16. This can require adding up to 15 elements, or 60 bytes to each row, as shown in Figure 12.8. With the kernel of Figure 12.9, the number of elements in the x dimension needs to be a multiple of $8 \times 16 = 128$. This is because each thread actually writes 8 elements in each iteration. Thus, one may need to pad up to 127 elements, or 1,016 bytes, to each row.

Furthermore, there is a potential problem with the last row of thread blocks. Since the grid array may not have enough rows, some of the threads may end up writing outside the grid data structure. Since the grid data structure is a 3D array, these threads will write into the next slice of grid points. As we discussed in Chapter 3, we can add a test in the kernel and avoid writing the array elements that are out of the known y dimension size. However, this would have added a number of overhead instructions and incurred control divergence. Another solution is to pad the y dimension of the grid structure so that it contains a multiple of tiles covered by thread blocks. This is shown in Figure 12.8 as the bottom padding in the grid structure. In general, one may need to add up to 15 rows due to this padding.

The cost of padding can be substantial for smaller grid structures. For example, if the energy grid has 100×100 grid points in each 2D slice, it would be padded into a 128×112 slice. The total number of grid points
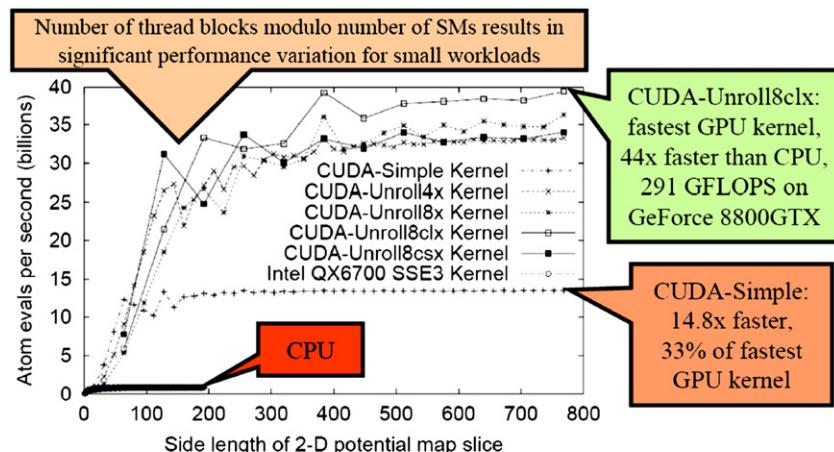
increase from 10,000 to 14,336, or a 43% overhead. If we had to pad the entire 3D structure, the grid points would have increased from $100 \times 100 \times 100$ (1,000,000) to $128 \times 112 \times 112$ (1,605,632), or a 60% overhead! This is part of the reason why we calculate the energy grids in 2D slices and use the host code to iterate over these 2D slices. Writing a single kernel to process the entire 3D structure would have incurred a lot more extra overhead. This type of trade-off appears frequently in simulation models, differential equation solvers, and video processing applications.

The DCS version 3 kernel shown in Figure 12.9 achieves about 291 GFLOPs, or 39.5 billion atom evaluations per second on a G80. On a later Fermi device, it achieves 535.16 GFLOPS, or 72.56 billion atom evaluations per sec. On a recent GeForce GTX680 Kepler 1, it achieves a whopping 1267.26 GFLOPS, or 171.83 billion atom evaluations per sec! This measured speed of the kernel also includes a slight boost from moving the read access to the `energygrid[]` array from the beginning of the kernel to the end of the kernel. The contribution to the grid points are first calculated in the loop. The code loads the original grid point data after the loop, adds the contribution to the data, and writes the updated values back. Although this movement exposes more of the global memory latency to each thread, it saves the consumption of eight registers. Since the kernel is already using many registers to hold the atom data and the distances, such savings in number of registers used relieves a critical bottleneck for the kernel. This allows more thread blocks to be assigned to each SM and achieves an overall performance improvement.

12.5 SUMMARY

Figure 12.10 shows a summary of the performance comparison between the various DCS kernel implementations and how they compare with an optimized single-core CPU execution. One important observation is that the relative merit of the kernels varies with grid dimension lengths. However, the DCS version 3 (CUDA-Unroll8clx) performs consistently better than all others once the grid dimension length is larger than 300.
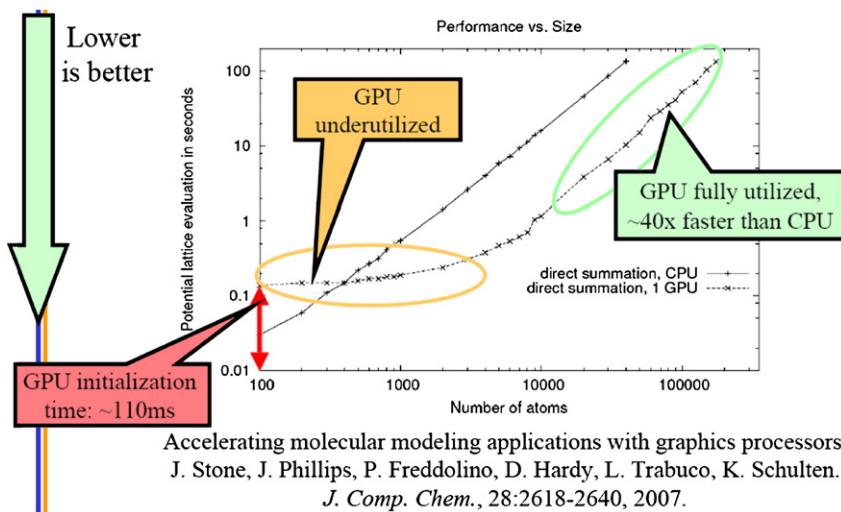
A detailed comparison between the CPU performance and the CPU−GPU joint performance shows a commonly observed trade-off. Figure 12.11 shows plot of the execution time of a medium-size grid system for a varying number of atoms to be evaluated. For 400 atoms or fewer, the CPU performs better. This is because the GPU has a fixed initialization overhead of 110 ms regardless of the number of atoms to be evaluated.

GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone,
J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.

FIGURE 12.10

Performance comparison of various DCS kernel versions.

Accelerating molecular modeling applications with graphics processors.
J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten.
J. Comp. Chem., 28:2618-2640, 2007.

FIGURE 12.11

Single-thread CPU versus CPU—GPU comparison.

Also, for a small number of atoms, the GPU is underutilized, thus the curve of the GPU execution time is quite flat between 100 and 1,000 atoms.

The plot in Figure 12.11 reenforces a commonly held principle that GPUs perform better for large amounts of data. Once the number of atoms reaches 10,000, the GPU is fully utilized. The slopes of the CPU and the CPU−GPU execution time become virtually identical, with the CPU−GPU execution being consistently $44 \times$ times faster than the CPU execution for all input sizes.

12.6 EXERCISES

12.1. Complete the implementation of the DCS kernel as outlined in Figure 12.5. Fill in all the missing declarations. Give the kernel launch statement with all the execution configuration parameters.

12.2. Compare the number of operations (memory loads, floating-point arithmetic, branches) executed in each iteration of the kernel in Figure 12.7 compared to that in Figure 12.5. Keep in mind that each iteration of the former corresponds to four iterations of the latter.

12.3. Complete the implementation of the DCS kernel version 3 in Figure 12.9. Explain in your own words how the thread accesses are coalesced in this implementation.

12.4. For the memory padding in Figure 12.8 and DCS kernel version 3 in Figure 12.9, show why one needs to pad up to 127 elements in the x dimension but only up to 15 elements in the y dimension.

12.5. Give two reasons for adding extra "padding" elements to arrays allocated in the GPU global memory, as shown in Figure 12.8.

12.6. Give two potential disadvantages associated with increasing the amount of work done in each CUDA thread, as shown in Section 12.3.

References

Humphrey, W., Dalke, A., & Schulten, K. (1996). VMD—Visual Molecular Dynamics. *Journal of Molecular Graphics*, *14*, 33−38.

Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., & Schulten, K. (2007). Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, *28*, 2618−2640.

This page intentionally left blank

Parallel Programming and Computational Thinking

13

We have so far concentrated on the practical experience of parallel programming, which consists of CUDA programming model features, performance and numerical considerations, parallel patterns, and application case studies. We will now switch gear to more abstract concepts. We will first generalize parallel programming into a computational thinking process of decomposing a domain problem into well-defined, coordinated work units that can each be realized with efficient numerical methods and well-known algorithms. A programmer with strong computational thinking skills not only analyzes but also transforms the structure of a domain problem: which parts are inherently serial, which parts are amenable to high-performance parallel execution, and the trade-offs involved in moving parts from the former category to the latter. With good problem decomposition, the programmer can select and implement algorithms that achieve an appropriate compromise between parallelism, computational efficiency, and memory bandwidth consumption. A strong combination of domain knowledge and computational thinking skills is often needed for creating successful computational solutions to challenging domain problems. This chapter will give readers more insight into parallel programming and computational thinking in general.

13.1 GOALS OF PARALLEL COMPUTING

Before we discuss the fundamental concepts of parallel programming, it is important for us to first review the three main reasons why people adopt parallel computing. The first goal is to solve a given problem in less time. For example, an investment firm may need to run a financial portfolio scenario risk analysis package on all its portfolios during after-trading hours. Such an analysis may require 200 hours on a sequential computer. However, the portfolio management process may require that analysis be completed in four hours to be in time for major decisions based on that information. Using parallel computing may speed up the analysis and allow it to complete within the required time window.

The second goal of using parallel computing is to solve bigger problems within a given amount of time. In our financial portfolio analysis example, the investment firm may be able to run the portfolio scenario risk analysis on its current portfolio within a given time window using sequential computing. However, the firm is planning on expanding the number of holdings in its portfolio. The enlarged problem size would cause the running time of analysis under sequential computation to exceed the time window. Parallel computing that reduces the running time of the bigger problem size can help accommodate the planned expansion to the portfolio.

The third goal of using parallel computing is to achieve better solutions for a given problem and a given amount of time. The investment firm may have been using an approximate model in its portfolio scenario risk analysis. Using a more accurate model may increase the computational complexity and increase the running time on a sequential computer beyond the allowed window. For example, a more accurate model may require consideration of interactions between more types of risk factors using a more numerically complex formula. Parallel computing that reduces the running time of the more accurate model may complete the analysis within the allowed time window.

In practice, parallel computing may be driven by a combination of the aforementioned three goals. It should be clear from our discussion that parallel computing is primarily motivated by increased speed. The first goal is achieved by increased speed in running the existing model on the current problem size. The second goal is achieved by increased speed in running the existing model on a larger problem size. The third goal is achieved by increased speed in running a more complex model on the current problem size. Obviously, the increased speed through parallel computing can be used to achieve a combination of these goals. For example, parallel computing can reduce the runtime of a more complex model on a larger problem size.

It should also be clear from our discussion that applications that are good candidates for parallel computing typically involve large problem sizes and high modeling complexity. That is, these applications process a large amount of data, perform many iterations on the data, or both. For such a problem to be solved with parallel computing, the problem must be formulated in such a way that it can be decomposed into subproblems that can be safely solved at the same time. Under such formulation and decomposition, the programmer writes code and organizes data to solve these subproblems concurrently.

In Chapters 11 and 12 we presented two problems that are good candidates for parallel computing. The MRI reconstruction problem involves a large amount of k-space sample data. Each k-space sample data is also used many times for calculating its contributions to the reconstructed voxel data. For a reasonably high-resolution reconstruction, each sample data is used a very large number of times. We showed that a good decomposition of the $F^H D$ problem in MRI reconstruction is to form subproblems that each calculate the value of an $F^H D$ element. All these subproblems can be solved in parallel with each other. We use a massive number of CUDA threads to solve these subproblems.

Figure 12.11 further shows that the electrostatic potential calculation problem should be solved with a massively parallel CUDA device only if there are 400 or more atoms. A realistic molecular dynamic system model typically involves at least hundreds of thousands of atoms and millions of energy grid points. The electrostatic charge information of each atom is used many times in calculating its contributions to the energy grid points. We showed that a good decomposition of the electrostatic potential calculation problem is to form subproblems that each calculate the energy value of a grid point. All the subproblems can be solved in parallel with each other. We use a massive number of CUDA threads to solve these subproblems.

The process of parallel programming can typically be divided into four steps: problem decomposition, algorithm selection, implementation in a language, and performance tuning. The last two steps were the focus of previous chapters. In the next two sections, we will discuss the first two steps with more generality as well as depth.

13.2 PROBLEM DECOMPOSITION

Finding parallelism in large computational problems is often conceptually simple but can be challenging in practice. The key is to identify the work

to be performed by each unit of parallel execution, which is a thread in CUDA, so that the inherent parallelism of the problem is well utilized. For example, in the electrostatic potential map calculation problem, it is clear that all atoms can be processed in parallel and all energy grid points can be calculated in parallel. However, one must take care when decomposing the calculation work into units of parallel execution, which will be referred to as *threading arrangement*. As we discussed in Section 12.2, the decomposition of the electrostatic potential map calculation problem can be atom-centric or grid-centric. In an atom-centric threading arrangement, each thread is responsible for calculating the effect of one atom on all grid points. In contrast, a grid-centric threading arrangement uses each thread to calculate the effect of all atoms on a grid point.

While both threading arrangements lead to similar levels of parallel execution and same execution results, they can exhibit very different performance in a given hardware system. The grid-centric arrangement has a memory access behavior called *gather*, where each thread gathers or collects the effect of input atoms into a grid point. Figure 13.1(a) illustrates the gather access behavior. Gather is a desirable thread arrangement in CUDA devices because the threads can accumulate their results in their private registers. Also, multiple threads share input atom values, and can effectively use constant memory caching or shared memory to conserve global memory bandwidth.

The atom-centric arrangement, on the other hand, exhibits a memory access behavior called *scatter*, where each thread scatters or distributes the
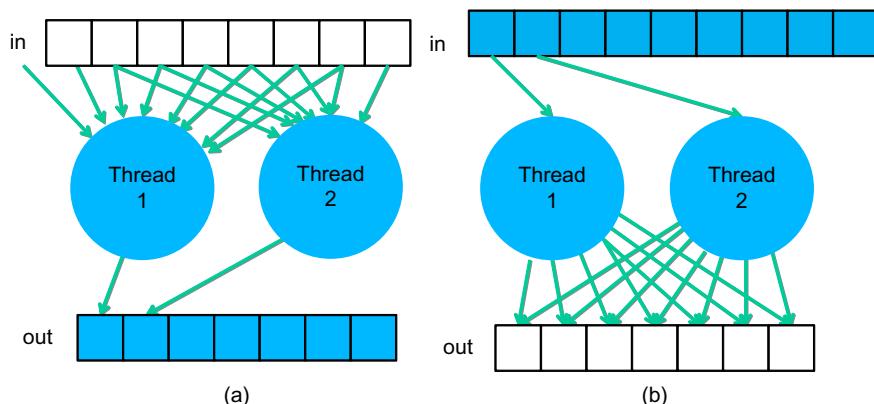
FIGURE 13.1

(a) Gather and (b) scatter based thread arrangements.

effect of an atom into grid points. The scatter behavior is illustrated in Figure 13.1(b). This is an undesirable arrangement in CUDA devices because the multiple threads can write into the same grid point at the same time. The grid points must be stored in a memory that can be written by all the threads involved. Atomic operations must be used to prevent race conditions and loss of value during simultaneous writes to a grid point by multiple threads. These atomic operations are much slower than the register accesses used in the atom-centric arrangement. Understanding the behavior of the threading arrangement and the limitations of hardware allows a parallel programmer to steer toward the more desired gather-based arrangement.

A real application often consists of multiple modules that work together. The electrostatic potential map calculation is one such module in molecular dynamics applications. Figure 13.2 shows an overview of major modules of a molecular dynamics application. For each atom in the system, the application needs to calculate the various forms of forces (e.g. vibrational, rotational, and nonbonded) that are exerted on the atom. Each form of force is calculated by a different method. At the high level, a programmer needs to decide how the work is organized. Note that the amount of work can vary dramatically between these modules. The non-bonded force calculation typically involves interactions among many atoms and incurs much more calculations than the vibrational and rotational forces. Therefore, these modules tend to be realized as separate passes over the force data structure. The programmer needs to decide if each pass is worth implementing in a CUDA device. For example, he or
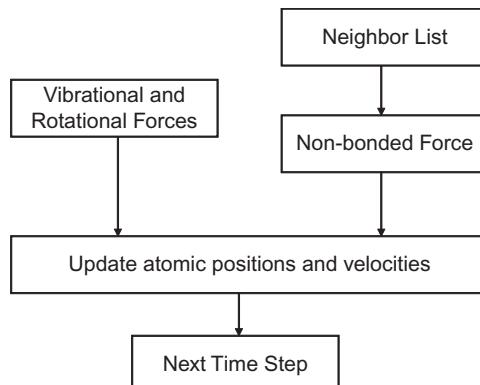
FIGURE 13.2

Major tasks of a molecular dynamics application.

she may decide that the vibrational and rotational force calculations do not involve a sufficient amount of work to warrant execution on a device. Such a decision would lead to a CUDA program that launches a kernel that calculates nonbonding forces for all the grid points while continuing to calculate the vibrational and rotational forces for the grid points on the host. The module that updates atomic positions and velocities may also run on the host. It first combines the vibrational and rotational forces from the host and the nonbonding forces from the device. It then uses the combined forces to calculate the new atomic positions and velocities.

The portion of work done by the device will ultimately decide the application-level speedup achieved by parallelization. For example, assume that the nonbonding force calculation accounts for 95% of the original sequential execution time and it is accelerated by $100\times$ using a CUDA device. Further assume that the rest of the application remains on the host and receives no speedup. The application-level speedup is $1/(5\% + 95\%/100) = 1/(5\% + 0.95\%) = 1/(5.95\%) = 17\times$. This is a demonstration of Amdahl's law: the application speedup due to parallel computing is limited by the sequential portion of the application. In this case, even though the sequential portion of the application is quite small (5%), it limits the application-level speedup to $17\times$ even though the nonbonding force calculation has a speedup of $100\times$. This example illustrates a major challenge in decomposing large applications: the accumulated execution time of small activities that are not worth parallel execution on a CUDA device can become a limiting factor in the speedup seen by the end users.

Amdahl's law often motivates task-level parallelization. Although some of these smaller activities do not warrant fine-grained massive parallel execution, it may be desirable to execute some of these activities in parallel with each other when the data set is large enough. This could be achieved by using a multicore host to execute such tasks in parallel. Alternatively, we could try to simultaneously execute multiple small kernels, each corresponding to one task. The previous CUDA devices did not support such parallelism but the new generation devices such as Kepler do.

An alternative approach to reducing the effect of sequential tasks is to exploit data parallelism in a hierarchical manner. For example, in a Message Passing Interface (MPI) [MPI2009] implementation, a molecular dynamics application would typically distribute large chunks of the spatial grids and their associated atoms to nodes of a networked computing cluster. By using the host of each node to calculate the vibrational and rotational force for its chunk of atoms, we can take advantage of multiple host CPUs to achieve speedup for these lesser modules. Each node can use a CUDA

device to calculate the nonbonding force at a higher level of speedup. The nodes will need to exchange data to accommodate forces that go across chunks and atoms that move across chunk boundaries. We will discuss more details of joint MPI-CUDA programming in Chapter 19. The main point here is that MPI and CUDA can be used in a complementary way in applications to jointly achieve a higher-level of speed with large data sets.

13.3 ALGORITHM SELECTION

An algorithm is a step-by-step procedure where each step is precisely stated and can be carried out by a computer. An algorithm must exhibit three essential properties: definiteness, effective computability, and finiteness. *Definiteness* refers to the notion that each step is precisely stated; there is no room for ambiguity as to what is to be done. *Effective computability* refers to the fact that each step can be carried out by a computer. *Finiteness* means that the algorithm must be guaranteed to terminate.

Given a problem, we can typically come up with multiple algorithms to solve the problem. Some require fewer steps of computation than others; some allow higher degrees of parallel execution than others; some have better numerical stability than others; and some consume less memory bandwidth than others. Unfortunately, there is often not a single algorithm that is better than others in all the four aspects. Given a problem and a decomposition strategy, a parallel programmer often needs to select an algorithm that achieves the best compromise for a given hardware system.
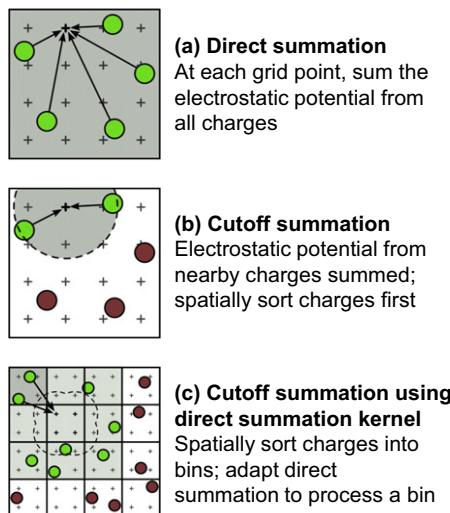
In our matrix−matrix multiplication example, we decided to decompose the problem by having each thread compute the dot product for an output element. Given this decomposition, we presented two different algorithms. The algorithm in Section 4.3 is a straightforward algorithm where every thread simply performs an entire dot product. Although the algorithm fully utilizes the parallelism available in the decomposition, it consumes too much global memory bandwidth. In Section 5.4, we introduced tiling, an important algorithm strategy for conserving memory bandwidth. Note that the tiled algorithm partitions the dot products into phases. All threads involved in a tile must synchronize with each other so that they can collaboratively load the tile of input data into the shared memory and collectively utilize the loaded data before they move on to the next phase. As we showed in Figure 5.12, the tiled algorithm requires each thread to execute more statements and incur more overhead in indexing the input arrays than the original algorithm. However, it runs much

faster because it consumes much less global memory bandwidth. In general, tiling is one of the most important algorithm strategies for matrix applications to achieve high performance.

As we demonstrated in Sections 6.4 and 12.3, we can systematically merge threads to achieve a higher level of instruction and memory access efficiency. In Section 6.4, threads that handle the same columns of neighboring tiles are combined into a new thread. This allows the new thread to access each M element only once while calculating multiple dot products, reducing the number of address calculation and memory load instructions executed. It also further reduces the consumption of global memory bandwidth. The same technique, when applied to the DCS kernel in electrostatic potential calculation, further reduces the number of distance calculations while achieving similar reduction in address calculations and memory load instructions.

One can often come up with even more aggressive algorithm strategies. An important algorithm strategy, referred to as *cutoff binning*, can significantly improve the execution efficiency of grid algorithms by sacrificing a small amount of accuracy. This is based on the observation that many grid calculation problems are based on physical laws where numerical contributions from particles or samples that are far away from a grid point can be collectively treated with an implicit method at much lower computational complexity. This is illustrated for the electrostatic potential calculation in Figure 13.3. Figure 13.3(a) shows the direct summation algorithms discussed in Chapter 12. Each grid point receives contributions from all atoms. While this is a very parallel approach and achieves excellent speedup over CPU-only execution for moderate-size energy grid systems, as we showed in Figure 12.11, it does not scale well to very large energy grid systems where the number of atoms increases proportional to the volume of the system. The amount of computation increases with the square of the volume. For large-volume systems, such an increase makes the computation excessively long even for massively parallel devices.

In practice, we know that each grid point needs to receive contributions from atoms that are close to it. The atoms that are far away from a grid point will have negligible contribution to the energy value at the grid point because the contribution is inversely proportional to the distance. Figure 13.3(b) illustrates this observation with a circle drawn around a grid point. The contributions to the grid point energy from atoms outside the circle (maroon) are negligible. If we can devise an algorithm where each grid point only receives contributions from atoms within a fixed radius of its coordinate (green), the computational complexity of the

(a) Direct summation
At each grid point, sum the electrostatic potential from all charges

(b) Cutoff summation
Electrostatic potential from nearby charges summed; spatially sort charges first

(c) Cutoff summation using direct summation kernel
Spatially sort charges into bins; adapt direct summation to process a bin

FIGURE 13.3

Cutoff summation algorithm.

algorithm would be reduced to linearly proportional to the volume of the system. This would make the computation time of the algorithm linearly proportional to the volume of the system. Such algorithms have been used extensively in sequential computation.

In sequential computing, a simple cutoff algorithm handles one atom at a time. For each atom, the algorithm iterates through the grid points that fall within a radius of the atom's coordinate. This is a straightforward procedure since the grid points are in an array that can be easily indexed as a function of their coordinates. However, this simple procedure does not carry easily to parallel execution. The reason is what we discussed in Section 13.2: the atom-centric decomposition does not work well due to it scatter memory access behavior. However, as we discussed in Chapter 9, it is important that a parallel algorithm matches the work efficiency of an efficient sequential algorithm.

Therefore, we need to find a cutoff binning algorithm based on the grid-centric decomposition: each thread calculates the energy value at one grid point. Fortunately, there is a well-known approach to adapting the direct summation algorithm, such as the one in Figure 12.9, into a cutoff binning algorithm. Rodrigues et al. present such an algorithm for the electrostatic potential problem [RSH 2008].

The key idea of the algorithm is to first sort the input atoms into bins according to their coordinates. Each bin corresponds to a box in the grid space and it contains all atoms of which the coordinate falls into the box. We define a "neighborhood" of bins for a grid point to be the collection of bins that contain all the atoms that can contribute to the energy value of a grid point. If we have an efficient way of managing neighborhood bins for all grid points, we can calculate the energy value for a grid point by examining the neighborhood bins for the grid point. This is illustrated in Figure 13.3(c). Although Figure 13.3(c) shows only one layer (2D) of bins that immediately surround that containing a grid point as its neighborhood, a real algorithm will typically have multiple layers (3D) of bins in a grid's neighborhood. In this algorithm, all threads iterate through their own neighborhood. They use their block and thread indices to identify the appropriate bins. Note that some of the atoms in the surrounding bins may not fall into the radius. Therefore, when processing an atom, all threads need to check if the atom falls into its radius. This can cause some control divergence among threads in a warp.

The main source of improvement in work efficiency comes from the fact that each thread now examines a much smaller set of atoms in a large grid system. This, however, makes constant memory much less attractive for holding the atoms. Since thread blocks will be accessing different neighborhoods, the limited-size constant memory will unlikely be able to hold all the atoms that are needed by all active thread blocks. This motivates the use of global memory to hold a much larger set of atoms. To mitigate the bandwidth consumption, threads in a block collaborate in loading the atom information in the common neighborhood into the shared memory. All threads then examine the atoms out of shared memory. Readers are referred to Rodrigues et al. [RHS2008] for more details of this algorithm.

One subtle issue with binning is that bins may end up with a different number of atoms. Since the atoms are statistically distributed in the grid system, some bins may have lots of atoms and some bins may end up with no atom at all. To guarantee memory coalescing, it is important that all bins are of the same size and aligned at appropriate coalescing boundaries. To accommodate the bins with the largest number of atoms, we would need to make the size of all other bins the same size. This would require us to fill many bins with dummy atoms of which the electrical charge is 0, which causes two negative effects. First, the dummy atoms still occupy global memory and shared memory storage. They also consumer data transfer bandwidth to the device. Second, the dummy atoms extend the execution time of the thread blocks of which the bins have few real atoms.

A well-known solution is to set the bin size at a reasonable level, typically much smaller than the largest possible number of atoms in a bin. The binning process maintains an overflow list. When processing an atom, if the atom's home bin is full, the atom is added to the overflow list instead. After the device completes a kernel, the result grid point energy values are transferred back to the host. The host executes a sequential cutoff algorithm on the atoms in the overflow list to complete the missing contributions from these overflow atoms. As long as the overflow atoms account for only a small percentage of the atoms, the additional sequential processing time of the overflow atoms is typically shorter than that of the device execution time. One can also design the kernel so that each kernel invocation calculates the energy values for a subvolume of grid points. After each kernel completes, the host launches the next kernel and processes the overflow atoms for the completed kernel. Thus, the host will be processing the overflow atoms while the device executes the next kernel. This approach can hide most if not all the delays in processing overflow atoms since it is done in parallel with the execution of the next kernel.

Figure 13.4 shows a comparison of scalability and performance of the various electrostatic potential map algorithms. Note that the CPU-SSE3 curve is based on a sequential cutoff algorithm. For a map with small volumes, around 1,000 Angstrom^3, the host (CPU with SSE) executes faster than the DCS kernel shown in Figure 13.4. This is because there is not enough work to fully utilize a CUDA device for such a small volume.
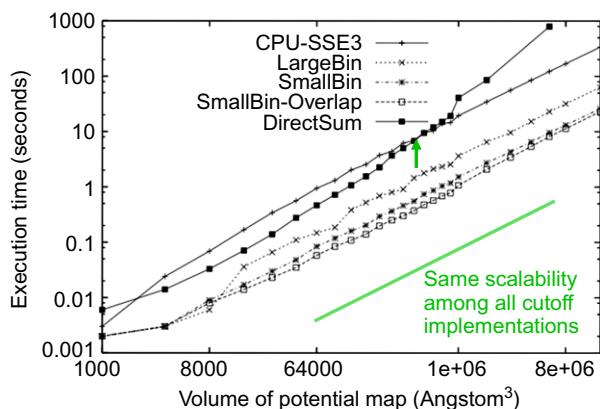
FIGURE 13.4

Scalability and performance of difference algorithms for calculating an electrostatic potential map.

However, for moderate volumes, between 2,000 and 500,000 Angstrom^3, the direct summation kernel performs significantly better than the host due to its massive execution. As we anticipated, the direct summation kernel scales poorly when the volume size reaches about 1,000,000 Angstrom^3, and runs longer than the sequential algorithm on the CPU! This is due to the fact that the algorithm complexity of the DCS kernel is higher than the sequential algorithm and thus the amount of work done by the kernel grows much faster than that done by the sequential algorithm. For a volume size larger than 1,000,000 Angstrom^3, the amount of work is so large that it swamps the hardware execution resources.

Figure 13.4 also shows the running time of three binned cutoff algorithms. The LargeBin algorithm is a straightforward adaptation of the DCS kernel for the cutoff. The kernel is designed to process a subvolume of the grid points. Before each kernel launch, the CPU transfers all atoms that are in the combined neighborhood of all the grid points in the subvolume. These atoms are still stored in the constant memory. All threads examine all atoms in the joint neighborhood. The advantage of the kernel is its simplicity. It is essentially the same as the direct summation kernel with a relatively large, preselected neighborhood of atoms. Note that the LargeBin approach performs reasonably well for moderate volumes and scales well for large volumes.

The SmallBin algorithm allows the threads running the same kernel to process a different neighborhood of atoms. This is the algorithm that uses global memory and shared memory for storing atoms. The algorithm achieves higher efficiency than the LargeBin algorithm because each thread needs to examine a smaller number of atoms. For moderate volumes, around 8,000 Angstrom^3, the LargeBin algorithm slightly outperforms the SmallBin algorithm. The reason is that the SmallBin algorithm does incur more instruction overhead for loading atoms from global memory into shared memory. For a moderate volume, there is a limited number of atoms in the entire system. The ability to examine a smaller number of atoms does not provide sufficient advantage to overcome the additional instruction overhead. However, the difference is so small at 8,000 Angstrom^3 that the SmallBin algorithm is still a clear win across all volume sizes. The SmallBin-Overlap algorithm overlaps the sequential overflow atom processing with the next kernel execution. It provides a slight but noticeable improvement in running time over the SmallBin algorithm. The SmallBin–Overlap algorithm achieves a $17\times$ speedup over an efficiently implemented sequential CPU-SSE cutoff algorithm, and maintains the same scalability for large volumes.

13.4 COMPUTATIONAL THINKING

Computational *thinking* is arguably the most important aspect of parallel application development [Wing2006]. We define computational thinking as the thought process of formulating domain problems in terms of computation steps and algorithms. Like any other thought processes and problem-solving skills, computational thinking is an art. As we mentioned in Chapter 1, we believe that computational thinking is best taught with an iterative approach where students bounce back and forth between practical experience and abstract concepts.

The electrostatic potential map kernels used in Chapter 12 and this chapter serve as good examples of computational thinking. To develop an efficient parallel application that solves the electrostatic potential map problem, one must come up with a good high-level decomposition of the problem. As we showed in Section 13.2, one must have a clear understanding of the desirable (e.g., gather in CUDA) and undesirable (e.g., scatter in CUDA) memory access behaviors to make a wise decision.

Given a problem decomposition, parallel programmers face a potentially overwhelming task of designing algorithms to overcome major challenges in parallelism, execution efficiency, and memory bandwidth consumption. There is a very large volume of literature on a wide range of algorithm techniques that can be hard to understand. It is beyond the scope of this book to have a comprehensive coverage of the available techniques. We did discuss a substantial set of techniques that have broad applicability. While these techniques are based on CUDA, they help readers build up the foundation for computational thinking in general. We believe that humans understand best when we learn from the bottom up. That is, we first learn the concepts in the context of a particular programming model, which provides us with solid footing before we generalize our knowledge to other programming models. An in-depth experience with the CUDA model also enables us to gain maturity, which will help us learn concepts that may not even be pertinent to the CUDA model.

There is a myriad of skills needed for a parallel programmer to be an effective computational thinker. We summarize these foundational skills as follows:

- *Computer architecture:* memory organization, caching and locality, memory bandwidth, SIMT versus SPMD versus SIMD execution, and floating-point precision versus accuracy. These concepts are critical in understanding the trade-offs between algorithms.

- *Programming models and compilers:* parallel execution models, types of available memories, array data layout, and thread granularity transformation. These concepts are needed for thinking through the arrangements of data structures and loop structures to achieve better performance.
- *Algorithm techniques:* tiling, cutoff, scatter−gather, binning, and others. These techniques form the toolbox for designing superior parallel algorithms. Understanding the scalability, efficiency, and memory bandwidth implications of these techniques is essential in computational thinking.
- *Domain knowledge:* numerical methods, precision, accuracy, and numerical stability. Understanding these ground rules allows a developer to be much more creative in applying algorithm techniques.

Our goal for this book is to provide a solid foundation for all the four areas. Readers should continue to broaden their knowledge in these areas after finishing this book. Most importantly, the best way of building up more computational thinking skills is to keep solving challenging problems with excellent computational solutions.

13.5 SUMMARY

In summary, we have discussed the main dimensions of algorithm selection and computational thinking. The key lesson is that given a problem decomposition decision, programmers will typically have to select from a variety of algorithms. Some of these algorithms achieve different trade-offs while maintaining the same numerical accuracy. Others involve sacrificing some level of accuracy to achieve much more scalable running times. The cutoff strategy is perhaps the most popular of such strategies. Even though we introduced cutoff in the context of electrostatic potential map calculation, it is used in many domains including ray tracing in graphics and collision detection in games. Computational thinking skills allow an algorithm designer to work around the roadblocks and reach a good solution.

13.6 EXERCISES

13.1 Write a host function to perform binning of atoms. Determine the representation of the bins as arrays. Think about coalescing

requirements. Make sure that every thread can easily find the bins it needs to process.

13.2 Write the part of the cutoff kernel function that determines if an atom is in the neighborhood of a grid point based on the coordinates of the atoms and the grid points.

References

Message Passing Interface Forum, MPI—A Message Passing Interface Standard Version 2.2, Available at: <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf >, 04.09.09.

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns of Parallel Programming,* Reading, MA: Addison-Wesley Professional.

Rodrigues, C. I., Stone, J., Hardy, D., Hwu, W. W.GPU Acceleration of Cutoff-Based Potential Summation. *ACM Computing Frontier Conference 2008,* Italy: May 2008.

Wing, J. (2006). Computational thinking. Communications of the ACM, *49*(3), pp. 33−35.

This page intentionally left blank

An Introduction to OpenCL™ 14

Now that we have discussed high-performance parallel programming using CUDA C, we would like to introduce another way to exploit the parallel computing capabilities of heterogeneous computing systems with GPUs and CPUs: OpenCL™. In this chapter, we will give a brief overview of OpenCL for CUDA programmers. The fundamental programming model of OpenCL is so similar to CUDA that there is a one-to-one correspondence for most features. With your understanding of CUDA, you will be able to start writing OpenCL programs with the material presented in this chapter. In our opinion, the best way to learn OpenCL is actually to learn CUDA first and then map the OpenCL features to their CUDA equivalents.

14.1 BACKGROUND

OpenCL is a standardized, cross-platform parallel computing API based on the C language. It is designed to enable the development of

portable parallel applications for systems with heterogeneous computing devices. The development of OpenCL was motivated by the need for a standardized high-performance application development platform for the fast-growing variety of parallel computing platforms. In particular, it addresses significant application portability limitations of the previous programming models for heterogeneous parallel computing systems.

CPU-based parallel programming models have been typically based on standards such as OpenMP but usually do not encompass the use of special memory types or SIMD (single instruction, multiple data) execution by high-performance programmers. Joint CPU−GPU heterogeneous parallel programming models such as CUDA have constructs that address complex memory hierarchies and SIMD execution but have been platform-, vendor-, or hardware-specific. These limitations make it difficult for an application developer to access the computing power of CPUs, GPUs, and other types of processing units from a single multiplatform source code base.

The development of OpenCL was initiated by Apple and managed by the Khronos Group, the same group that manages the OpenGL standard. On one hand, it draws heavily on CUDA in the areas of supporting a single code base for heterogeneous parallel computing, data parallelism, and complex memory hierarchies. This is the reason why a CUDA programmer will find these aspects of OpenCL familiar once we connect the terminologies. Readers will especially appreciate the similarities between OpenCL and the low-level CUDA driver model.

On the other hand, OpenCL has a more complex platform and device management model that reflects its support for multiplatform and multivendor portability. OpenCL implementations already exist on AMD/ATI and NVIDIA GPUs as well as X86 CPUs. In principle, one can envision OpenCL implementations on other types of devices such as digital signal processors (DSPs) and field programmable gate arrays (FPGAs). While the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come for free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. Also, many OpenCL features are optional and may not be supported on all devices. A portable OpenCL code will need to avoid using these optional features. However, some of these optional features allow applications to achieve significantly more performance in devices that support them. As a result, a portable OpenCL code may not be able to achieve its performance potential on any of the devices. Therefore, one should expect that a portable application that achieves high performance on multiple devices

will employ sophisticated runtime tests and chose among multiple code paths according to the capabilities of the actual device used.

The objective of this chapter is not to provide full details on all programming features of OpenCL. Rather, the objective is to give a CUDA programmer a conceptual understanding of the OpenCL programming model features. It also provides some basic host and kernel code patterns for jumpstarting an OpenCL coding project. With this foundation, readers can immediately start to program in OpenCL and consult the OpenCL specification [KHR] and programming guides [NVIDIA,AMD] on a need basis.
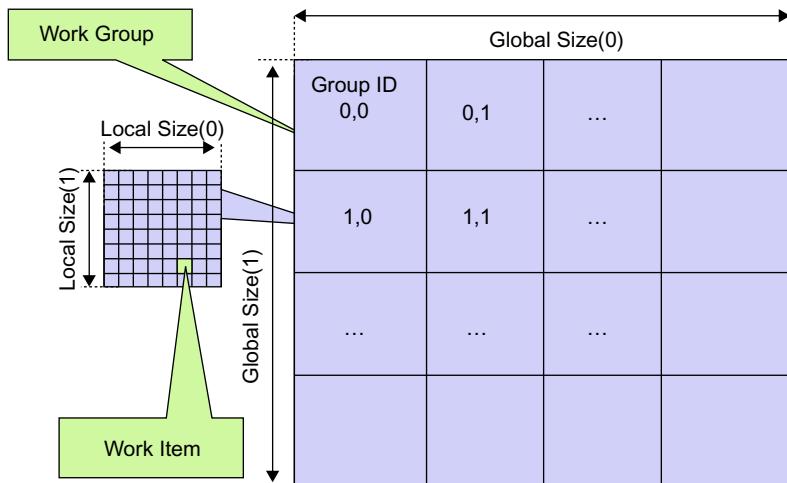
14.2 DATA PARALLELISM MODEL

OpenCL employs a data-parallel execution model that has direct correspondence with CUDA. An OpenCL program consists of two parts: kernels that execute on one or more OpenCL devices and a host program that manages the execution of kernels. Table 14.1 summarizes the mapping of OpenCL data parallelism concepts to their CUDA equivalents. Like CUDA, the way to submit work for parallel execution in OpenCL is for the host program to launch kernel functions. We will discuss the additional kernel preparation, device selection, and management work that an OpenCL host program needs to do as compared to its CUDA counterpart in Section 14.4.

When a kernel function is launched, its code is run by *work items*, which correspond to CUDA threads. An index space defines the work items and how data is mapped to the work items. That is, OpenCL work items are identified by global dimension index ranges (NDRanges). Work items form *work groups* that correspond to CUDA thread blocks. Work items in the

Table 14.1 Mapping between OpenCL and CUDA Data Parallelism Model Concepts

OpenCL Parallelism Concept	CUDA Equivalent
Kernel	Kernel
Host program	Host program
NDRange (index space)	Grid
Work item	Thread
Work group	Block

FIGURE 14.1

Overview of the OpenCL parallel execution model.

same work group can synchronize with each other using barriers that are equivalent to `__syncthreads()` in CUDA. Work items in different work groups cannot synchronize with each other except by terminating the kernel function and launching a new one. As we discussed in Chapter 4, this limited scope of barrier synchronization enables transparent scaling.

Figure 14.1 illustrates the OpenCL data-parallel execution model. Readers should compare Figure 14.1 with Figure 12.8 for similarities. The NDRange (CUDA grid) contains all work items (CUDA threads). For this example, we assume that the kernel is launched with a 2D NDRange.

All work items have their own unique global index values. There is a minor difference between OpenCL and CUDA in the way they manage these index values. In CUDA, each thread has a `blockIdx` value and a `threadIdx` value. The two values are combined to form a global thread ID value for the thread. For example, if a CUDA grid and its blocks are organized as 2D arrays, the kernel code can form a unique global thread index value in the x dimension as `blockIdx.x*blockDim.x + threadIdx.x`. These `blockIdx` and `threadIdx` values are accessible in a CUDA kernel as predefined variables.

In an OpenCL kernel, a thread can get its unique global index values by calling an API function `get_global_id()` with a parameter that identifies the dimension. See the `get_global_id(0)` entry in Table 14.2. The

Table 14.2 Mapping of OpenCL Dimensions and Indices to CUDA Dimensions and Indices
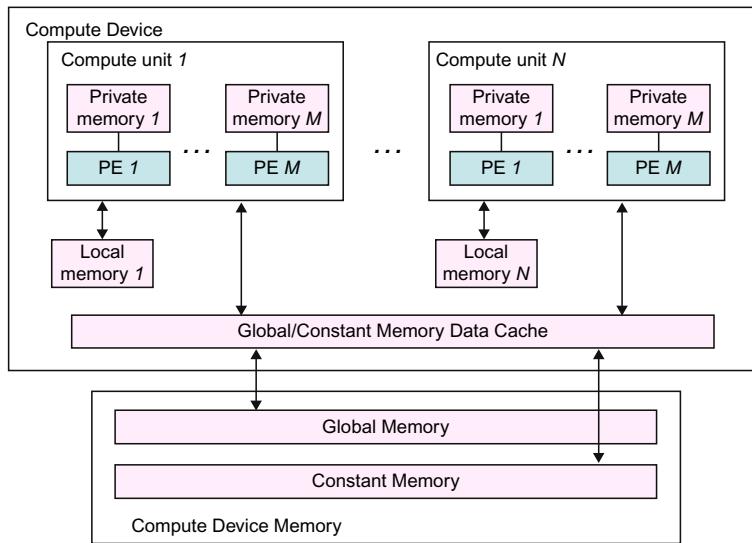
OpenCL API Call	Explanation	CUDA Equivalent
get_global_id (0)	Global index of the work item in the *x* dimension	blockIdx. x*blockDim. x + threadIdx.x
get_local_id(0)	Local index of the work item within the work group in the *x* dimension	threadIdx.x
get_global_size (0)	Size of NDRange in the *x* dimension	gridDim. x*blockDim.x
get_local_size (0)	Size of each work group in the *x* dimension	blockDim.x

functions get_global_id(0) and get_global_id(1) return the global thread index values in the *x* dimension and the *y* dimension, respectively. The global index value in the *x* dimension is equivalent to the blockIdx. x*blockDim.x + threadIdx.x in CUDA. See Table 14.2 for the get_local_id(0) function, which is equivalent to threadIdx.x. We did not show the parameter values in Table 14.2 for selecting the higher-dimension indices: 1 for the *y* dimension and 2 for the *z* dimension.

An OpenCL kernel can also call an API function get_global_size() with a parameter that identifies the dimensional sizes of its NDRanges. The calls get_global_size(0) and get_global_size(1) return the total number of work items in the *x* and *y* dimensions of the NDRanges. Note that this is slightly different from the CUDA gridDim values, which are in terms of blocks. The CUDA equivalent for the get_global_size(0) return value would be gridDim.x * blockDim.x.

14.3 DEVICE ARCHITECTURE

Like CUDA, OpenCL models a heterogeneous parallel computing system as a host and one or more OpenCL devices. The host is a traditional CPU that executes the host program. Figure 14.2 shows the conceptual architecture of an OpenCL device. Each device consists of one or more *compute units* (CUs) that correspond to CUDA streaming multiprocessors (SMs). However, a compute unit can also correspond to CPU cores or other types of execution units in compute accelerators such as DSPs and FPGAs.

FIGURE 14.2

Conceptual OpenCL device architecture.

Each compute unit, in turn, consists of one or more *processing elements* (PEs), which corresponds to the streaming processors (SPs) in CUDA. Computation on a device ultimately happens in individual PEs.

Like CUDA, OpenCL also exposes a hierarchy of memory types that can be used by programmers. Figure 14.2 illustrates these memory types: global, constant, local, and private. Table 14.3 summarizes the supported use of OpenCL memory types and the mapping of these memory types to CUDA memory types. The OpenCL global memory corresponds to the CUDA global memory. Like CUDA, the global memory can be dynamically allocated by the host program and supports read/write access by both host and devices.

Unlike CUDA, the constant memory can be dynamically allocated by the host. Like CUDA, the constant memory supports read/write access by the host and read-only access by devices. To support multiple platforms, OpenCL provides a device query that returns the constant memory size supported by the device.

The mapping of OpenCL local memory and private memory to CUDA memory types is more interesting. The OpenCL local memory actually

Table 14.3 Mapping of OpenCL Memory Types to CUDA Memory Types

Memory Type	Host Access	Device Access	CUDA Equivalent
Global memory	Dynamic allocation; read/write access	No allocation; read/write access by all work items in all work groups, large and slow, but may be cached in some devices	Global memory
Constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items	Constant memory
Local memory	Dynamic allocation; no access	Static allocation; shared read/write access by all work items in a work group	Shared memory
Private memory	No allocation; no access	Static allocation; read/write access by a single work item	Registers and local memory

corresponds to CUDA shared memory. The OpenCL local memory can be dynamically allocated by the host or statically allocated in the device code. Like the CUDA shared memory, the OpenCL local memory cannot be accessed by the host and it supports shared read/write access by all work items in a work group. The private memory of OpenCL corresponds to the CUDA automatic variables.

14.4 KERNEL FUNCTIONS

OpenCL kernels have an identical basic structure as CUDA kernels. All OpenCL kernel declarations start with a `__kernel` keyword, which is equivalent to the `__global__` keyword in CUDA. Figure 14.3 shows a simple OpenCL kernel that performs vector addition.

The function takes three arguments: pointers to the two input arrays and one pointer to the output array. The `__global` declarations in the function header indicate that the input and output arrays all reside in the global memory. Note that this keyword has the same meaning in OpenCL as in CUDA, except that there are two underscore characters (__) after the global keyword in CUDA.

The body of the kernel function is instantiated once for each work item. In Figure 14.3, each work item calls the `get_global_id(0)` function

```
__kernel void  vadd(__global const float *a,
    __global const float *b, __global float *result) {

    int i = get_global_id(0);
    result[i] = a[i] + b[i];
}
```

FIGURE 14.3

A simple OpenCL kernel example.
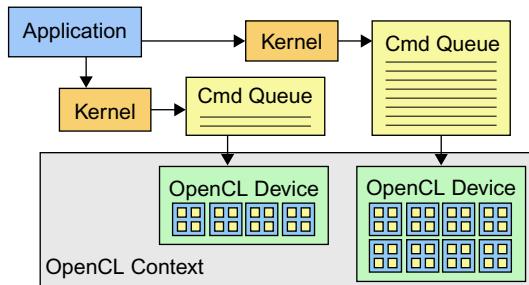
to receive their unique global index. This index value is then used by the work item to select the array elements to work on. Once the array element index i is formed, the rest of the kernel is virtually identical to the CUDA kernel.

14.5 DEVICE MANAGEMENT AND KERNEL LAUNCH

OpenCL defines a much more complex model of device management than CUDA. The extra complexity stems from the OpenCL support for multiple hardware platforms. OpenCL supports runtime construction and compilation of kernels to maximize an application's ability to address portability challenges across a wide range of CPUs and GPUs. Interested readers should refer to the OpenCL specification for more insight into the work that went into the OpenCL specification to cover as many types of potential OpenCL devices as possible [KHR2011].

In OpenCL, devices are managed through *contexts*. Figure 14.4 illustrates the main concepts of device management in OpenCL. To manage one or more devices in the system, the OpenCL programmer first creates a context that contains these devices. A context is essentially an address space that contains the accessible memory locations to the OpenCL devices in the system. This can be done by calling either clCreateContext() or clCreateContextFromType() in the OpenCL API.

Figure 14.5 show a simple host code pattern for managing OpenCL devices. In line 4, we use clGetContextInfo() to get the number of bytes needed (parmsz) to hold the device information, which is used in line 5 to allocate enough memory to hold the information about all the devices available in the system. This is because the amount of memory needed to hold the information depends on the number of OpenCL devices in the system. We then call clGetContextInfo() again in line 6 with the size of the device information and a pointer to the allocated memory for the

FIGURE 14.4

An OpenCL context is needed to manage devices.

```
...
1. cl_int clerr = CL_SUCCESS;

2. cl_context clctx=clCreateContextFromType(0, CL_DEVICE_TYPE_ALL,
      NULL, NULL, &clerr);

3. size_t parmsz;
4. clerr= clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

5. cl_device_id* cldevs= (cl_device_id *) malloc(parmsz);
6. clerr= clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);

7. cl_command_queue clcmdq=clCreateCommandQueue(clctx, cldevs[0], 0, &clerr);
```

FIGURE 14.5

Creating OpenCL context and command queue.

device information so that the function can deposit information on all the devices in the system into the allocated memory. An application could also use the `clGetDeviceIDs()` API function to determine the number and types of devices that exist in a system. Readers should read the *OpenCL Programming Guide* on the details of the parameters to be used for these functions [Khronos].

To submit work for execution by a device, the host program must first create a command queue for the device. This can be done by calling the `clCreateCommandQueue()` function is the OpenCL API. Once a command queue is created for a device, the host code can perform a sequence of API function calls to insert a kernel along with its execution configuration

parameters into the command queue. When the device is available for executing the next kernel, it removes the kernel at the head of the queue for execution.

Figure 14.5 shows a simple host program that creates a context for a device and submits a kernel for execution by the device. Line 2 shows a call to create a context that includes all OpenCL available devices in the system. Line 4 calls the `clGetContextInfo()` function to inquire about the number of devices in the context. Since line 2 asks that all OpenCL available devices be included in the context, the application does not know the number of devices actually included in the context after the context is created. The second argument of the call in line 4 specifies that the information being requested is the list of all devices included in the context. However, the fourth argument, which is a pointer to a memory buffer where the list should be deposited, is a NULL pointer. This means that the call does not want the list itself. The reason is that the application does not know the number of devices in the context and does not know the size of the memory buffer required to hold the list.

Rather, line 4 provides a pointer to the variable `parmsz`. After line 4, the `parmsz` variable holds the size of the buffer needed to accommodate the list of devices in the context. The application now knows the amount of memory buffer needed to hold the list of devices in the context. It allocates the memory buffer using `parmsz` and assigns the address of the buffer to the pointer variable `cldevs` at line 5.

Line 6 calls `clGetContextInfo()` again with the pointer to the memory buffer in the fourth argument and the size of the buffer in the third argument. Since this is based on the information from the call at line 4, the buffer is guaranteed to be the right size for the list of devices to be returned. The `clGetContextInfo` function now fills the device list information into the memory buffer pointed to by `cldevs`.

Line 7 creates a command queue for the first OpenCL device in the list. This is done by treating `cldevs` as an array of which the elements are descriptors of OpenCL devices in the system. Line 7 passes `cldevs[0]` as the second argument into the `clCreateCommandQueue(0)` function. Therefore, the call generates a command queue for the first device in the list returned by the `clGetContextInfo()` function.

Readers may wonder why we did not need to see this complex sequence of API calls in our CUDA host programs. The reason is that we have been using the CUDA runtime API that hides all this complexity for the common case where there is only one CUDA device in the system. The kernel launch in CUDA handles all the complexities on behalf of the

host code. If the developer wanted to have direct access to all CUDA devices in the system, he or she would need to use the CUDA driver API, where similar API calling sequences would be used. To date, OpenCL has not defined a higher-level API that is equivalent to the CUDA runtime API. Until such a higher-level interface is available, OpenCL will remain much more tedious to use than the CUDA runtime API. The benefit, of course, is that an OpenCL application can execute on a wide range of devices.

14.6 ELECTROSTATIC POTENTIAL MAP IN OPENCL

We now present an OpenCL case study based on the DCS kernel in Figure 12.9. This case study is designed to give a CUDA program a practical, top-to-bottom experience with OpenCL. The first step in porting the kernel to OpenCL is to design the organization of the NDRange, which is illustrated in Figure 14.5. The design is a straightforward mapping of CUDA threads to OpenCL work items and CUDA blocks to OpenCL work groups. As shown in Figure 14.6, each work item will calculate up to eight grid points and each work group will have 64 to 256 work items. All the efficiency considerations in Chapter 12 also apply here.

The work groups are assigned to the CUs the same way that CUDA blocks are assigned to the SMs. Such assignment is illustrated in
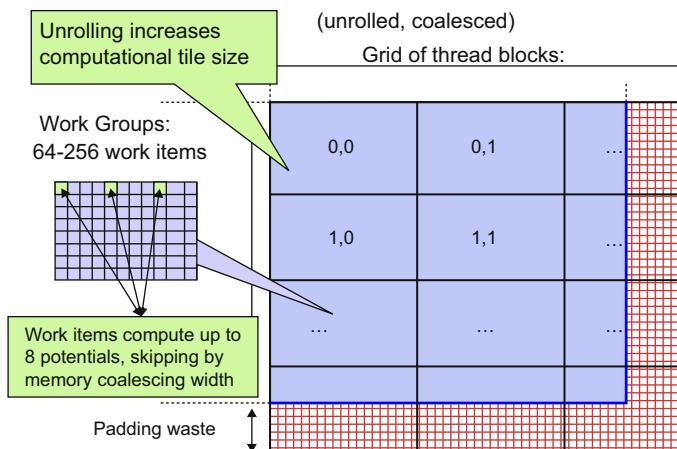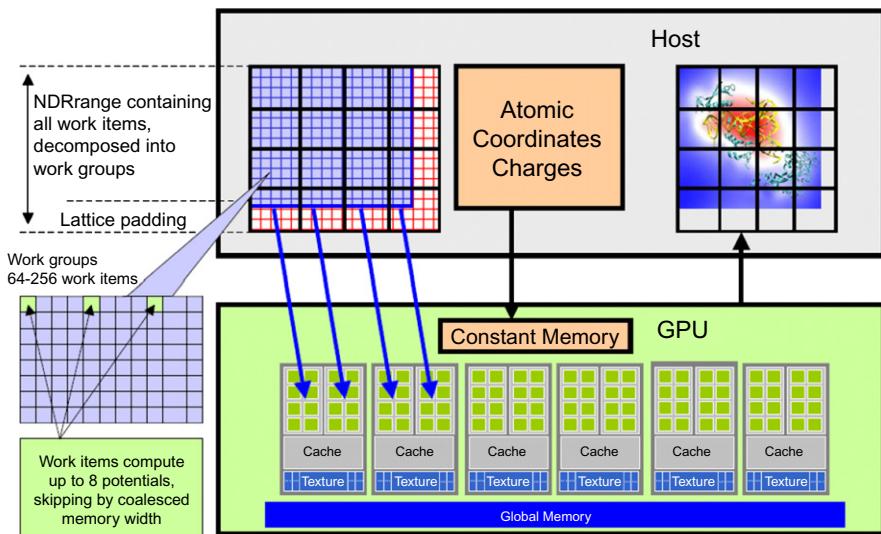
FIGURE 14.6

DCS kernel version 3 NDRange configuration.

FIGURE 14.7

Mapping DCS NDRange to OpenCL device.

Figure 14.7. One can use the same methodology used in Chapters 6 and 12 to derive high-performance OpenCL DCS kernel. Although the syntax is different, the underlying thought process involved in developing a high-performance OpenCL kernel is very much the same as CUDA.

The OpenCL kernel function implementation matches closely the CUDA implementation. Figure 14.8 shows the key differences. One is the `__kernel` keyword in OpenCL versus the `__global` keyword in CUDA. The main difference lies in the way the data access indices are calculated. In this case, the OpenCL `get_global_id(0)` function returns the equivalent of CUDA `blockIdx.x*blockDim.x + threadIdx.x`.

Figure 14.9 shows the inner loop of the OpenCL kernel. Readers should compare this inner loop with the CUDA code in Figure 12.9. The only difference is that the `__rsqrt()` call has been changed to the `native_rsqrt()` call, the OpenCL way for using the hardware implementation of math functions on a particular device.

OpenCL adopts a dynamic compilation model. Unlike CUDA, the host program can explicitly compile and create a kernel program at runtime. This is illustrated in Figure 14.10 for the DCS kernel. Line 1 declares the entire OpenCL DCS kernel source code as a string. Line 3 delivers the

```
Device
OpenCL:
__kernel voidclenergy(…) {
unsigned int xindex= get_global_id(0);
unsigned int yindex= get_global_id(1);
unsigned int outaddr= get_global_size(0) * UNROLLX
*yindex+xindex;

CUDA:
__global__ void cuenergy(…) {
Unsigned int xindex= blockIdx.x *blockDim.x +threadIdx.x;
unsigned int yindex= blockIdx.y *blockDim.y +threadIdx.y;
unsigned int outaddr= gridDim.x *blockDim.x *
UNROLLX*yindex+xindex
```

FIGURE 14.8

Data access indexing in OpenCL and CUDA.

```
…
for (atomid=0; atomid<numatoms; atomid++) {
float dy = coory -atominfo[atomid].y;
float dyz2= (dy * dy) + atominfo[atomid].z;
float dx1 = coorx –atominfo[atomid].x;
float dx2 = dx1 + gridspacing_coalesce;
float dx3 = dx2 + gridspacing_coalesce;
float dx4 = dx3 + gridspacing_coalesce;
float charge = atominfo[atomid].w;
energyvalx1 += charge* native_rsqrt(dx1*dx1 + dyz2);
energyvalx2 += charge* native_rsqrt(dx2*dx2 + dyz2);
energyvalx3 += charge* native_rsqrt(dx3*dx3 + dyz2);
energyvalx4 += charge* native_rsqrt(dx4*dx4 + dyz2);
}
```

FIGURE 14.9

Inner loop of the OpenCL DCS kernel.

source code string to the OpenCL runtime system by calling the clCreateProgramWith Source() function. Line 4 sets up the compiler flags for the runtime compilation process. Line 5 invokes the runtime compiler to build the program. Line 6 requests that the OpenCL runtime create the kernel and its data structures so that it can be properly launched. After line 6, clkern points to the kernel that can be submitted to a command queue for execution.

Figure 14.11 shows the host code that launches the DCS kernel. It assumes that the host code for managing OpenCL devices in Figure 14.5

FIGURE 14.10

Building OpenCL kernel.

```
 1. doutput= clCreateBuffer(clctx, CL_MEM_READ_WRITE,volmemsz,
        NULL, NULL);
 2. datominfo= clCreateBuffer(clctx, CL_MEM_READ_ONLY,
        MAXATOMS *sizeof(cl_float4), NULL, NULL);
 …
 3. clerr= clSetKernelArg(clkern, 0,sizeof(int), &runatoms);
 4. clerr= clSetKernelArg(clkern, 1,sizeof(float), &zplane);
 5. clerr= clSetKernelArg(clkern, 2,sizeof(cl_mem), &doutput);
 6. clerr= clSetKernelArg(clkern, 3,sizeof(cl_mem), &datominfo);
 7. cl_event event;
 8. clerr= clEnqueueNDRangeKernel(clcmdq,clkern, 2, NULL,
        Gsz,Bsz, 0, NULL, &event);
 9. clerr= clWaitForEvents(1, &event);
10. clerr= clReleaseEvent(event);
 …
11. clEnqueueReadBuffer(clcmdq,doutput, CL_TRUE, 0,
        volmemsz, energy, 0, NULL, NULL);
12. clReleaseMemObject(doutput);
13. clReleaseMemObject(datominfo);
```

FIGURE 14.11

OpenCL host code for kernel launch and

has been executed. Lines 1 and 2 allocate memory for the energy grid data and the atom information. The `clCreateBuffer()` function corresponds to the `cudaMalloc()` function. The constant memory is implicitly requested by setting the mode of access to read only for the `atominfo` array. Note that each memory buffer is associated with a context, which is specified by the first argument to the `clCreateBuffer()` function call.

Lines 3−6 in Figure 14.11 set up the arguments to be passed into the kernel function. In CUDA, the kernel functions are launched with C function call syntax extended with $<<<>>>$, which is followed by the regular list of arguments. In OpenCL, there is no explicit call to kernel functions. Therefore, one needs to use the `clSetKernelArg()` functions to set up the arguments for the kernel function.

Line 8 in Figure 14.11 submits the DCS kernel for launch. The arguments to the `clEnqueueNDRangeKernel()` function specifies the command queue for the device that will execute the kernel, a pointer to the kernel, and the global and local sizes of the NDRange. Lines 9 and 10 check for errors if any.

Line 11 transfers the contents of the output data back into the energy array in the host memory. The OpenCL `clEnqueueReadBuffer()` copies data from the device memory to the host memory and corresponds to the device the host direction of the `cudaMemcpy()` function.

The `clReleaseMemObject()` function is a little more sophisticated than `cudaFree()`. OpenCL maintains a reference count for data objects. OpenCL host program modules can retain (`clRetainMemObject()`) and release (`clReleaseMemObject()`) data objects. Note that `clCreateBuffer()` also serves as a retain call. With each retain call, the reference count of the object is incremented. With each release call, the reference count is decremented. When the reference count for an object reaches 0, the object is freed. This way, a library module can "hang on" to a memory object even though the other parts of the application no longer need the object and thus have released the object.

14.7 SUMMARY

OpenCL is a standardized, cross-platform API designed to support portable parallel application development on heterogeneous computing systems. Like CUDA, OpenCL addresses complex memory hierarchies and data-parallel execution. It draws heavily on the CUDA driver API experience. This is the reason why a CUDA programmer finds these

aspects of OpenCL familiar. We have seen this through the mappings of the OpenCL data parallelism model concepts, NDRange API calls, and memory types to their CUDA equivalents.

On the other hand, OpenCL has a more complex device management model that reflects its support for multiplatform and multivendor portability. While the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come for free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. We see that the OpenCL device management model, the OpenCL kernel compilation model, and the OpenCL kernel launch are much more complex than their CUDA counterparts.

We have by no means covered all the programming features of OpenCL. Readers are encouraged to read the OpenCL specification [KHR2011] and tutorials [Khronos] for more OpenCL features. In particular, we recommend that readers pay special attention to the device query, object query, and task parallelism model.

14.8 EXERCISES

14.1 Use the code base in Appendix A and examples in Chapters 3, 4, 5, and 6 to develop an OpenCL version of the matrix−matrix multiplication application.

14.2 Read the "OpenCL Platform Layer" section of the OpenCL specification. Compare the platform querying API functions with what you have learned in CUDA.

14.3 Read the "Memory Objects" section of the OpenCL specification. Compare the object creation and access API functions with what you have learned in CUDA.

14.4 Read the "Kernel Objects" section of the OpenCL specification. Compare the kernel creation and launching API functions with what you have learned in CUDA.

14.5 Read the "OpenCL Programming Language" section of the OpenCL specification. Compare the keywords and types with what you have learned in CUDA.

References

AMD OpenCL Resources. Available at: <http://developer.amd.com/gpu/ATIStreamSDK/pages/TutorialOpenCL.aspx>.

Khronos Group, The OpenCL Specification version 1.1, rev44. Available at: <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>, 2011.

Khronos OpenCL samples, tutorials, etc., Available at: <http://www.khronos.org/developers/resources/opencl/>.

NVIDIA OpenCL Resources. Available at: <http://www.nvidia.com/object/cuda_opencl.html>.

This page intentionally left blank

Parallel Programming with OpenACC

15

With special contributions from Yuan Lin and Vinod Grover

CHAPTER OUTLINE

The OpenACC Application Programming Interface (API) provides a set of compiler directives, library routines, and environment variables that can be used to write data-parallel FORTRAN, C, and C++ programs that run on accelerator devices, including GPUs. It is an extension to the host language. The OpenACC specification was initially developed by the Portland Group (PGI), Cray Inc., and NVIDIA, with support from CAPS enterprise. This chapter presents an introduction to OpenACC to parallel programmers who are already familiar with CUDA C.

15.1 OPENACC VERSUS CUDA C

One big difference between OpenACC and CUDA C is the use of compiler directives in OpenACC. To understand what a compiler directive is and the advantages of using compiler directives, let's take a look at our first OpenACC program in Figure 15.1, which does the matrix multiplication that we've already seen before.

```
1  void computeAcc(float *P, const float *M, const float *N, int Mh,
     int Mw, int Nw)
2  {
3
4    #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
       copyout(P[0:Mh*Nw])
5    for (int i=0; i<Mh; i++) {
6        #pragma acc loop
7        for (int j=0; j<Nw; j++) {
8            float sum = 0;
9            for (int k=0; k<Mw; k++) {
10               float a = M[i*Mw+k];
11               float b = N[k*Nw+j];
12               sum += a*b;
13           }
14           P[i*Nw+j] = sum;
15       }
16   }
17 }
```

FIGURE 15.1

Our first OpenACC program.

The code in the figure is almost identical to the sequential version, except for the two lines with #pragma at lines 4 and 6. In C and C++, the #pragma directive is the method to provide, to the compiler, information that is not specified in the standard language. OpenACC uses the compiler directive mechanism to extend the base language. In this example, the #pragma at line 4 tells the compiler to generate code for the i loop at lines 5-16 so that the loop iterations are executed in parallel on the accelerator. The copyin clause and the copyout clause specify how the matrix data should be transferred between the host and the accelerator. The #pragma at line 6 instructs the compiler to map the inner j loop to the second level of parallelism on the accelerator.

Compared with CUDA C/C++/FORTRAN, by using compiler directives, OpenACC brings quite a few benefits to programmers:

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives. They can leave most of the heavy lifting to the OpenACC compiler. The details of data transfer between host and accelerator memories, data caching, kernel launching, thread scheduling, and parallelism mapping are all handled by OpenACC compiler and runtime. The entry barrier of heterogeneous programmers for accelerators becomes much lower with OpenACC.

- OpenACC provides an incremental path for moving legacy applications to accelerators. This is attractive because adding directives disturbs the existing code less than other approaches. Some existing scientific applications are large and their developers don't want to rewrite them for accelerators. OpenACC lets these developers keep their applications looking like normal C, C + +, or FORTRAN code, and they can go in and put the directives in the code where they are needed one place a time.
- A non-OpenACC compiler is not required to understand and process OpenACC directives, therefore it can just ignore the directives and compile the rest of the program as usual. By using the compiler directive approach, OpenACC allows a programmer to write OpenACC programs in such a way that when the directives are ignored, the program can still run sequentially and gives the same result as when the program is run in parallel. Parallel programs that have equivalent sequential versions are much easier to debug than those that don't have. The matrix multiplication code in Figure 15.1 has this property— the code gives the same result regardless of whether lines 4 and 6 are honored or not. Such programs essentially have both the parallel version and the sequential version in one. OpenACC permits a common code base for accelerated and nonaccelerated enabled systems.

OpenACC users need to be aware of the following issues:

- Some OpenACC directives are hints to the OpenACC compiler, which may or may not be able to take full advantage of such hints. Therefore, the performance of an OpenACC program depends more on the capability of the OpenACC compiler used. On the other hand, a CUDA C/C + +/FORTRAN program expresses parallelism explicitly and relies less on the compiler for parallel performance.
- While it is possible to write OpenACC programs that give the same execution result as when the directives are ignored, this property does not hold automatically for all OpenACC programs. If compiler directives are ignored, some OpenACC programs may give different results or some may not work correctly.

In the rest of this chapter, we first explain the execution model and memory model used by OpenACC. We then walk through some concrete code examples to illustrate usage of some of the more commonly used OpenACC directives and APIs. We also show how an OpenACC implementation can map *parallel regions* and *kernel regions* to the CUDA GPU architecture. We believe certain behind-the-scenes knowledge can help users to get the

better performance out of OpenACC implementations. We conclude this article by outlining the future directions we see OpenACC going in.

15.2 EXECUTION MODEL

The OpenACC target machine has a host and an attached accelerator device, such as a GPU. Most accelerator devices can support multiple levels of parallelism. Figure 15.2 illustrates a typical accelerator that supports three levels of parallelism. At the outermost coarse-grain level, there are multiple execution units. Within each execution unit, there are multiple threads. At the innermost level, each thread is capable of executing vector operations. Currently, OpenACC does not assume any synchronization capability on the accelerator, except for thread forking and joining. Once work is distributed among the execution units, they will execute in parallel from start to finish. Similarly, once work is distributed among the threads within an execution unit, the threads execute in parallel. Vector operations are executed in lockstep.

An OpenACC program starts its execution on the host single-threaded (Figure 15.3). When the host thread encounters a `parallel` or a `kernels` construct, a *parallel region* or a *kernels region* that comprises all the code enclosed in the construct is created and launched on the accelerator device. The parallel region or kernels region can optionally execute asynchronously with the host thread and join with the host thread at a future synchronization point. The *parallel region* is executed entirely on the accelerator device.
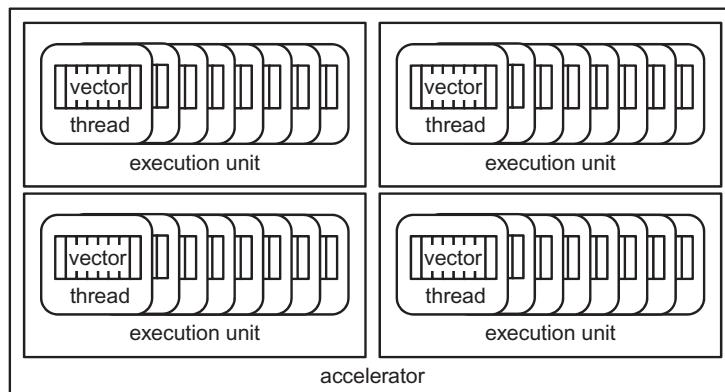
FIGURE 15.2

A typical accelerator device

The *kernels region* may contain a sequence of kernels, each of which is executed on the accelerator device.

The kernel execution follows a fork-join model. A group of gangs are used to execute each kernel. A group of workers can be forked to execute a parallel work-sharing loop that belongs to a gang. The workers are disbanded when the loop is done. Typically a gang executes on one execution unit, and a worker runs on one thread within an execution unit.

The programmer can instruct how the work within a parallel region or a kernels region is to be distributed among the different levels of parallelism on the accelerator.

15.3 MEMORY MODEL

In an OpenACC memory model, the host memory and the device memory are treated as separated. It is assumed that the host is not able to access device memory directly and the device is not able to access host memory directly. This is to ensure that the OpenACC programming model can support a wide range of accelerator devices, including most of the current GPUs that do not have the capability of unified memory access between
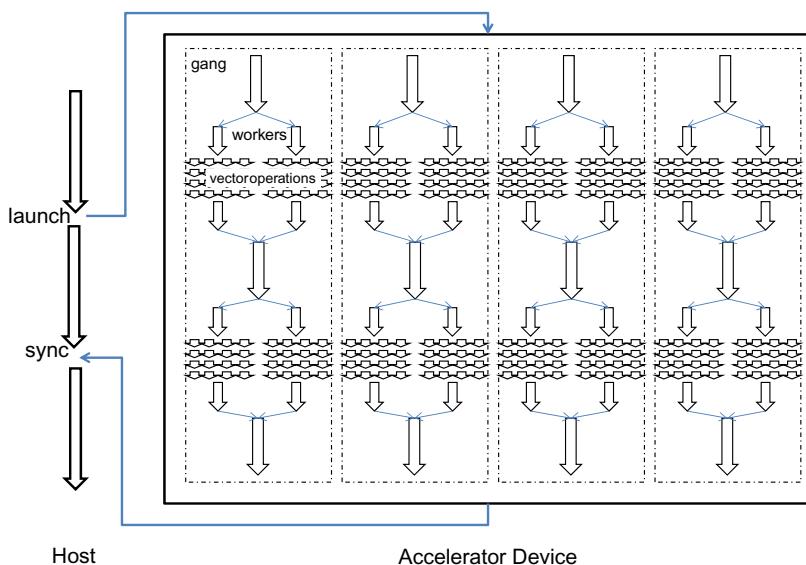
FIGURE 15.3

OpenACC execution model.

GPUs and CPUs. The unified virtual addressing and the GPUDirect introduced by NVIDIA in CUDA 4.0 allow a single virtual address space for both host memory and device memory and allow direct cross-device memory access between different GPUs. However, cross-host and device memory access is still not possible.

Just like in CUDA C/C++, in OpenACC input data needs to be transferred from the host to the device before kernel launches and result data needs to be transferred back from the device to the host. However, unlike in CUDA C/C++ where programmers need to explicitly code data movement through API calls, in OpenACC they can just annotate which memory objects need to be transferred, as shown by line 4 in Figure 15.1. The OpenACC compiler will automatically generate code for memory allocation, copying, and de-allocation.

OpenACC adopts a fairly weak consistency model for memory on the accelerator device. Although data on the accelerator can be shared by all execution units, OpenACC does not provide a reliable way to allow one execution unit to consume the data produced by another execution unit. There are two reasons for this. First, recall OpenACC does not provide any mechanism for synchronization between execution units. Second, memories between different execution units are not coherent. Although some hardware provides instructions to explicitly invalidate and update cache, they are not exposed at the OpenACC level. Therefore, in OpenACC, different execution units are expected to work on disjoint memory sets. Threads within an execution unit can also share memory and threads have coherent memory. However, OpenACC currently only mandates a memory fence at the thread fork and join, which are also the only synchronizations OpenACC provides for threads. While the device memory model may appear very limiting, it is not so in practice. For data−race free OpenACC data-parallel applications, the weak memory model works quite well.

15.4 BASIC OPENACC PROGRAMS

In this section, we will dive in to details of how one can write basic OpenACC programs.

Parallel Construct
Parallel Region, Gangs, and Workers
The single #pragma at line 4 in Figure 15.1 is actually a syntax sugar of two #pragma (Figure 15.4). We explain the parallel construct here and will explain the loop construct in the next section.

```
#pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
     copyout(P[0:Mh*Nw])
for (int i=0; i<Mh; i++) {
     …
}

#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
     copyout(P[0:Mh*Nw])
{
     #pragma acc loop
     for (int i=0; i<Mh; i++) {
          …
     }
}
```

FIGURE 15.4

#pragma acc parallel loop is #pragma acc parallel and #pragma acc loop in one.

```
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
a = 23;
}
```

FIGURE 15.5

Redundant execution in a parallel region.

The parallel construct is one of the two constructs (the other is the kernels construct) that can be used to specify which part of the program is to be executed on the accelerator. When a program encounters a parallel construct, the execution of the code within the structured block of the construct (also called a *parallel region*) is moved to the accelerator. Gangs of workers on the accelerator are created to execute the parallel region, as shown in Figure 15.3. Initially only one worker (let us call it a gang lead) within each gang will execute the parallel region. The other workers are conceptually idle at this point. They will be deployed when there is more parallel work at an inner level. The number of gangs can be specified by the num_gangs clause, and the number of workers within each gang can be specified by the num_workers clause.

In this example (Figure 15.5), a total of $1,024 \times 32 = 32,768$ workers are created. The $a = 23$ statement will be executed in parallel and redundantly by 1,024 gang leads. You may ask why anyone would want to write accelerator code like this. The usefulness of the parallel construct will become clear when it is used with the loop construct.

If a `parallel` construct does not have an explicit `num_gangs` clause or an explicit `num_workers` clause, then the implementation will pick the numbers at runtime. Once a parallel region starts executing, the number of gangs and the number of workers within each gang remain fixed during the execution of the parallel region. This is similar to CUDA in which once a kernel is launched, the number of blocks in the grid and the number of threads in a block cannot be changed.

Loop Construct
Gang Loop

Suppose you have a loop where all the iterations can be independently executed in parallel and you want to speed up its execution by running it on the accelerator. Can you write the code like Figure 15.6?

As we learned from the previous section, although the loop will be executed on the accelerator, you won't get any speedup because all 2,048 iterations will be executed sequentially and redundantly by the 1,024 gang leads. To get speedup, you need to distribute the 2,048 iterations among the gangs. And to do that, you need to use the gang `loop` construct, as shown in Figure 15.7.

```
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        …
    }
}
```

FIGURE 15.6

An unannotated loop in a parallel region is also redundantly executed.

```
#pragma acc parallel num_gangs(1024)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        …
    }
}
```

FIGURE 15.7

Use the `loop` construct to make a loop work-shared.

A gang `loop` construct is always associated with a loop. The gang `loop` construct is a *work-sharing* construct. The compiler and runtime will make sure that the iterations of a gang loop are shared among all gang leads encountered in the `loop` construct. In Figure 15.7, because 1,024 gang leads will encounter the `loop` construct, each lead will be assigned two iterations. Now the execution of the parallel loop will be more efficient and likely to achieve speedup.

Worker Loop

What if you also have an inner loop that can be executed in parallel? Well, that is when the worker `loop` construct can be beneficial.

The `worker loop` construct is also a work-sharing construct. The compiler and runtime will make sure that the iterations of a worker loop are shared among all workers within a gang. In Figure 15.8, the 32 workers in a gang will work collectively on the 512 iterations of the j loop in each of the two iterations of the i loop assigned to the gang. A total of $2,048 \times 512 = 1$ M instances of `foo()` will be executed in the sequential version or the parallel version. In the parallel version, $1,024 \times 32 = 32$ K workers are used and each worker will execute 1 M $\div 32$ K $= 32$ instances of `foo()`.

OpenACC Versus CUDA

Readers who are familiar with CUDA C may ask how the OpenACC code in Figure 15.8 is different from the CUDA C code in Figure 15.9? They may wonder, can't I just write the CUDA C version and achieve the same effect?

Yes, they are similar in this case. And as a matter of fact, some OpenACC implementations may actually generate the CUDA C version in Figure 15.9 from the OpenACC version in Figure 15.8 and pass it to the

```
#pragma acc parallel num_gangs(1024)
num_workers(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            foo(i,j);
        }
    }
}
```

FIGURE 15.8

Using the `worker` clause.

```
__global__ void kernel(…)
{
    for (int ii=0; ii<2; ii++) {
        int i=blockidx.x*2+ii;
        for (int jj=0; jj<16; jj++) {
            int j=threadidx.x*16+jj;
            foo(i,j);
        }
    }
}
… = kernel<<<1024, 32>>>(…);
```

FIGURE 15.9

A possible CUDA C implementation of the parallel region in Figure 15.8.

```
{
    Statement 1;
    Statement 2;
    for (int i=0; i<n; i++) {
        Statement 3;
        Statement 4;
    }
    Statement 5;
    Statement 6;
    for (int i=0; i<m; i++) {
        Statement 7;
        Statement 8;
    }
    Statement 9;
    if (condition)
        Statement 10;
}
```

FIGURE 15.10

A piece of nontrivial code.

CUDA C compiler. But one clear advantage of the OpenACC version is that it is much closer to the sequential version than the CUDA C version. Only a few code modifications are required. Compared with CUDA C, OpenACC gives you less control of how the final code on the accelerator will be. However, the strength of OpenACC lies in its ability to tackle more complicated existing sequential code, especially when the original code you want to port to execute on the accelerator is not a perfectly nested loop nest.

Take the code snippet in Figure 15.10 for example. Let's assume both loops are parallel loops. If you want to move the whole code snippet to

```
#pragma acc parallel num_gangs(32)
{
    Statement 1;
    Statement 2;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 3;
        Statement 4;
    }
    Statement 5;
    Statement 6;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 7;
        Statement 8;
    }
    Statement 9;
    if (condition)
        Statement 10;
}
```

FIGURE 15.11

Porting is easier with OpenACC (Part 1).

execute on the accelerator, it is much easier with OpenACC. If the code can give you the same result when statements 1, 2, 5, 6, 9, and 10 are executed redundantly by multiple gang leaders, then you can do what is shown in Figure 15.11.

The first pragma in Figure 15.11 creates 32 gangs. Statements 1 and 2 are executed by all gangs. Note that in the original code, these statements are executed only once. However, after the annotation, the compiler will generate code that executes these statements 32 times. This is equivalent to moving a statement into a loop. As long as the statement can be executed extra times without producing incorrect results, this is not a problem.

The second pragma in Figure 15.11 assigns the work of the for loop to the 32 gangs. Each gang will further distribute its share of the work to multiple workers. The exact number of workers in each gang will likely be decided at runtime when the number of iterations and the number of execution units are known.

If statements 1, 2, 5, 6, 9, and 10 can only be executed once, then you can still make the annotations shown in Figure 15.12. In this case, only one gang with 32 workers will be created. The gang leader will execute statements 1, 2, 5, 6, 9, and 10. It will assign the work for the two for

loops to its 32 workers. Obviously, the number of workers will be much lower than the previous case, which employs 32 gangs, each of which has multiple workers.

The important point here is that to achieve the same effect with CUDA, more significant code changes are required between the two cases. In the first case, statements 1 and 2 need to be pushed into the loop so that a kernel can be formed with statements 1, 2, 3, and 4. Similarly, another kernel needs to be formed with statements 5, 6, 7, 8, 9, and 10. In the second case, statements 1, 2, 5, 6, 9, and 10 will remain as the host code, whereas statements 1 and 2 will form a kernel and statements 7 and 8 will form a second kernel. We leave the detailed implementation of the kernels in both cases as exercises.

Vector Loop

Recall that OpenACC was designed to support multiple levels of parallelism found in a typical accelerator. The vector clause on a loop construct is often used to express the innermost vector or SIMD (single instruction, multiple data) mode loop in an accelerator region, as illustrated in Figure 15.13.

```
#pragma acc parallel num_gangs(1) num_workers(32)
{
    Statement 1;
    Statement 2;
    #pragma acc loop worker
    for (int i=0; i<n; i++) {
        Statement 3;
        Statement 4;
    }
    Statement 5;
    Statement 6;
    #pragma acc loop worker
    for (int i=0; i<m; i++) {
        Statement 7;
        Statement 8;
    }
    Statement 9;
    if (condition)
        Statement 10;
}
```

FIGURE 15.12

Porting is easier with OpenACC (Part 2).

```
#pragma acc parallel num_gangs(1024) num_workers(32)
vector_length(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            #pragma acc loop vector
            for (int k=0; k<1024; k++) {
                foo(i,j,k);
            }
        }
    }
}
```

FIGURE 15.13

Using the vector clause.

On a GPU, a possible implementation is to map a gang to a CUDA block, a worker to a CUDA warp, and a vector element to a thread within a warp. However, this is not mandated by the OpenACC specification and an implementation (compiler/runtime) may choose a different mapping based on the code pattern within an accelerator region for best performance.

Kernels Construct

Prescriptive Versus Descriptive

Like the parallel construct, the kernels construct also allows a programmer to specify which part of a program he or she wants to be executed on an accelerator. And a loop construct can be used inside a kernels construct. One major difference between the two is that a kernels region may be broken into a sequence of kernels, each of which will be executed on the accelerator, while the whole parallel region will become a kernel and be executed on the accelerator. Typically, each loop nest in a kernels construct may become a kernel, as illustrated in Figure 15.14.

In the figure, the kernels region may be broken into three kernels, one for each loop, and they will be executed on the accelerator in order. It is also possible that some implementations may decide not to generate a kernel for the k loop and therefore this kernels region will contain two kernels—one for the i loop and the j loop each and the k loop is executed on the host.

```
#pragma acc kernels
{
    #pragma acc loop num_gangs(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop num_gangs(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }
    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

FIGURE 15.14

A kernels region may be broken into a sequence of kernels.

A kernels region may contain multiple kernels and each may use a different number of gangs, a different number of workers, and different vector lengths. Therefore, there is no num_gangs, num_workers, or vector_length clause on the kernels construct. You can specify them on the enclosed loop construct if you want to, as illustrated in Figure 15.14.

Now let's take a look at another major difference between the parallel construct and the kernels construct. In Figure 15.14, how many times will the k loop be executed? Previously we've learned that the non-work-sharing code inside a parallel construct will be executed redundantly by the gang leads (see Figures 15.5 and 15.6). If the k loop in Figure 15.14 were inside a parallel construct, then the statement d[k] = c[k] is *executed* 2,048 times the number of gangs. This is different in a kernels construct, in which case it is just 2,048 times.

The parallel construct and the kernels construct were designed from two different perspectives. The kernels construct is more descriptive. It describes the intention of the programmer. The compiler is responsible for mapping and partitioning the program to the underlying hardware. Notice we use the word "may" when we explain the kernels code in Figure 15.14. It is possible that an OpenACC-compliant compiler decides not to generate any kernels at all for the kernels region in Figure 15.14. The loop constructs used for the i and j loops tells the compiler to generate such code that the loop iterations will be shared among the gang leaders only if the compiler decides to generate kernels for these loops. There are two common reasons why a compiler decides not to generate a kernel for a loop construct. One reason is safety. The compiler checks whether

parallelizing the loop will give the same execution result as the sequential version does. A series of analyses will be performed on the loop and the rest of the program. If the compiler finds it is not safe to parallelize the loop or cannot decide whether it is safe due to lack of information, then the compiler will not parallelize the loop and hence will not generate a kernel for the `loop` construct. The other reason is performance. The ultimate goal for using OpenACC directive is to get speedup. The compiler may decide not to parallelize and execute a loop on the accelerator if it finds doing so will only slow down the program. Since the compiler will mostly take care of the parallelization issues, the descriptive approach makes porting programs to OpenACC relatively easier. The downside is the quality of the generated accelerated code depends significantly on the capability of the compiler used. A high-quality compiler is expected to give feedbacks to the programmer on how it compiles `kernels` constructs and why it does not parallelize certain loops. With this information, the programmer can be sure whether his or her intension is achieved and may provide more hints to the compiler to achieve his or her goal. In the next section, we will show a few ways to help an OpenACC compiler.

The `parallel` construct is more prescriptive. The compiler does what the programmer instructs it to do. The programmer ultimately has more control of where to generate kernels and how to parallelize and schedule loops. Different OpenACC compilers should perform the similar transformations to a `parallel` construct. The downside is that there is no safety net. If a loop has data dependence between different iterations and is unsafe to be parallelized, then a programmer should not put such a serial loop inside a `loop` construct. This is the same philosophy taken by OpenMP, which is another successful directive-based approach for parallel programming. Programmers who are familiar with OpenMP should feel comfortable using `parallel` constructs.

Ways to Help an OpenACC Compiler
To parallelize a loop inside a kernels region, an OpenACC compiler generally needs to prove there is no cross-iteration data dependence in the loop. There is no data dependence in the `i` loop in Figure 15.15. All iterations can be executed in parallel and give the same result as when the iterations are executed sequentially. An OpenACC compiler should have no trouble deciding the `i` loop is parallelizable. In the `j` loop, each iteration uses the value of array element `a[]` defined in the previous iteration, therefore, the result will be different if the loops are executed in parallel.

```
void foo(int *x, int *y, int n, int m) {
   int a[2048], b[2048];

   #pragma acc kernels copy(x[0:2048], y[0:2048], a, b)
   {
      // No data dependence
      #pragma acc loop
      for (int i=0; i<2047; i++) {
          a[i] = b[i+1] + 1;
      }

      // Data dependence
      #pragma acc loop
      for (int j=0; j<2047; j++) {
          a[j] = a[j+1] + 1;
      }

      // No data dependence if x[] is not aliased with y[]
      #pragma acc loop
      for (int k=0; k<2047; k++) {
          x[k] = y[k+1] + 1;
      }

      // No data dependence if n>=m
      #pragma acc loop
      for (int l=0; l<m; l++) {
          x[l] = x[l+n] + 1;
      }

   }
   …
}
```

FIGURE 15.15

Data dependence.

An OpenACC compiler should have no trouble deciding the 'j' loop is not parallelizable.

For the k loop, there is no data dependence if x[] and y[] are not aliased. However, this cannot be decided by examining the function foo() alone. If an OpenACC compiler does not perform interprocedural analysis or the call site of function foo() is not available, then the compiler has to conservatively assume there is data dependence. If x[] and y[] are indeed never aliased, we can add the C restrict qualifier to the declaration of pointer argument x and y as illustrated in Figure 15.16. An OpenACC compiler should then be able to use this information to decide if the k loop is parallelizable.

```
void foo(int * restrict x, int * restrict y, int n, int m) {
```

FIGURE 15.16

Use `restrict` qualifier to specify no alias.

```
#pragma acc loop independent
for (int l=0; l<m; l++) {
    x[l] = x[l+n] + 1;
}
```

FIGURE 15.17

Use the 'independent' clause to declare a loop is parallelizable.

Now what to do with the *l* loop? This loop is parallelizable if the value of n is no less than m. Let's assume this is always true in this program. However, there is no C language construct to express such information. In this case, we can add an `independent` clause to the `loop` construct, as illustrated in Figure 15.17. The `independent` clause simply tells the compiler that the associated loop is parallelizable and no analysis is required. You can also add the `independent` clause to the 'i'. You could also add the clause to the 'j' loop construct, but that would not be correct.

Data Management
Data Clauses
So far you have seen `copy`, `copyin`, and `copyout` clauses used on `parallel` and `kernels` constructs. These are called *data clauses*. A data clause has a list of arguments separated by a comma. Each argument can be a variable name or a subarray specification. The OpenACC compiler and runtime will create a copy of the variable or subarray in the device memory. Reference to the variable or subarray within the `parallel` or `kernels` constructs will be made to the device copy.

The code snippet in Figure 15.18 is from the matrix multiplication example in Figure 15.4. Here, three pieces of memory are allocated on the device. Arrays M and N are the input data, so they are declared as `copyin`. The `copyin` from the host memory to the device memory happens right before the parallel region starts execution. Array P is the output data, so it is declared as `copyout`. The `copyout` from the device memory to the host

```
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Nw*Mw])
    copyout(P[0:Mh*Nw])
```

FIGURE 15.18

Data clauses used in Figure 15.4.

memory happens right after the parallel region ends. The `copy` clause can be used to declare data that needs to be both copied in and copied out.

Notice that the subarray specification is used for `M`, `N`, and `P` here. That's because `M`, `N`, and `P` are actually pointers and we need to specify the range of memory that needs to be copied. The value before and after the `:` specifies the starting array element and the number of array elements, respectively. So `M[0:Mh*Mw]` means `M[0]`, `M[1]`, `M[2]`, ..., and `M[Mh*Mw-1]`. A common programmer error is mistaking the second value as the last array element.

Some variables do not need to be copied in or copied out—their values are generated and consumed within a kernel. In such cases, the `create` clause can be used.

Another commonly used data clause is the `deviceptr` clause. This clause takes a list of pointers as its argument and declares that these are actually device pointers so that the data does not need to be allocated or moved between the host and the device for memory pointed by these pointers. When a program uses both OpenACC and CUDA kernels (or CUDA libraries, such as cuFFT, cuBLAS, etc.), the `deviceptr` clause becomes handy. Figure 15.19 shows an example of doing the matrix multiplication twice, first using a CUDA kernel and then using an OpenACC parallel region—both work on the same device memory allocated by `cudaMalloc()`.

Data Construct

In OpenACC, host memory and device memory are separated. Data transfer between the host and the accelerator can play a significant role in the overall performance of an OpenACC application. For example, when a computationally intense loop nest of an iterative solver, implemented using a `parallel` loop, transfers data back and forth between host and the accelerator at every iteration, then there may be a loss of performance. The OpenACC `data` construct allows one to exploit reuse by avoiding data transfers during multiple executions of parallel or kernels regions.

Figure 15.20 shows a simplified implementation of a 2D Jacobi relaxation. Each element in the array `field` is updated by taking the average of

```
__global__ void MatrixMulKernel(float *M, float *N, float *P, int n) {
    …
}

void MatrixMulAcc(float *M, float *N, float *P, int n) {
#pragma acc parallel loop deviceptr(M, N, P)
{
    …
}
}

void matrixMul(float *M, float *N, float *P, int n) {
    unsigned int size = n * n * sizeof(float);
    float *Md = NULL;
    float *Nd = NULL;
    float *Pd = NULL;

    cudaMalloc((void**) &Md, size);
    cudaMalloc((void**) &Nd, size);
    cudaMalloc((void**) &Pd, size);

    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(TILE_WIDTH,TILE_WIDTH);
    dim3 dimGrid(n/TILE_WIDTH, n/TILE_WIDTH);

    // Use CUDA Kernel
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, n);
    cudaThreadSynchronize();
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Use OpenACC
    MatrixMulACC(Md, Nd, Pd, n);
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

}
```

FIGURE 15.19

Use `deviceptr` to pass `cudaMalloc()` data to OpenACC parallel or kernels region.

each element with its eight neighbors. This is repeated 256 times. Another array `tmpfield` is used to make the relaxation parallel. In each pass, the values are read from one array and the average is computed and then it is written into the same position in the second array. Since the two arrays do not overlap, the updates are completely data parallel. Lines 6-24 implement one pass of the relaxation. Each pass is executed in an OpenACC

parallel region. We group the 256 passes into 128 pairs. Each pair contains two parallel regions—one updates `tmpfield` with `field`, and the other updates `field` with `tmpfield`. Recall that there is no synchronization between gangs. Therefore, we need two `parallel` constructs to make sure

```
1   #define N 1024
2   double field[N*N];
3   double tmpfield[N*N];
4   #define pos(i,j) (j)*N+(i)
5
6   void relaxation(double *oldValue, double *newValue) {
7       #pragma acc loop
8       for (int x=1; x<N-1; x++) {
9           #pragma acc loop
10          for (int y=1; y<N-1; y++) {
11              double sum = oldValue[pos(x,y)];
12              sum += oldValue[pos(x-1,y)];
13              sum += oldValue[pos(x+1,y)];
14              sum += oldValue[pos(x,y+1)];
15              sum += oldValue[pos(x,y-1)];
16              sum += oldValue[pos(x+1,y+1)];
17              sum += oldValue[pos(x-1,y+1)];
18              sum += oldValue[pos(x+1,y-1)];
19              sum += oldValue[pos(x-1,y-1)];
20              sum = sum / 9;
21              newValue[pos(x,y)] = sum;
22          }
23      }
24  }
25
26  void transform() {
27      ...
28      #pragma acc data copy(field) create(tmpfield)
29      {
30          for (int i=0; i<128; i++) {
31              #pragma acc parallel
32              relaxation(field, tmpfield);
33              #pragma acc parallel
34              relaxation(tmpfield, field);
35              if (check) {
36                  #pragma acc update host(field)
37                  display(field);
38              }
39          }
40      }
41      ...
42  }
```

FIGURE 15.20

Use of `data` and `update` constructs.

the writes to one array are completed before the array can be used as the source of updates in the next pass.

We want the data to stay on the device during all the 256 passes. This is achieved by using the data construct in line 28. The data region specified by the data construct is from line 30-38, including all called functions. The copy(field) clause says we need to create a device copy of array field, copy its data from the host to the device when the data region starts, and copy its data back to the host when the data region ends. And for the enclosed parallel constructs at lines 31 and 33, just use this copy of field. The create(tmpfield) clause says we need to create a device copy of array tmpfield for this data region, and use this copy for the enclosed parallel constructs at lines 31 and 33.

Now the data is on the device all the time during the passes. What if we want to check the intermediate result on the host occasionally? We can do it by using the update directive, as illustrated at line 36. This says the value of the host array 'fields' should be updated with that of the device copy at this point. Since the update is performed conditionally in the code, the data transfer won't happen if not required. The update directive can also be used to update the value of the device copy with that of the host copy.

Asynchronous Computation and Data Transfer

OpenACC provides support for asynchronous computation and data transfer. An async clause can be added to parallel, kernels, or update directive to enable asynchronous execution. If there is no async clause, the host process will wait until the parallel region, kernels region, or updates are complete before continuing. If there is an async clause, the host process will continue with the code following the directive when the parallel region, kernels region, or updates are processed asynchronously. An asynchronous event can be waited by using the wait directive or OpenACC runtime library routines.

In the Jacobi relaxation example in Figure 15.20, the update of the host copy of field (line 37) and the display of it on the host could happen in parallel with the compute of tmpfield (lines 31 and 32) on the device.

In Figure 15.21, to enable the asynchronous execution, we move the update and display in between the two parallel regions, add an async clause to the parallel directive at line 31, and add a wait directive before the second parallel directive at line 33.

```
26 void transform() {
27     ...
28     #pragma acc data copy(field) create(tmpfield)
29     {
30         for (int i=0; i<128; i++) {
31             #pragma acc parallel async
32             relaxation(field, tmpfield);
35             if (check) {
36                 #pragma acc update host(field)
37                 display(field);
38             }
33-1           #pragma acc wait
33-2           #pragma acc parallel
34             relaxation(tmpfield, field);
39         }
40     }
41     ...
42 }
```

FIGURE 15.21

async and wait.

We can replace the wait directive with a call to the OpenACC acc_a-sync_wait_all() routine and achieve the same effect. OpenACC provides a richer set of routines to support the asynchronous wait functionality, including the capability to test whether an asynchronous activity has completed rather than just waiting for its completion.

15.5 FUTURE DIRECTIONS OF OPENACC

We believe using OpenACC will become a promising and effective approach to port existing applications to accelerators and even write accelerated applications from scratch. The following are a few directions we see the OpenACC programming model going in.

• *Be more general.* The current OpenACC model and implementations have quite a few limitations, such as function calls must be able to be in-lined, no support for dynamic memory allocation on the device, etc. It is due to the fact that most OpenACC features were originally designed at the CUDA 3.0 timeframe. Since then, more software and hardware features have been developed on the CUDA platform. For example, in CUDA 4.0, GPUs can be shared across multiple threads, and C++ new/delete and virtual functions support are added. In

CUDA 5.0, separate compilation and device code linking is available. OpenACC will certainly take advantage of these new technologies to make the program model more general.

- *Integrated with OpenMP*. OpenMP and OpenACC both use the directive approach to parallel programming. OpenMP has traditionally been focusing on shared memory systems. The OpenMP ARB has formed an accelerator working group to extend OpenMP support on accelerators. All OpenACC founding members are members of this working group. These members intend to merge the two specifications to create a common one.

Last but not least, we encourage you to follow the latest development of OpenACC by visiting the official OpenACC web site at *openacc.org*. Besides the latest update to the specification itself, the web site provides a rich resource for documents, FAQs, tutorials, code samples, vendor news, and discussion forums.

15.6 EXERCISES

15.1. In the following parallel region, how many instances of statement 1 will be executed in total?

```
#pragma acc parallel gang(1024) worker(32)
{
#pragma acc loop worker
for (int i = 0; i < 2048; i++) {
statement 1;
}
}
```

15.2. What are the two major differences between the `parallel` construct and the `kernels` construct?

15.3. Implement the matrix multiplication using the `kernels` construct.

15.4. Reimplement Jacobi relaxation using the `kernels` constructs. Use a different number of gangs, works, and vector lengths to see how they affect performance.

This page intentionally left blank

Thrust: A Productivity-Oriented Library for CUDA

16

With special contributions by Nathan Bell, Jared Hoberock and Chris Rodrigues

CHAPTER OUTLINE

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Thrust provides a set of type-generic parallel algorithms that can be used with user-defined data types. These parallel algorithms can significantly reduce the effort of developing parallel applications. Applications written with Thrust are concise, readable, and efficient.

16.1 BACKGROUND

C++ provides a way for programmers to define generics. In situations when a programming problem has the same solution for many different

data types, the solution can be written once and for all using *generics*. For example, the two C++ functions shown in the following code sum a `float` array and an `int` array. They are defined without using type generics. The only difference between the first and second function is that `float` is changed to `int`.

```
float sum(int n, float *p) {          int sum(int n, int *p) {
 float a = 0;                          int a = 0;
 for (int i = 0; i < n; i++) a          for (int i = 0; i < n; i++) a
 + = p[i];                              + = p[i];
 return a;                             return a;
}                                     }
```

Instead of writing a different version of `sum` for each data type, the following generic `sum` function can be used with any data type. The idea is that the programmer prepares a template of the `sum` function that can be instantiated on different types of array. The `template` keyword indicates the beginning of a type-generic definition. From this point on, we will use type-generic and generic interchangeably.

```
template < typename T >
T sum(int n, T *p) {
 T a = 0;
 for (int i = 0; i < n; i++) a + = p[i];
 return a;
}
```

The code uses `T` as a placeholder where the actual type needs to be. Replacing `T` by `float` in the generic code yields one of the two definitions of `sum`, while replacing `T` by `int` yields the other. `T` could also be replaced by other types, including user-defined types. A C++ compiler will make the appropriate replacement each time the `sum` function is used. Consequently, `sum` behaves much like the preceding overloaded C++ function, and it can be used as if it were an overloaded function. The central concept of generic programming is the use of *type parameters*, like `T` in this example, that can be replaced by arbitrary types.

Thrust is a library of generic functions. By providing generic functions for each type of computation to be supported, Thrust does not need to have multiple versions of each function replicated for each eligible data type.

In fact, not all data types can be used with a generic function. Because sum uses addition and initializes a to 0, it requires the type T to behave (broadly speaking) like a number. Replacing T by the numeric types int or float produces a valid function definition, but replacing T by void or FILE* does not. Such requirements are called *concepts*, and when a type satisfies a requirement it is said to *model* a concept. In sum, whatever replaces T must model the "number" concept. That is, sum will compute a sum provided that it's given a pointer to some type T that acts like a numeric type. Otherwise, it may produce an error or return a meaningless result. Generic libraries like Thrust rely on concepts as part of their interface.

C++ classes can be generic as well. The idea is similar to generic functions, with the extra feature that a class's fields can depend on type parameters. Generics are commonly used to define reusable *container classes*, such as those in the STL [HB2011]. Container classes are implementations of data structures, such as queues, linked lists, and hash tables, that can be used to hold arbitrary data types. For instance, a very simple generic array container class could be defined as follows:

```
template < typename T >
class Array {
 T contents[10];
public:
 T read(int i) {return contents[i];}
 void write(int i, T x) {contents[i] = x;}
};
```

Containers for different data types can be created using this generic class. Their types are written as the generic class name followed by a type in angle brackets: Array < int > for an array of int, Array < float * > for an array of float*, and so forth. The type given in angle brackets replaces the type parameter in the class definition.

While this is not a complete description of how generics work, it conveys the essential ideas for understanding the use of generics in this chapter.

We will introduce one more background concept: iterators. In the same way that pointers are used to access arrays, iterators are used to access container classes. The term *iterator* refers to both a C++ concept and a value of which the type is a model of this concept. An iterator represents a position within a container: it can be used to access the element at that position, used to go to a neighboring position, or compared to other positions.

Pointers are a model of the iterator concept, and they can be used to loop over an array as shown in the following:

```
int a[50];
for (int *i = a; i < a + 50; i++) *i = 1;
```

Iterators can be used to loop over an STL vector in a very similar way:

```
vector < int > a(50);
for (vector < int>::iterator i = a.begin(); i < a.end(); i++) *i
  = 1;
```

The member functions `begin()` and `end()` return iterators referencing the beginning and just past the end of the vector. The ++, <, and * operators are overloaded to act like their pointer counterparts. Because many container classes provide an iterator interface, generic C++ code using iterators can be reused to process different kinds of containers.

16.2 MOTIVATION

CUDA C allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C is an important feature: it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks that (1) merit significant optimization, and (2) profit from low-level control of the mapping onto hardware. For this class of computational tasks CUDA C is an excellent solution.

Thrust [HB2011] solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture, or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of *high-level* algorithms and completely *delegate* the decision of how to implement the computation to the library. This abstract interface allows programmers to describe *what to compute* without placing any additional restrictions on how to carry out the computation. By capturing the programmer's intent at a high level, Thrust has the discretion to make informed decisions on behalf of the programmer and select the most efficient implementation.

The value of high-level libraries is broadly recognized in high-performance computing. For example, the widely used BLAS standard provides an abstract interface to common linear algebra operations. First

conceived more than three decades ago, BLAS remains relevant today in large part because it allows valuable, platform-specific optimizations to be introduced behind a uniform interface.

Whereas BLAS is focused on numerical linear algebra, Thrust provides an abstract interface to fundamental parallel algorithms such as scan, sort, and reduction. Thrust leverages the power of C++ templates to make these algorithms generic, enabling them to be used with arbitrary user-defined types and operators. Thrust establishes a durable interface for parallel computing with an eye toward generality, programmer productivity, and real-world performance.

16.3 BASIC THRUST FEATURES

Before going into greater detail, let us consider the program in Figure 16.1, which illustrates the salient features of Thrust.

```cpp
#include <thrust/host vector.h>
#include <thrust/device vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    //  generate 16M random numbers on the host

    thrust::host vector<int> h vec(1 << 24);
    thrust::generate(h vec.begin(), h vec.end(), rand);

        //  transfer  data  to  the  device
    thrust::device vector<int>  d_vec  =  h vec;

    //  sort data on the device

    thrust::sort(d vec.begin(), d vec.end());

        //  transfer  data  back  to  host
    thrust::copy(d vec.begin(),  d_vec.end(),  h_vec.begin());

    return  0;
}
```

FIGURE 16.1

A complete Thrust program that sorts data on a GPU.

Thrust provides two vector *containers*: `host_vector` and `device_vector`. As the names suggest, `host_vector` is stored in the host memory while `device_vector` lives in the device memory on a GPU. Like the vector container in the C++ STL, `host_vector` and `device_vector` are generic containers (i.e., they are able to store any data type) that can be resized dynamically. As the example shows, containers automate the allocation and de-allocation of memory and simplify the process of exchanging data between the host and the device.

The program acts on the vector containers using the `generate`, `sort`, and `copy` algorithms. Here, we adopt the STL convention of specifying *ranges* using pairs of *iterators*. In this example, the iterators `h_vec.begin ()` and `h_vec.end()` point to the first element and the element one past the end of the array, respectively. Together the pair defines a *range* of integers of size `h_vec.end()` − `h_vec.begin()`.

Note that even though the computation implied by the call to the `sort` algorithm suggests one or more CUDA kernel launches, the programmer has not specified a launch configuration. Thrust's interface *abstracts* these details. The choice of performance-sensitive variables such as grid and block size of the library, the details of memory management, and even the choice of sorting algorithm are left to the discretion of the implementer.

Iterators and Memory Space

Although vector iterators are similar to pointers, they carry additional information. Notice that we did not have to instruct the `sort` algorithm that it was operating on the elements of a `device_vector` or hint that the `copy` was from the device memory to the host memory. In Thrust the memory spaces of each range are automatically *inferred* from the iterator arguments and used to dispatch the appropriate implementation.

In addition to memory space, Thrust's iterators implicitly encode a wealth of information that can guide the dispatch process. For instance, our `sort` example in Figure 16.1 operates on `int`, a primitive data type with a fundamental comparison operation. In this case, Thrust dispatches a highly tuned radix sort algorithm [MG2010] that is considerably faster than alternative comparison-based sorting algorithms such as merge sort [SHG2009]. It is important to realize that this dispatch process incurs no performance or storage overhead: metadata encoded by iterators exists only at compile time, and dispatch strategies based on it are selected

statically. In general, Thrust's static dispatch strategies may capitalize on any information that is derivable from the type of an iterator.

Interoperability

Thrust is implemented entirely within CUDA C/C++ and maintains interoperability with the rest of the CUDA ecosystem. Interoperability is an important feature because no single language or library is the best tool for every problem. For example, although Thrust algorithms use CUDA features like shared memory internally, there is no mechanism for users to exploit shared memory directly through Thrust. Therefore, it is sometimes necessary for applications to access CUDA C directly to implement a certain class of specialized algorithms, as illustrated in the software stack of Figure 16.2. Interoperability between Thrust and CUDA C allows the programmer to replace a Thrust kernel with a CUDA kernel and vice versa by making a small number of changes to the surrounding code.

Interfacing Thrust to CUDA C is straightforward and analogous to the use of the C++ STL with standard C code. Data that resides in a Thrust container can be accessed by external libraries by extracting a "raw" pointer from the vector. The code sample in Figure 16.3 illustrates the use of a raw pointer cast to obtain an int point to the contents of a device vector.

In Figure 16.3(a), the function raw_pointer_cast() takes the address of element 0 of a device vector d_vec and returns a raw C pointer raw_ptr. This pointer can then be used to call CUDA C API functions
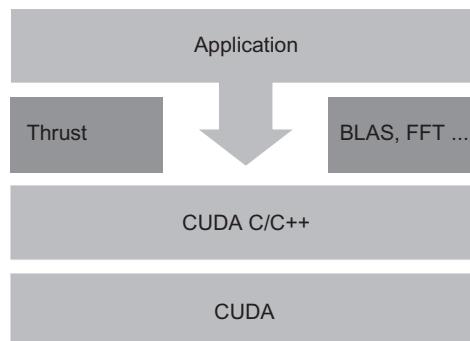
FIGURE 16.2

Thrust is an abstraction layer on top of CUDA C/C++.

```
size t  N  =  1024;
```

```
size t  N  =  1024;                      // raw  pointer  to  device  memory
                                         int raw ptr;
// allocate Thrust container             cudaMalloc(&raw ptr, N sizeof(int));
device vector<int>  d vec(N);
                                         // wrap raw pointer with a device ptr
// extract  raw  pointer  from           device ptr<int> dev ptr = device
container                                pointer cast(raw ptr);

int raw ptr = raw pointer cast(&d
vec[0]);
                                         // use device ptr in Thrust  algorithms
// use raw ptr in non Thrustfunctions     sort(dev ptr, dev ptr + N);

cudaMemset(raw ptr,  0,  N
sizeof(int));
                                         // access device memory through device
// pass raw ptr to a kernel              ptr
my kernel<<<N / 128, 128>>>(N, raw       dev ptr[0] = 1;
ptr);
                                         // free  memory
// memory is automatically freed         cudaFree(raw ptr);
```

(a) Interfacing Thrust to CUDA   (b) Interfacing CUDA to Thrust

FIGURE 16.3

Thrust interoperates smoothly with CUDA C/C++: (a) interfacing Thrust to
CUDA, and (b) interfacing CUDA to Thrust.

(cudaMemset() in this example) or passed as a parameter to a CUDA C
kernel (my_kernel in this example).

Applying Thrust algorithms to raw C pointers is also straightforward.
Once the raw pointer has been wrapped by a device_ptr it can be used
like an ordinary Thrust iterator. In Figure 16.3(b), the C pointer raw_ptr
points to a piece of device memory allocated by cudaMalloc(). It can be
converted or wrapped into a device pointer to a device vector by the devi-
ce_pointer_cast() function. The wrapped pointer provides the memory
space information Thrust needs to invoke the appropriate algorithm imple-
mentation and also allows a convenient mechanism for accessing device
memory from the host. In this case, the information indicates that dev_ptr
points to a vector in the device memory and the elements are of type int.

Thrust's native CUDA C interoperability ensures that Thrust always
complements CUDA C and that a Thrust plus CUDA C combination is
never worse than either Thrust or CUDA C alone. Indeed, while it may be
possible to write whole parallel applications entirely with Thrust functions,
it is often valuable to implement domain-specific functionality directly in
CUDA C. The level of abstraction targeted by native CUDA C affords

programmers fine-grained control over the precise mapping of computational resources to a particular problem. Programming at this level provides developers the flexibility to implement exotic or otherwise specialized algorithms. Interoperability also facilitates an iterative development strategy: (1) quickly prototype a parallel application entirely in Thrust, (2) identify the application's hot spots, and (3) write more specialized algorithms in CUDA C and optimize as necessary.

16.4 GENERIC PROGRAMMING

Thrust presents a style of programming emphasizing code reusability and composability. Indeed, the vast majority of Thrust's functionality is derived from four fundamental parallel algorithms: `for_each`, `reduce`, `scan`, and `sort`. For example, the `transform` algorithm is a derivative of `for_each` while the inner product is implemented with `reduce`.

Thrust algorithms are generic in both the type of the data to be processed and the operations to be applied to the data. For instance, the `reduce` algorithm may be employed to compute the sum of a range of integers (a `plus` reduction applied to `int` data) or the maximum of a range of floating-point values (a `max` reduction applied to `float` data). This generality is implemented via C++ templates, which allows user-defined types and functions to be used in addition to built-in types such as `int` or `float`, or Thrust operators such as `plus`.

Generic algorithms are extremely valuable because it is impractical to anticipate precisely which particular types and operators a user will require. Indeed, while the computational structure of an algorithm is fixed, the number of *instantiations* of the algorithm is limitless. However, it is also worth mentioning that while Thrust's interface is general, the abstraction affords implementors the opportunity to specialize for specific types and operations known to be important use cases. These opportunities may be exploited statically.

In Thrust, user-defined operations take the form of C++ function objects, or *functors*. Functors allow the programmer to adapt a generic algorithm to perform a specific user-defined operation. For example, the code samples in Figure 16.4 implement SAXPY, the well-known BLAS operation, using CUDA C and Thrust, respectively. The CUDA C code should be very familiar and is provided for comparison.

The Trust code has two parts. In the first part, the code sets up a SAXPY functor that receives an input floating value `a` and maintains it as

```
global
void saxpy kernel(int n, float a, "float*" x, "float*" y)
{
const int i = blockDim.x * blockIdx.x + threadIdx.x;

if (i < n) y[i] = a * x[i] + y[i];
}

void saxpy(int n, float a, "float*" x, "float*" y)
{
// set launch configuration parameters int block size = 256;
int grid size = (n + block_size - 1) / block size;

// launch saxpy kernel

saxpy kernel<<< grid_size, block_size>>>(n, a, x, y);
}
```

(a) CUDA C

```
struct saxpy_functor
{
  const float a;

  saxpy_functor(float  _a)  :  a(_a)  {}

  __host__  __device__
  float operator() (float x, float y)
  {
   return a * x + y;
  }
}

void saxpy(float a, device_vector <float> &x, device_vector<float>
    &y)
{
   // setup functor
  saxpy_functor  func(a);

   //    call transform
  transform(x.begin(), x.end(), y.begin(), y.end(), func);
}
```

(b) Thrust

FIGURE 16.4.

SAXPY implementations in (a) CUDA C and (b) Thrust.

a state. It can then be called as an operator that performs $a*x + y$ on two input values x and y. Finally, the generic transform algorithm is called with the user-defined saxpy_functor func. The iterators provided to the

transform algorithm will apply func to each pair of the x and y elements and produce the SAXPY results. Note that the operator defined in the sax-py_functor declaration can be overloaded so that different types of a, x, y can be passed into the transform algorithm and the correct operator will be invoked to generate the expected output values for each type of inputs. This makes it possible to create a generic SAXPY function.

C++ FUNCTION OBJECTS

A C library developer can set up a generic function by allowing the user to provide a callback function. For example, a sort function can allow the user to pass a function pointer as a parameter to perform the comparison operation for determining the order between two input values. This allows the user to pass any types of input as long as he or she can define a comparison function between two input values.

It is sometimes desirable for a callback function to maintain a state. The C++ function object, or functor, provides a convenient way to do so. A functor is really a function defined on an object that holds a state. The function that is passed as the callback function is just a member function defined in the class declaration of the object. In the case of the saxpy_ functor class, a is the class data and operator is the member function defined on the data. When an instance of saxpy_functor func is passed to a generic algorithm function such as transform(), the operator will be called to operate on each pair of x and y elements.

16.5 BENEFITS OF ABSTRACTION

In this section we'll describe the benefits of Thrust's abstraction layer with respect to programmer productivity, robustness, and real-world performance.

16.6 PROGRAMMER PRODUCTIVITY

Thrust's high-level algorithms enhance programmer productivity by automating the mapping of computational tasks onto the GPU. Recall the two implementations of SAXPY shown in Figure 16.4. In the CUDA C implementation of SAXPY the programmer has described a specific decomposition of the parallel vector operation into a grid of blocks with 256 threads per block. In contrast, the Thrust implementation does not prescribe a launch configuration. Instead, the only specifications are the input and output ranges and a functor to apply to them. Otherwise, the two codes are roughly the same in terms of length and code complexity.

Delegating the launch configuration to Thrust has a subtle yet profound implication: the launch parameters can be automatically chosen based on a model of machine performance. Currently, Thrust targets *maximal occupancy* and will compare the resource usage of the kernel (e.g., number of registers, amount of shared memory) with the resources of the target GPU to determine a launch configuration with the highest occupancy. While the maximal occupancy heuristic is not necessarily optimal, it is straightforward to compute and effective in practice. Furthermore, there is nothing to preclude the use of more sophisticated performance models. For instance, a runtime tuning system that examined hardware performance counters could be introduced behind this abstraction without altering client code.

Thrust also boosts programmer productivity by providing a rich set of algorithms for common patterns. For instance, the map-reduce pattern is conveniently implemented with Thrust's `sort by key` and `reduce by key` algorithms, which implement key-value sorting and reduction, respectively.

Robustness

Thrust's abstraction layer also enhances the robustness of CUDA applications. In the previous section we noted that by delegating the launch configuration details to Thrust we could automatically obtain maximum occupancy during execution. In addition to maximizing occupancy, the abstraction layer also ensures that algorithms "just work," even in uncommon or pathological use cases. For instance, Thrust automatically handles limits on grid dimensions (no more than 64 K in current devices), works around limitations on the size of global function arguments, and accommodates large user-defined types in most algorithms. To the degree possible, Thrust circumvents such factors and ensures correct program execution across the full spectrum of CUDA-capable devices.

Real-World Performance

In addition to enhancing programmer productivity and improving robustness, the high-level abstractions provided by Thrust improve performance in real-world use cases. In this section we examine two instances where the discretion afforded by Thrust's high-level interface is exploited for meaningful performance gains.

To begin, consider the operation of filling an array with a particular value. In Thrust, this is implemented with the `fill` algorithm. Unfortunately, a straightforward implementation of this seemingly simple operation is subject to severe performance hazards. Recall that processors based on the G80

architecture (i.e., compute capability 1.0 and 1.1) impose strict conditions on which memory access patterns may benefit from memory coalescing [NVIDIA2010]. In particular, memory accesses of subword granularity (i.e., less than 4 bytes) are not coalesced by these processors. This artifact is detrimental to performance when initializing arrays of char or short types.
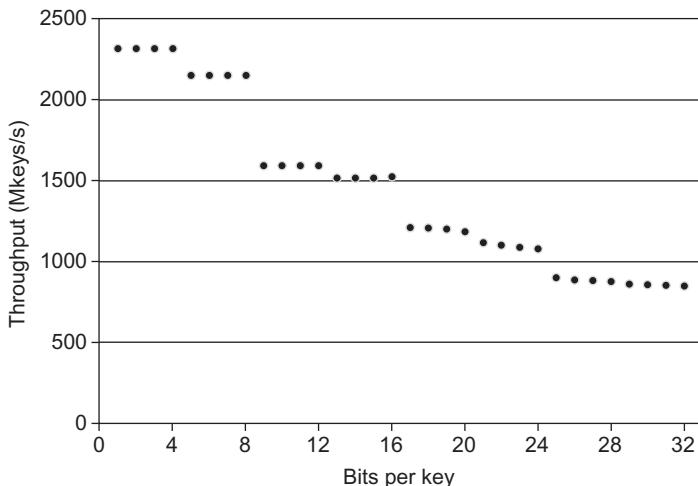
Fortunately, the iterators passed to fill implicitly encode all the information necessary to intercept this case and substitute an optimized implementation. Specifically, when fill is dispatched for smaller types, Thrust selects a "wide" version of the algorithm that issues word-sized accesses per thread. While this optimization is straightforward to implement, users are unlikely to invest the effort of making this optimization themselves. Nevertheless, the benefit, shown in Table 16.1, is worthwhile, particularly on earlier architectures. Note that with the relaxed coalescing rules on the more recent processors, the benefit of the optimization has somewhat decreased but is still significant.

Like fill, Thrust's sorting functionality exploits the discretion afforded by the abstract sort and stable sort functions. As long as the algorithm achieves the promised result, we are free to utilize sophisticated static (compile-time) and dynamic (runtime) optimizations to implement the sorting operation in the most efficient manner.

As mentioned in Section 16.3, Thrust statically selects a highly optimized radix sort algorithm [MG2010] for sorting primitive types (e.g., char, int, float, and double) with the standard less and greater comparison operators. For all other types (e.g., user-defined data types)

Table 16.1 Memory Bandwidth of Two fill Kernels				
GPU	**Data Type**	**Naïve** fill	**Thrust** fill	**Speedup**
GeForce GTS8800	char	1.2 GB/s	41.2 GB/s	34.15×
	short	2.4 GB/s	41.2 GB/s	17.35×
	int	41.2 GB/s	41.2 GB/s	1×
	long	40.7 GB/s	40.7 GB/s	1×
GeForce GTX280	char	33.9 GB/s	75 GB/s	2.21×
	short	51.6 GB/s	75 GB/s	1.45×
	int	75 GB/s	75 GB/s	1×
	long	69.2 GB/s	69.2 GB/s	1×
GeForce GTX480	char	74.1 GB/s	156.9 GB/s	2.12×
	short	136.6 GB/s	156.9 GB/s	1.15×
	int	146.1 GB/s	156.9 GB/s	1.07×
	long	156.9 GB/s	156.9 GB/s	1×

FIGURE 16.5

Sorting integers on the GeForce GTX480: Thrust's dynamic sorting optimizations improve performance by a considerable margin in common use cases where keys are less than 32 bits.

and comparison operators, Thrust uses a general merge sort algorithm. Because sorting primitives with radix sort is considerably faster than merge sort, this static optimization has significant value.

Thrust also applies dynamic optimizations to improve sorting performance. Since the cost of radix sort is proportional to the number of significant key bits, we can exploit unused key bits to reduce the cost of sorting. For instance, when all integer keys are in the range [0, 16), only 4 bits must be sorted, and we observe a $2.71 \times$ speedup versus a full 32-bit sort. The relationship between key bits and radix sort performance is plotted in Figure 16.5.

16.7 BEST PRACTICES

In this section we highlight three high-level optimization techniques that programmers may employ to yield significant performance speedups when using Thrust.

Fusion

The balance of computational resources on modern GPUs implies that algorithms are often *bandwidth limited*. Specifically, computations with low CGMA (Computation to Global Memory Access) ratio, the ratio of calculations per memory access, are constrained by the available memory bandwidth and do not fully utilize the computational resources of the GPU. One technique for increasing the computational intensity of an algorithm is to *fuse* multiple pipeline stages together into a single operation. In this section we demonstrate how Thrust enables developers to exploit opportunities for kernel fusion and better utilize GPU memory bandwidth.

The simplest form of kernel fusion is scalar function composition. For example, suppose we have the functions $f(x) \rightarrow y$ and $g(y) \rightarrow z$ and would like to compute $g(f(x)) \rightarrow z$ for a range of scalar values. The most straightforward approach is to read x from memory, compute the value $y = f(x)$, and then write y to memory, and then do the same to compute $z = g(y)$. In Thrust this approach would be implemented with two separate calls to the transform algorithm, one for f and one for g. While this

```
struct square
{

    __host__ __device__
    float operator() (float x) const
    {
    return x *x;
    }
}

float snrm2_slow(const thrust::device vector<float>& x)
{
    //  without  fusion
    device vector<float> temp(x.size()); transform(x.begin(),
    x.end(), temp.begin(), square());

    return sqrt(  reduce(temp.begin(),  temp.end())  );
}
float snrm2_fast(const thrust::device vector<float>& x)
{
    //  with  fusion
    return sqrt( transform_reduce(x.begin(),x.end(),square(),0.0f,
    plus<float>());
}
```

FIGURE 16.6

snrm2 has low arithmetic intensity and therefore benefits greatly from fusion.

approach is straightforward to understand and implement, it needlessly wastes memory bandwidth, which is a scarce resource.

A better approach is to fuse the functions into a single operation $g(f(x))$ and halve the number of memory transactions. Unless f and g are computationally expensive operations, the fused implementation will run approximately twice as fast as the first approach. In general, scalar function composition is a profitable optimization and should be applied liberally.

Thrust enables developers to exploit other, less obvious opportunities for fusion. For example, consider the two Thrust implementations of the BLAS function `snrm2` shown in Figure 16.6, which computes the Euclidean norm of a `float` vector.

Note that `snrm2` has low arithmetic intensity: each element of the vector participates in only two floating-point operations—one multiply (to square the value) and one addition (to sum values together). Therefore, an implementation of `snrm2` using the transform reduce algorithm, which fuses the square transformation with a plus reduction, should be considerably faster. Indeed, this is true and `snrm2_fast` is fully 3.8 times faster than `snr2_slow` for a 16 M element vector on a Tesla C1060.

While the previous examples represent some of the more common opportunities for fusion, we have only scratched the surface. As we have seen, fusing a transformation with other algorithms is a worthwhile optimization. However, Thrust would become unwieldy if all algorithms came with a `transform` variant. For this reason Thrust provides `transform iterator`, which allows transformations to be fused with any algorithm. Indeed, `transform reduce` is simply a convenience wrapper for the appropriate combination of transform iterator and reduce. Similarly, Thrust provides `permutation iterator`, which enables gather and scatter operations to be fused with other algorithms.

Structure of Arrays

In the previous section we examined how fusion minimizes the number of off-chip memory transactions and conserves bandwidth. Another way to improve memory efficiency is to ensure that all memory accesses benefit from *coalescing*, since coalesced memory access patterns are considerably faster than noncoalesced transactions.

Perhaps the most common violation of the memory coalescing rules arises when using a so-called array of structures (AoS) data layout. Generally speaking, access to the elements of an array filled with C `struct` or C++ `class` variables will be uncoalesced. Only explicitly

```
struct float3                    struct float3_soa
{
                                 {

  float  x;                        float    x[100];
  float  y;                        float    y[100];
  float  z;                        float    z[100];
}                                }
float3 aos[100];                 float3_soa soa;
     ...                              ...
aos[0].x = 1.0f;                 soa.x[0] = 1.0f;

(a) Array of Structures          (b) Structure of Arrays
```

FIGURE 16.7

Data layouts for 3D float vectors: (a) AoS and (b) SoA.

aligned structures such as the uint2 or float4 vector types satisfy the memory coalescing rules.

An alternative to the AoS layout is the structure of arrays (SoA) approach, where the components of each struct are stored in separate arrays. Figure 16.7 illustrates the AoS and SoA methods of representing a range of 3D float vectors. The advantage of the SoA method is that regular access to the *x*, *y*, and *z* components of a given vector is coalesceable (because float satisfies the coalescing rules), while regular access to the float3 structures in the AoS approach is not.

The problem with SoA is that there is nothing to logically encapsulate the members of each element into a single entity. Whereas we could immediately apply Thrust algorithms to AoS containers like device vector < float3 >, we have no direct means of doing the same with three separate device_vector < float > containers. Fortunately, Thrust provides zip iterator, which provides encapsulation of SoA ranges.

The zip iterator [BIL] takes a number of iterators and *zips* them together into a virtual range of tuples. For instance, binding three device_ vector < float > iterators together yields a range of type tuple < float, float, float >, which is analogous to the float3 structure.

Consider the code sample in Figure 16.8 that uses zip iterator to construct a range of 3D float vectors stored in SoA format. Each vector is transformed by a rotation matrix in the rotate tuple functor before being written out again. Note that zip iterator is used for both input and output ranges, transparently packing the underlying scalar ranges into tuples and then unpacking the tuples into the scalar ranges. On a Tesla C1060,

```
struct rotate tuple {
   __host__ __device__
   tuple<float,float,float> operator()(tuple<float,float,float>& t) {
      float x = get<0>(t);
      float y = get<1>(t);
      float z = get<2>(t);
      float rx = 0.36f * x + 0.48f * y + 0.80f * z;
      float ry = 0.80f * x + 0.60f * y + 0.00f * z;
      float rz = 0.48f * x + 0.64f * y + 0.60f * z;
       return make_tuple(rx, ry, rz);
   }
};
device vector<float> x(N), y(N), z(N);
transform(make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
    make_zip_iterator(make_tuple(x.end(), y.end(), z.end())),
    make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
    rotate tuple());
```

FIGURE 16.8

The make_zip_iterator facilitates processing of data in structure of arrays format.

SoA implementation is $2.85 \times$ faster than the analogous AoS implementation (not shown).

Implicit Ranges

In the previous sections we considered ways to efficiently transform ranges of values and ways to construct ad hoc tuples of values from separate ranges. In either case, there was some underlying data stored *explicitly* in memory. In this section we illustrate the use of *implicit* ranges, that is, ranges of which the values are defined programmatically and not stored anywhere in memory.

For instance, consider the problem of finding the index of the element with the smallest value in a given range. We could implement a special reduction kernel for this algorithm, which we'll call min index, but that would be time consuming and unnecessary. A better approach is to implement min_index in terms of existing functionality, such as a specialized reduction over (value, index) tuples, to achieve the desired result. Specifically, we can zip the range of values v[0], v[1], v[2], ::: together with a range of integer indices 0, 1, 2, ::: to form a range of tuples (v[0], 0), (v[1], 1), (v[2],2), ::: and then implement min index with

```
struct smaller_tuple {
   tuple<float,int> operator()(tuple<float,int> a,tuple<float,int> b) {
      // return the tuple with the smaller float value
      if (get<0>(a) < get<0>(b)) return a;
      else  return b;
   }
};

int min_index(device vector<float>& values) {
  // [begin,end) form the implicit sequence [0,1,2, ... value.size())
   counting iterator<int> begin(0);
   counting iterator<int> end(values.size());

  // initial value of the reduction
   tuple<float,int> init(values[0], 0);

  // compute the smallest tuple
  tuple<float,int> smallest =
    reduce(make_zip_iterator(make_tuple(values.begin(), begin)),
    make_zip_iterator(make_tuple(values.end(), end)),
    init, smaller_tuple());
   // return the index
   return get<1>(smallest);
}
```

FIGURE 16.9

Implicit ranges improve performance by conserving memory bandwidth.

the standard `reduce` algorithm. Unfortunately, this scheme will be much slower than a customized reduction kernel, since the index range must be created and stored explicitly in memory.

To resolve this issue Thrust provides `counting_iterator` [BIL], which acts just like the explicit range of values we need to implement `min_index`, but does not carry any overhead. Specifically, when `counting_iterator` is dereferenced it generates the appropriate value on-the-fly and yields that value to the caller. An efficient implementation of `min_index` using `counting iterator` is shown in Figure 16.9.

16.8 EXERCISES

1. Here `counting iterator` has allowed us to efficiently implement a special-purpose reduction algorithm without the need to write a new, special-purpose kernel. In addition to `counting iterator`, Thrust provides `constant iterator`, which defines an implicit range of

constant value. Note that these implicitly defined iterators can be combined with the other iterators to create more complex implicit ranges. For instance, `counting iterator` can be used in combination with `transform iterator` to produce a range of indices with nonunit stride.

Read Figure 16.9 and explain the operation of the algorithm using as small example. In practice, there is no need to implement `min index` since Thrust's `min element` algorithm provides the equivalent functionality. Nevertheless the `min index` example is instructive of best practices. Indeed, Thrust algorithms such as `min element`, `max element`, and `find if` apply the exact same strategy internally.

References

Boost Iterator Library. Available at: <www.boost.org/doc/libs/release/libs/iterator/>.

Hoberock, J., & Bell, N. (2011). Thrust: A Parallel Template Library, [version 1.4.0]

Merrill D., & Grimshaw, A., Revisiting sorting for GPGPU stream architectures. Technical Report CS2010-03. *University of virginia, department of computer science,* Charlottesville. 2010.

NVIDIA Corporation. *CUDA C best practices guide v3.2*. Santa Clara, CA: NVIDIA Corporation. 2010 (Section 3.2.1).

Satish, N., Harris, M. & Garland, M. Designing efficient orting algorithms for many-core GPUs. *Proceedings twenty third IEEE international parallel and distributed processing symposium, IEEE computer society*. Washington, DC: 2009.

CUDA FORTRAN

17

With special contributions from Greg Ruetsch and Massimiliano Fatica

CHAPTER OUTLINE

This chapter gives an introduction to CUDA FORTRAN, the FORTRAN interface to the CUDA architecture. CUDA FORTRAN was developed in 2009 as a joint effort between the Portland Group (PGI) and NVIDIA. CUDA FORTRAN shares much in common with CUDA C, as it is based on the runtime API, however, there are some differences in how the CUDA concepts are expressed using FORTRAN 90 constructs. The first section of this chapter discusses some of the basic differences between CUDA FORTRAN and CUDA C at a high level, and subsequent sections use various examples to illustrate CUDA FORTRAN programming.

17.1 **CUDA FORTRAN AND CUDA C DIFFERENCES**

CUDA FORTRAN and CUDA C have much in common, as CUDA FORTRAN is based on the CUDA C runtime API. Just as CUDA C is C with a few language extensions, CUDA FORTRAN is FORTRAN with a similar set of language extensions. Before we jump into CUDA FORTRAN code, it is helpful to summarize some of differences between these two programming interfaces to the CUDA architecture.

FORTRAN is a strongly typed language, and this strong typing carries over into the CUDA FORTRAN implementation. Device data declared in CUDA FORTRAN host code is declared with the `device` variable attribute, unlike CUDA C where both host and device data are declared the same way. Differentiating host and device data when variables are declared can simplify several aspects of dealing with device data. Allocation of device data can occur where the variable is declared, for example

```
real, device :: a_d(N)
```

will allocate `a_d` to contain `N` elements on device 0. Device data can also be declared as allocatable, and allocated using the FORTRAN 90's `allocate` statement:

```
real, device, allocatable :: a_d(:)
...
allocate(a_d(N))
```

where the FORTRAN `allocate` routine has been overloaded to allocate arrays on the current device in the same way `cudaMalloc` does in CUDA C. CUDA FORTRAN's strong typing also affects how data transfers between the host and the device can be performed. While one can use the `CudaMemcpy` function to perform host-to-device and device-to-host blocking transfers, it is far easier to use assignment statements:

```
real :: a(N)
real, device :: a_d(N)
...
a_d = a
```

where the FORTRAN array assignment kicks off a `cudaMemcpy` behind the scenes. Transfer via assignment statements applies only to blocking or synchronous transfers; for asynchronous transfers one must use the `cudaMemcpyAsync` call.

CUDA FORTRAN makes use of other variable attributes besides the `device` attribute. The attributes `shared`, `constant`, `pinned`, and `value` also find frequent use in CUDA FORTRAN. Shared memory used in device code uses the `shared` variable attribute just as CUDA C uses the

__shared__ qualifier. Constant memory must be declared in a FORTRAN module that contains the device code where it is used, and the module must be used in the host code where it is initialized. The initialization of constant data in the host code is done via an assignment statement rather than by function calls. Pinned host memory is declared using the pinned variable attribute, and must also be declared allocatable. Since FORTRAN passes data by reference by default and in CUDA we typically deal with separate memory spaces for the host and the device, host parameters passed to a kernel via the argument list must be declared in the kernel with the value variable attribute.

CUDA FORTRAN also uses the attributes(global) and attributes (device) function attributes in the same way CUDA C uses declaration specifiers __global__ and __device__ to declare kernels and device functions.

Within CUDA FORTRAN device code the predefined variables gridDim, blockDim, blockIdx, and threadIdx are available as they are in CUDA C. Following typical FORTRAN convention, the components of blockIdx and threadIdx have a unit, rather than 0, offset, so a typical index calculation would look like the following:

```
i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
```

This is in contrast to CUDA C's:

```
i = blockDim.x*blockIdx.x + threadIdx.x;
```

This rounds out the major differences in the expression of CUDA concepts between CUDA C and CUDA FORTRAN. The CUDA FORTRAN notation will become clearer as we go through several examples in the following sections.

17.2 A FIRST CUDA FORTRAN PROGRAM

The SAXPY routine has been used several times to illustrate various aspects of CUDA programming, and we continue this tradition with our first CUDA FORTRAN example:

```
module mathOps
contains
attributes(global) subroutine saxpy(x, y, a)
real :: x(:), y(:)
real, value :: a
integer :: i, n
n = size(x)
i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
if (i <= n) y(i) = y(i) + a*x(i)
```

```
 end subroutine saxpy
end module mathOps

program testSaxpy
 use cudafor
 use mathOps
 implicit none
 integer, parameter :: N = 40000
 real :: x(N), y(N), a
 real, device :: x_d(N), y_d(N)
 type(dim3) :: grid, tBlock

 tBlock = dim3(256,1,1)
 grid = dim3(ceiling(real(N)/tBlock%x),1,1)

 x = 1.0; y = 2.0; a = 2.0

 x_d = x
 y_d = y
 call saxpy<<<grid,tBlock>>>(x_d, y_d, a)
 y = y_d

 write(*,*) 'Max error: ', maxval(abs(y-4.0))
 end program testSaxpy
```

In this complete code the SAXPY kernel is defined in the FORTRAN module mathOps using the attributes(global) qualifier. The kernel has three arguments: the 1D arrays x and y, and the scalar value a. The size of the x and y arrays does not need to be passed as a kernel argument since x and y are declared as assumed-shape arrays allowing the FORTRAN size () intrinsic to be used. Because a is defined on the host and must be passed by value, the value variable attribute is required in a's declaration in the kernel. The predefined blockDim, blockIdx, and threadIdx variables are used to calculate a global index i used to access elements of x and y. Once again note that blockIdx and threadIdx have a unit offset as opposed to CUDA C's zero offset. After checking for inbound access, the SAXPY operation is performed.

The host code uses the cudafor module, which defines CUDA runtime API routines, constants, and types, such as the type(dim3) used to declare the execution configuration variables grid and tBlock. In the host code, both host arrays x and y are declared as well as their device counterparts, x_d and y_d, where the latter are declared with the device variable attribute. The thread block and grid are defined in the first executable lines of host code, where the ceiling function is used to launch enough blocks to process all array elements in the case that the size of the array is not

evenly divisible by the number of threads in a thread block. After the host arrays x and y, as well as the parameter a, are initialized, the assignment statements x_d = x and y_d = y are used to transfer the data from the host to the device. The scalar a is not passed to the device in this manner, as it is passed by value as a kernel argument. Since the transfers by assignment statement are blocking transfers, we can call the SAXPY kernel after the transfers without any synchronization. The kernel invocation specifies the execution configuration in the triple chevrons placed between the kernel name and its argument list as is done in CUDA C. Also similar to CUDA C, integer expressions can be used between the triple chevrons in place of the type(dim3) variables. This is followed by a device-to-host transfer of the resultant array, which is then checked for correctness.

17.3 MULTIDIMENSIONAL ARRAY IN CUDA FORTRAN

Multidimensional arrays are first-class citizens in FORTRAN, and the ease of dealing with multidimensional data in FORTRAN is extended to CUDA FORTRAN. We have already seen one aspect of this in array assignments used for transfers between the host and the device. The ease of programming kernel code is evident from the following CUDA FORTRAN implementation of matrix multiply:

```
module mathOps
 integer, parameter :: TILE_WIDTH = 16
 contains
 attributes(global) subroutine matrixMul(Md, Nd, Pd)
 implicit none
 real, intent(in) :: Md(:,:), Nd(:,:)
 real, intent(out) :: Pd(:,:)

 real, shared :: Mds(TILE_WIDTH, TILE_WIDTH)
 real, shared :: Nds(TILE_WIDTH, TILE_WIDTH)
 integer :: i, j, k, m, tx, ty, width
 real :: Pvalue

 tx = threadIdx%x; ty = threadIdx%y
 i = (blockIdx%x-1)*TILE_WIDTH + tx
 j = (blockIdx%y-1)*TILE_WIDTH + ty
 width = size(Md,2)

 Pvalue = 0.0
 do m = 1, width, TILE_WIDTH
 Mds(tx,ty) = Md(i,m + ty-1)
 Nds(tx,ty) = Nd(m + tx-1,j)
```

```
                  call syncthreads()
                  do k = 1, TILE_WIDTH
                  Pvalue = Pvalue + Mds(tx,k)*Nds(k,ty)
                  enddo
                  call syncthreads()
                  enddo
                  Pd(i,j) = Pvalue

                  end subroutine matrixMul
                 end module mathOps

                 program testMatrixMultiply
                  use cudafor
                  use mathOps
                  implicit none
                  integer,   parameter   ::   m = 4*TILE_WIDTH,   n = 6*TILE_WIDTH,
                   k = 2*TILE_WIDTH
                  real :: a(m,k), b(k,n), c(m,n), c2(m,n)
                  real, device :: a_d(m,k), b_d(k,n), c_d(m,n)
                  type(dim3) :: grid, tBlock
                  call random_number(a); a_d = a
                  call random_number(b); b_d = b

                  tBlock = dim3(TILE_WIDTH, TILE_WIDTH, 1)
                  grid = dim3(m/TILE_WIDTH, n/TILE_WIDTH, 1)

                  call matrixMul << < grid, tBlock >> > (a_d, b_d, c_d)
                  c = c_d

                  ! test against FORTRAN 90 matmul intrinsic
                  c2 = matmul(a, b)
                  write(*,*) 'max error: ', maxval(abs(c-c2))
                  end program testMatrixMultiply
```

The matrixMul kernel uses shared memory tiles Mds and Nds just as in the CUDA C code, however, passing in 2D arrays as kernel arguments allows for a more intuitive indexing on the global arrays Md and Nd when copying to shared memory.

17.4 OVERLOADING HOST/DEVICE ROUTINES WITH GENERIC INTERFACES

In the preceding matrix multiplication, we used the FORTRAN 90 matmul intrinsic to check our results. Because of the distinction between host and device data in the host code, it is possible to build generic interfaces that

overload routines to execute either on the host or on the device depending on whether the arguments are host or device data. To illustrate how this is done, we present a generic interface to the matrix multiplication example in the previous section:

```fortran
module mathOps
 integer, parameter :: TILE_WIDTH = 16

 interface matrixMultiply
 module procedure mmCPU, mmGPU
 end interface matrixMultiply
contains
 function mmCPU(a, b) result(c)
 implicit none
 real :: a(:,:), b(:,:), c(:,:)
 c = matmul(a,b)
 end function mmCPU

 function mmGPU(a_d, b_d) result(c)
 use cudafor
 implicit none
 real, device :: a_d(:,:), b_d(:,:)
 real :: c(:,:)
 real, device, allocatable :: c_d(:,:)
 integer :: m, n
 type(dim3) :: grid, tBlock

 m = size(c,1); n = size(c,2)
 allocate(c_d(m,n))
 tBlock = dim3(TILE_WIDTH, TILE_WIDTH, 1)
 grid = dim3(m/TILE_WIDTH, n/TILE_WIDTH, 1)
 call matrixMul <<< grid, tBlock >> >(a_d, b_d, c_d)
 c = c_d
 deallocate(c_d)
 end function mmGPU

 attributes(global) subroutine matrixMul(Md, Nd, Pd)
 implicit none
 real, intent(in) :: Md(:,:), Nd(:,:)
 real, intent(out) :: Pd(:,:)

 real, shared :: Mds(TILE_WIDTH, TILE_WIDTH)
 real, shared :: Nds(TILE_WIDTH, TILE_WIDTH)
 integer :: i, j, k, m, tx, ty, width
 real :: Pvalue

 tx = threadIdx%x; ty = threadIdx%y
 i = (blockIdx%x-1)*TILE_WIDTH + tx
```

```
      j = (blockIdx%y-1)*TILE_WIDTH + ty
      width = size(Md,2)

      Pvalue = 0.0
      do m = 1, width, TILE_WIDTH
      Mds(tx,ty) = Md(i,m + ty-1)
      Nds(tx,ty) = Nd(m + tx-1,j)
      call syncthreads()
      do k = 1, TILE_WIDTH
      Pvalue = Pvalue + Mds(tx,k)*Nds(k,ty)
      enddo
      call syncthreads()
      enddo
      Pd(i,j) = Pvalue

      end subroutine matrixMul
    end module mathOps

    program testMatrixMultiply
    use cudafor
    use mathOps
    implicit none
    integer,   parameter   ::   m = 4*TILE_WIDTH,   n = 6*TILE_WIDTH,
      k = 2*TILE_WIDTH
    real :: a(m,k), b(k,n), c(m,n), c2(m,n)
    real, device :: a_d(m,k), b_d(k,n)

    call random_number(a); a_d = a
    call random_number(b); b_d = b

    c = matrixMultiply(a_d, b_d)
    c2 = matrixMultiply(a, b)

    write(*,*) 'max error: ', maxval(abs(c-c2))
    end program testMatrixMultiply
```

The interface to `matrixMultiply` in this code is overloaded using two procedures defined in the module, `mmCPU` and `mmGPU`. `mmCPU` operates on host data and simply calls the FORTRAN 90 intrinsic `matmul`. `mmGPU` takes device data for the input matrices, and returns a host array with the result. (It could just have easily been defined to return a device array.) The device array used for the result in `mmGPU`, `c_d`, is a local array that is declared on the sixth line of `mmGPU`, and allocated on the tenth line of that routine. After this allocation, the locally defined execution configuration parameters are determined and the kernel is launched, which is followed by a device-to-host transfer and the de-allocation of `c_d`. The actual matrix

multiple kernel is not modified from the previous section. In the host code, `matrixMultiply` is used to access both of these routines.

17.5 **CALLING CUDA C VIA** ISO_C_BINDING

In the previous section we demonstrated how an interface can be used to allow a single call to perform operations on either the host or device depending on where the input data resides. An interface can also be used to call C or CUDA C functions from CUDA FORTRAN using the iso_-c_binding module introduced in FORTRAN 2003. Such functions can either be CUDA C routines developed by the user or library routines. In our matrix multiplication code, for example, we might wish to call the CUBLAS version of SGEMM rather than our hand-coded version. This can be done in the following manner:

```
module cublas_m
 interface cublasInit
 integer function cublasInit() bind(C,name = 'cublasInit')
 end function cublasInit
 end interface
 interface cublasSgemm
 subroutine cublasSgemm(cta,ctb,m,n,k,alpha,A,lda,B,ldb,beta,
  c,ldc) &
 bind(C,name = 'cublasSgemm')
 use iso_c_binding
 character(1,c_char), value :: cta, ctb
 integer(c_int), value :: k, m, n, lda, ldb, ldc
 real(c_float), value :: alpha, beta
 real(c_float), device :: A(lda,*), B(ldb,*), C(ldc,*)
 end subroutine cublasSgemm
 end interface cublasSgemm
end module cublas_m

program sgemmDevice
 use cublas_m
 use cudafor
 implicit none
 integer, parameter :: m = 100, n = 100, k = 100
 real :: a(m,k), b(k,n), c(m,n), c2(m,n)
 real, device :: a_d(m,k), b_d(k,n), c_d(m,n)
 real, parameter :: alpha = 1.0, beta = 0.0
 integer :: lda = m, ldb = k, ldc = m
 integer :: istat

 call random_number(a); a_d = a
```

```
call random_number(b); b_d = b
istat = cublasInit()
call    cublasSgemm('n','n',m,n,k,alpha,a_d,lda,b_d,ldb,beta,
 c_d,ldc)
c = c_d

c2 = matmul(a,b)
write(*,*) 'max error = ', maxval(abs(c-c2))
end program sgemmDevice
```

Here the module `cublas_m` contains interfaces for the CUBLAS routines `cublasInit` and `cublasSgemm`, which are bound to C functions as dictated by the `bind(C, name = '...')` clause. The `iso_c_binding` module is used in the `cublasSgemm` interface as this module contains the type kind parameters used in the declarations for the function arguments.

One could manually write these interfaces for all of the CUBLAS routines, but this has already been done in the `cublas` module provided with the PGI CUDA FORTRAN compiler. In the preceding code, one can simply remove the `cublas_m` module and change the `use cublas_m` to `use cublas` in the main program. The `cublas` module also contains generic interfaces to overload the standard BLAS functions to execute the CUBLAS versions when the array arguments are device arrays. So we can further change the preceding program to call `sgemm` rather than `cublasSgemm`. The complete program then becomes as follows:

```
program sgemmDevice
 use cublas
 use cudafor
 implicit none
 integer, parameter :: m = 100, n = 100, k = 100
 real :: a(m,k), b(k,n), c(m,n), c2(m,n)
 real, device :: a_d(m,k), b_d(k,n), c_d(m,n)
 real, parameter :: alpha = 1.0, beta = 0.0
 integer :: lda = m, ldb = k, ldc = m
 integer :: istat

 call random_number(a); a_d = a
 call random_number(b); b_d = b

 istat = cublasInit()
 call sgemm('n','n',m,n,k,alpha,a_d,lda,b_d,ldb,beta,c_d,ldc)
 c = c_d

 c2 = matmul(a,b)
 write(*,*) 'max error = ', maxval(abs(c-c2))
 end program sgemmDevice
```

17.6 KERNEL LOOP DIRECTIVES AND REDUCTION OPERATIONS

There are many occasions when one wishes to perform simple operations on device data, such as scaling or normalization of a device array. For such operations, it can be cumbersome to write separate kernels, and fortunately CUDA FORTRAN provides kernel CUDA FORTRAN loop directives, or CUF kernels. CUF kernels essentially allow the programmer to inline simple kernels in host code. For example, our SAXPY code using CUF kernels becomes

```
program testSaxpy
 use cudafor
 implicit none
 integer, parameter :: N = 40000
 real :: x(N), y(N), a
 real, device :: x_d(N), y_d(N)
 integer :: i

 x = 1.0; x_d = x
 y = 2.0; y_d = y
 a = 2.0

 !$cuf kernel do <<<*,*>>>
 do i = 1, N
 y_d(i) = y_d(i) + a*x_d(i)
 end do

 y = y_d

 write(*,*) 'Max error: ', maxval(abs(y-4.0))
 end program testSaxpy
```

In this complete code, the module containing the `saxpy` kernel has been removed and in its place in the host code is the loop that contains device arrays. The directive `!$cuf kernel do` informs the compiler to generate a kernel for the operation in the following `do` loop. The execution configuration can be manually specified in the $<<<\ldots,\ldots>>>$, or asterisks can be used to have the compiler choose an execution, as is done in this case. CUF kernels can operate on nested loops, and can use nondefault streams.

One particular useful aspect of CUF kernels is their ability to perform reductions. When the left side of an expression in a CUF kernel loop is a host scalar variable, a reduction operation is performed on the device. This is useful because coding a well-performing reduction in CUDA is not a

trivial matter. The calculation of the sum of the device array elements using compiler-generated CUF kernels looks like the following:

```
program testReduction
 use cudafor
 implicit none
 integer, parameter :: N = 40000
 real :: x(N), xsum
 real, device :: x_d(N)
 integer :: i

 x = 1.0; x_d = x
 xsum = 0.0

 !$cuf kernel do <<<*,*>>>
 do i = 1, N
 xsum = xsum + x_d(i)
 end do
 write(*,*) 'Error: ', abs(xsum - sum(x))
end program testReduction
```

17.7 DYNAMIC SHARED MEMORY

In our matrix multiplication example we demonstrated how static shared memory is used, which is essentially analogous to how it is declared in CUDA C. For dynamic shared memory, there are several options in CUDA FORTRAN. If a single dynamic shared memory array is used, then once again the CUDA FORTRAN implementation parallels what is done in CUDA C:

```
attributes(global) subroutine dynamicReverse1(d)
real :: d(:)
integer :: t, tr
real, shared :: s(*)

t = threadIdx%x
tr = size(d)-t+1

s(t) = d(t)
call syncthreads()
d(t) = s(tr)
end subroutine dynamicReverse1
```

where the shared memory array s, used to reverse elements of a single thread block array in this kernel, is declared with as an assumed-size array. The size of this dynamic shared memory array is determined from the

number of bytes of dynamic shared memory specified in the third execution configuration parameter:

```
threadBlock = dim3(n,1,1)
grid = dim3 (1 ,1 ,1)
...
call dynamicReverse1 << < grid,threadBlock,4*threadBlock%x >> >
  (d_d)
```

When multiple dynamic shared memory arrays are used in CUDA C, essentially one large block of memory is allocated and pointer arithmetic is used to determine offsets into this block for the various variables. In CUDA FORTRAN, automatic arrays are used:

```
attributes (global) subroutine dynamicReverse2(d, nSize)
real :: d(nSize)
integer, value :: nSize

integer :: t, tr
real, shared :: s(nSize)
t = threadIdx%x
tr = nSize-t + 1

s(t) = d(t)
call syncthreads()
d(t) = s(tr)
end subroutine dynamicReverse2
```

Here `nSize` is not known at compile time, hence `s` is not a static shared memory array. Any in-scope variable, such as a variable declared in the module that contains this kernel, can be used to determine the size of the automatic shared memory arrays. Multiple dynamic shared memory arrays of different types can be specified in this fashion. The total amount of dynamic shared memory must still be specified in the third execution configuration parameter.

17.8 ASYNCHRONOUS DATA TRANSFERS

Asynchronous data transfers are performed using the `cudaMemcpy*Async()` API calls as is done in CUDA C, with a couple of differences that apply not only to these asynchronous data transfer API calls but also to the synchronous `cudaMemcpy*()` variants. The first difference is that the size of the transfer specified in the third argument is in terms of the number of elements rather than the number of bytes, and the second is that the direction of transfer is an optional argument since the direction can be inferred from the types of the first two arguments.

As with CUDA C, for asynchronous transfers the host memory must be pinned, which is accomplished through the `pinned` variable attribute rather than through a specific allocation function. Pinned memory in CUDA FORTRAN must be allocatable, and can be allocated and de-allocated through the FORTRAN 90 `allocate()` and `deallocate()` statements.

To overlap kernel execution and data transfers, in addition to pinned host memory, the data transfer and kernel must use different, nondefault streams. Nondefault streams are required for this overlap because memory copy, memory set functions, and kernel calls that use the default stream begin only after all preceding calls on the device (in any stream) have completed, and no operation on the device (in any stream) commences until they are finished. The following is an example of overlapping kernel execution and data transfer:

```
real, allocatable, pinned :: a(:)
...
integer (kind = cuda_stream_kind) :: stream1, stream2
...
allocate(a(nElements))
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
istat = cudaMemcpyAsync(a_d , a, nElements, stream1)
call kernel << <gridSize ,blockSize ,0, stream2 >> >(b_d)
```

In this example, two streams are created and used in the data transfer and kernel executions as specified in the last arguments of the `cudaMemcpyAsync()` call and the kernels execution configuration. We make use of two device arrays, `a_d` and `b_d`, and assign work on `a_d` to `stream1` and `b_d` to `stream2`.

If the operations on a single data array in a kernel are independent, then data can be broken into chunks and transferred in multiple stages, multiple kernels launched to operate on each chunk as it arrives, and each chunk's results transferred back to the host when the relevant kernel completes. The following code segments demonstrate two ways of breaking up data transfers and kernel work to hide transfer time:

```
! baseline case - sequential transfer and execute
a = 0
istat = cudaEventRecord(startEvent ,0)
a_d = a
call kernel << <n/blockSize , blockSize >> >(a_d, 0)
a = a_d

istat = cudaEventRecord(stopEvent , 0)

! Setup for multiple stream processing
```

```
strSize = n / nStreams
strGridSize = strSize / blocksize
i = 1, nStreams
 istat = cudaStreamCreate(stream(i))
enddo

! asynchronous version 1: loop over {copy, kernel, copy}
a = 0
istat = cudaEventRecord(startEvent ,0)
do i = 1, nStreams
 offset = (i-1)* strSize
 istat=cudaMemcpyAsync(a_d(offset + 1), a(offset + 1), strSize,
  stream(i))
 call  kernel  << <strGridSize, blockSize, 0, stream(i)>> >
  (a_d, offset)
 istat=cudaMemcpyAsync(a(offset + 1), a_d(offset + 1), strSize,
  stream(i))
enddo
istat = cudaEventRecord(stopEvent , 0)
! asynchronous version 2:
! loop over copy, loop over kernel, loop over copy
a = 0
istat = cudaEventRecord(startEvent ,0)

do i = 1, nStreams
 offset = (i-1)* strSize
 istat=cudaMemcpyAsync(a_d(offset + 1), a(offset + 1), strSize,
  stream(i))
enddo
do i = 1, nStreams
 offset = (i-1)* strSize
 call  kernel  << <strGridSize, blockSize, 0, stream(i)>> >
  (a_d, offset)
enddo
do i = 1, nStreams
 offset = (i-1)* strSize
 istat = cudaMemcpyAsync(a(offset + 1), a_d(offset + 1), strSize
  ,stream(i))
enddo
istat = cudaEventRecord(stopEvent , 0)
```

The asynchronous cases are similar to the sequential case, only that there are multiple data transfers and kernel launches that are distinguished by different streams and an offset corresponding to the particular stream. In this code, we limit the number of streams to four, although for large arrays there is no reason why a larger number of streams could not be

used. Note that the same kernel is used in the sequential and asynchronous cases in the code, as an offset is sent to the kernel to accommodate the data in different streams. The difference between the two asynchronous versions is the order in which the copies and kernels are executed. The first version loops over each stream and for each stream issues a host-to-device copy, kernel, and device-to-host copy. The second version issues all host-to-device copies, then all kernel launches, and then all device-to-host copies. We also make use of a third approach, which is a variant of the second where a dummy event is recorded after each kernel launch:

```
do i = 1, nStreams
 offset = (i-1)* strSize
 call kernel << <strGridSize, blockSize, 0, stream(i)>> >
  (a_d, offset)
 ! Add a dummy event
 istat = cudaEventRecord(dummyEvent, stream(i))
enddo
```

At this point you may be asking why we have three versions of the asynchronous case. The reason is that these variants perform differently on different hardware. Running this code on the NVIDIA Tesla C1060 produces the following:

```
Device: Tesla C1060
Time for sequential transfer and execute (ms): 12.92381
Time for asynchronous V1 transfer and execute (ms): 13.63690
Time for asynchronous V2 transfer and execute (ms): 8.845888
Time for asynchronous V3 transfer and execute (ms): 8.998560
```

And on the NVIDIA Tesla C2050 we get the following:

```
Device: Tesla C2050
Time for sequential transfer and execute (ms): 9.984512
Time for asynchronous V1 transfer and execute (ms): 5.735584
Time for asynchronous V2 transfer and execute (ms): 7.597984
Time for asynchronous V3 transfer and execute (ms): 5.735424
```

To decipher these results we need to understand a bit more about how devices schedule and execute various tasks. CUDA devices contain engines for various tasks, and operations are queued up in these engines as they are issued. Dependencies between tasks in different engines are maintained, but within any engine all dependence is lost, as tasks in an engine's queue are executed in the order they are issued by the host thread. For example, the C1060 has a single copy engine and a single kernel engine. For the preceding code, timelines for the execution on the device are schematically shown in Figure 17.1. In this schematic we have assumed that the time required for the host-to-device transfer, kernel execution, and device-to-host transfer are

approximately the same, and in the code provided, a kernel was chosen to make these times comparable.

For the sequential kernel, there is no overlap in any of the operations as one would expect. For the first asynchronous version of our code the order of execution in the copy engine is: H2D stream(1), D2H stream(1), H2D stream(2), D2H stream(2), and so forth. This is why we do not see any speedup when using the first asynchronous version on the C1060: tasks were issued to the copy engine in an order that precludes any overlap of kernel execution and data transfer. For versions two and three, however, where all the host-to-device transfers are issued before any of the device-to-host transfers, overlap is possible as indicated by the lower execution time. From our schematic, we would expect the execution of versions two and three to be 8/12 of the sequential version, or 8.7 ms, which is what is observed in the timing in the code.

On the C2050, two features interact to cause different behavior than that observed on the C1060. The C2050 has two copy engines, one for host-to-device transfers and another for device-to-host transfers, in addition to a single kernel engine. Having two copy engines explains why the first asynchronous version achieves good speedup on the C2050: the
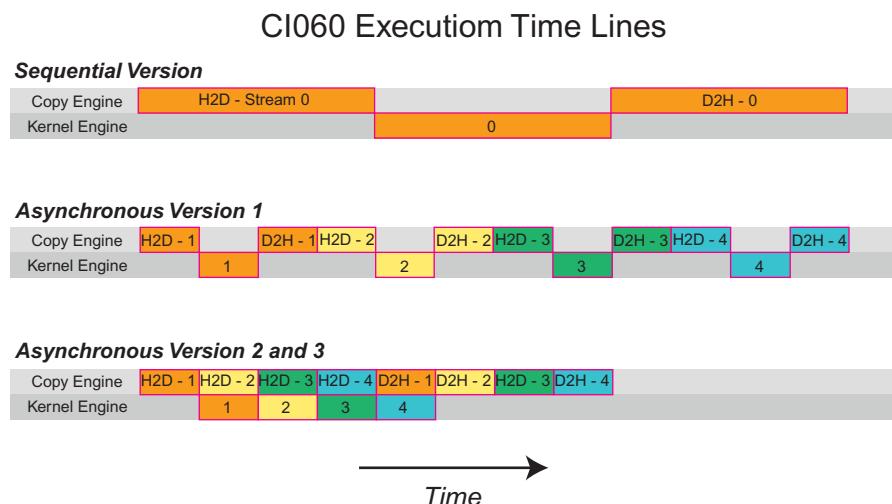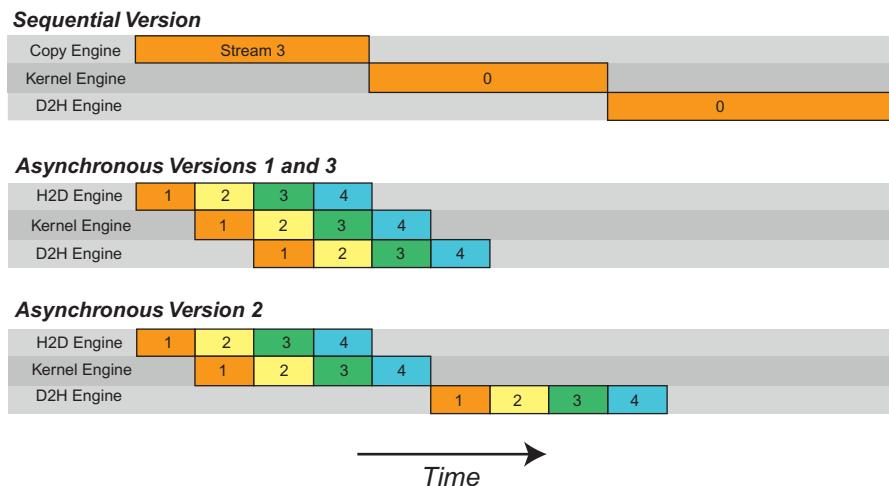
FIGURE 17.1

Data transfer and kernel execution timing for the sequential and asynchronous versions when there is only one copy engine.

Sequential Version

Asynchronous Versions 1 and 3

Asynchronous Version 2

Time

FIGURE 17.2

Data transfer and kernel execution timing for the sequential and asynchronous versions when there are two copy engines.

device-to-host transfer of data in `stream(i)` does not block the host-to-device transfer of data in `stream(i + 1)` as it did on the C1060 because these two operations are in different engines on the C2050, which is schematically shown in Figure 17.2.

From the schematic we would expect the execution time to be cut in half relative to the sequential version, which is roughly what is observed in the timings in the code. This does not explain the performance degradation observed in the second asynchronous approach, however, which is related to the C2050's support to concurrently run multiple kernels. When multiple kernels are issued back-to-back, the scheduler tries to enable concurrent execution of these kernels, and as a result delays a signal that normally occurs after each kernel completion (and is responsible for kicking off the device-to-host transfer) until all kernels complete. So, while there is overlap between host-to-device transfers and kernel execution in the second version of our asynchronous code, there is no overlap between kernel execution and device-to-host transfers. From Figure 17.2 one would expect an overall time for the second asynchronous version to be 9/12 of the time for the sequential version, or 7.5 ms, which is what we observe from the timings in the code. This situation can be rectified by recording a dummy CUDA event between each kernel, which will inhibit concurrent

kernel execution but will enable overlap of data transfers and kernel execution, as is done in the third asynchronous version.

17.9 COMPILATION AND PROFILING

CUDA FORTRAN codes are compiled using PGI FORTRAN compiler. Files with the `.cuf` or `.CUF` extensions have CUDA FORTRAN enabled automatically, and the compiler option `-Mcuda` can be used when compiling a file with other extensions to enable CUDA FORTRAN. Compilation of CUDA FORTRAN code can be as simple as issuing the command

```
pgf90 saxpy.cuf
```

Behind the scenes, a multistep process takes place. The first step is a source-to-source compilation where CUDA C device code is generated by CUDA FORTRAN. From there, compilation is similar to compilation of CUDA C. The device code is compiled into the intermediate representation PTX, and the PTX code is then further compiled to an executable code for a particular compute capability. The host code is compiled using `pgFORTRAN`. The final executable contains the host binary, the device binary, and the PTX. The PTX is included so that a new device binary can be created when the executable is run on a card of different compute capability than originally compiled for.

Specifics of this compilation process can be controlled through options to `-Mcuda`. A specific compute capability can be targeted, for example, `-Mcuda = cc20` generates executables for devices of compute capability 2.0. There is an emulation mode where device code is run on the host, specified by `-Mcuda = emu`. The specific version of the CUDA toolkit can be specified, for example, `-Mcuda = cuda4.0` causes compilation with the 4.0 CUDA toolkit. CUDA has a set of fast, but less accurate, intrinsics for single-precision functions like `sin()` and `cos()`, which can be enabled by the `-Mcuda = fastmath` option. Use of these functions requires no change in the CUDA FORTRAN source code, as the intermediate CUDA C code will be generated with the corresponding `__sinf()` and `__cosf()` functions, respectively. For finer (selective) control, the latter versions are available when the `cudadevice` module is used in the device code. The option `-Mucda = maxregcount:N` can be used to limit the number of registers used per thread to `N`. And the option `-Mcuda = ptxinfo` prints information on memory usage in kernels. Multiple options to `-Mcuda` can be given in a comma-separated list, for example, `-Mcuda = cc20,cuda4.0,ptxinfo`.

Profiling CUDA FORTRAN codes can be performed using the command-line profiling facility used in CUDA C. Setting the environment variable `COMPUTE_PROFILE` to 1,

```
% export COMPUTE_PROFILE = 1
```

and executing the code generates a file of profiling results, by default `cuda_profile_0.log`. For use of the command-line profiler see the documentation distributed with the CUDA toolkit.

17.10 CALLING THRUST FROM CUDA FORTRAN

Previously, we demonstrated calling external CUDA C libraries from CUDA FORTRAN, in particular the CUBLAS library, using the `iso_c_binding` module. In this section we demonstrate how CUDA FORTRAN can interface with Thrust, the standard template library for the GPU discussed in Chapter 16. Relative to calling CUDA C functions, interfacing with Thrust requires the additional step of creating C pointers that access the Thrust device containers, such as in the following code segment:

```
// allocate device vector
thrust::device_vector d_vec(4);
// obtain raw pointer to device vector's memory
int *ptr = thrust::raw_pointer_cast(&d_vec[0]);
```

The basic procedure to interface Thrust with CUDA FORTRAN is to create C wrapper functions that access Thrust's functions through standard C pointers, and then use the `iso_c_binding` module to access these functions through a generic interface in CUDA FORTRAN. For an example, we use Thrust's `sort` routine. The wrapper functions for the `int`, `float`, and `double` `sort` routines are as follows:

```
// Filename: csort.cu
// nvcc -c -arch sm_20 csort.cu
#include <thrust/device_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
extern "C" {
//Sort for integer arrays
void sort_int_wrapper( int *data, int N)
{
// Wrap raw pointer with a device_ptr
```

```
thrust::device_ptr <int> dev_ptr(data);
// Use device_ptr in Thrust sort algorithm
thrust::sort(dev_ptr, dev_ptr + N);
}
//Sort for float arrays
void sort_float_wrapper( float *data, int N)
{
thrust::device_ptr <float> dev_ptr(data);
thrust::sort(dev_ptr, dev_ptr + N);
}
//Sort for double arrays
void sort_double_wrapper( double *data, int N)
{
thrust::device_ptr <double> dev_ptr(data);
thrust::sort(dev_ptr, dev_ptr + N);
}
}
```

Compiling the code using

```
nvcc -c -arch sm_20 csort.cu
```

will generate an object file, `csort.o`, that we will use later on in the linking stage of the CUDA FORTRAN code.

With the C wrapper functions available, we can now write a FORTRAN module with a generic interface to Thust's `sort` functionality:

```
module thrust
 interface thrustsort
 subroutine sort_int(input,N) bind(C,name = "sort_int_wrapper")
 use iso_c_binding
 integer(c_int),device:: input(*)
 integer(c_int),value:: N
 end subroutine sort_int
 subroutine sort_float(input,N) bind(C,name = "sort_float_wrapper")
 use iso_c_binding
 real(c_float),device:: input(*)
 integer(c_int),value:: N
 end subroutine sort_float
 subroutine            sort_double(input,N)            bind(C,
  name = "sort_double_wrapper")
 use iso_c_binding
 real(c_double),device:: input(*)
 integer(c_int),value:: N
 end subroutine sort_double
 end interface thrustsort
 end module thrust
```

With the C wrapper functions and the FORTRAN module written, we can now turn to the main FORTRAN code that generates and transfers the data to the device, calls the sort functions, and transfers the data back to the host:

```fortran
program testsort
use thrust
! Declare two arrays, one on CPU (cpuData), one on GPU (gpuData)
real, allocatable :: cpuData(:)
real, allocatable, device :: gpuData(:)
integer:: N = 10
! Allocate the arrays using standard allocate
allocate(cpuData(N),gpuData(N))

! Generate random numbers on the CPU
do i = 1,N
cpuData(i) = random(i)
end do
cpuData(5) = 100.

print *,"Before sorting", cpuData

! Copy the data to GPU with a simple assignment
gpuData = cpuData

! Call the Thrust sorting function. The generic interface will
! select the proper routine, in this case the one operating on
 floats
call thrustsort(gpuData,size(gpuData))

! Copy the data back to CPU with a simple assignment
cpuData = gpuData

print *,"After sorting", cpuData

! Deallocate the arrays using standard deallocate
allocate(cpuData(N),gpuData(N))
end program testsort
```

If we save the module in a file `mod_thrust.cuf` and the program in `simplesort.cuf`, we are ready to compile and execute:

```
$ pgf90 -Mcuda = cc20 -O3 -o simple_sort mod_thrust.cuf simple_-
  sort.cuf csort.o

$ ./simple_sort
Before sorting 4.1630346E-02 0.9124327 0.7832350 0.6540373
100.0000 0.3956419 0.2664442 0.1372465
8.0488138E-03 0.8788511
```

```
After sorting 8.0488138E-03 4.1630346E-02 0.1372465 0.2664442
0.3956419 0.6540373 0.7832350 0.8788511
0.9124327 100.0000
```

We can modify the main code to evaluate the performance using the CUDA event API as follows:

```
program timesort
 use cudafor
 use thrust
 implicit none
 real, allocatable :: cpuData(:)
 real, allocatable, device :: gpuData(:)
 integer:: i,N = 100000000
 ! CUDA events for elapsing time
 type (cudaEvent):: startEvent , stopEvent
 real:: time, random
 integer:: istat

 ! Create events
 istat = cudaEventCreate(startEvent)
 istat = cudaEventCreate(stopEvent)

 ! Allocate arrays
 allocate(cpuData(N),gpuData(N))

 do i = 1,N
 cpuData(i) = random(i)
 end do

 print *,"Sorting array of ",N, " single precision"

 gpuData = cpuData

 istat = cudaEventRecord ( startEvent , 0)
 call thrustsort(gpuData,size(gpuData))

 istat = cudaEventRecord ( stopEvent , 0)
 istat = cudaEventSynchronize ( stopEvent )
 istat = cudaEventElapsedTime ( time , startEvent , stopEvent )

 cpuData = gpuData

 print *," Sorted array in:",time," (ms)"

 !Print the first five elements and the last five.
 print *,"After sorting", cpuData(1:5),cpuData(N-4:N)
 end program timesort
```

With the CUDA events, we are timing only the execution time of the sorting kernel. We can sort a vector of 100 M elements in 0.222 second on a Tesla M2050 with ECC on when the data is resident in GPU memory:

```
$ pgf90 -Mcuda = cc20 -O3 -o time_sort mod_thrust.cuf time_sort.
  cuf csort.o
$ ./time_sort
Sorting array of 100000000 single precision
Sorted array in: 222.1711 (ms)
After   sorting   7.0585919E-09   1.0318221E-08   1.9398616E-08
  3.1738640E-08
4.4078664E-08 0.9999999 0.9999999 1.000000 1.000000 1.000000
```

17.11 EXERCISES

17.1. Write a CUF kernel version of a matrix multiplication.

17.2. Write a CUDA FORTRAN code that reverses elements of a 4,096-element array.

An Introduction to C++ AMP

18

With special contributions from David Callahan

CHAPTER OUTLINE

C++ Accelerated Massive Parallelism, or C++ AMP, is a programming model for expressing data-parallel algorithms and exploiting heterogeneous computers using mainstream tools. C++ AMP was designed to offer productivity, portability, and performance. Developed initially by Microsoft, C++ AMP is defined by an open specification which takes input from multiple sources, including from AMD and NVIDIA. In this chapter we provide an overview of C++ AMP.

The focus of C++ AMP is to express the important data-parallel algorithm pattern while providing minimum new language features and shielding common scenarios from the intricacies of today's GPU programming. This provides a foundation of portability for applications written in C++ AMP across a range of different hardware. This portability creates future-proofing to preserve investment as hardware continues to evolve, as well as improving reusability of code across different devices and different manufacturers. At the same time, the full C++ AMP feature set includes advanced mechanisms for achieving performance when system intricacies must be addressed. In this chapter, we discuss first the most

straightforward examples of C++ AMP, and then we more lightly address these advanced features.

C++ AMP is a small extension to the current C++ 11 standard and is dependent on some of the core features of that standard. In particular, we will assume readers are familiar with modern C++, including the use of lambda expressions to build function closures, the use of templates for type-generic programming, the use of namespaces to control visibility of names, and the standard template library (STL). The common patterns are simple, so a deep understanding is not a prerequisite to use C++ AMP. Unlike CUDA and OpenCL, C++ AMP allows a rich subset of C++ inside data-parallel computations as well as using C++ for the host. C++ AMP has the same base compilation model as C++ with header files for interface specification and separate compilation units combined into a single executable.

C++ AMP does rely on two extensions to the language. The first places restrictions on the C++ operations that may be used in bodies of functions, and the second supports a form of limited cross-thread data sharing within data-parallel kernels. Both of these will be illustrated in Section 18.1. All other aspects of C++ AMP are delivered as a library accessed via a few header files.

C++ AMP shares many concepts with CUDA. In the following text we will illustrate this by showing C++ AMP equivalents for CUDA examples from earlier chapters. C++ AMP terminology differs from CUDA in small ways and we will highlight those differences as they arise.

18.1 CORE C++ AMP FEATURES

We describe the core features of C++ AMP by translating an example used in Chapter 3 from CUDA into C++ AMP. Figure 18.1 is the CUDA code for performing vector addition on host vectors using a CUDA device.

The corresponding C++ AMP code is shown in Figure 18.2. Line 1 includes the C++ AMP header, amp.h, which provides the declarations of the core features. The C++ AMP classes and functions are part of the concurrency namespace. The using directive on the next line makes the C++ AMP names visible in the current scope. It is optional but avoids the need to prefix C++ AMP names with a concurrency:: scope specifier.

The function vecAdd on line 4 in Figure 18.2 is functionally identical to the same function starting in line 6 in Figure 18.1. This function is

```
__global__ void vecAddKernel(float* d_A, float* d_B, float* d_C,
int n)
{
      int i = blockDim.x * blockIdx.x + threadIdx.x;
      if (i < n) C[i] = A[i] + B[i];
}
void vecAdd(float* A, float* B, float* C, int n)
{
      int size = n * sizeof (float); float* d_A, d_B, d_C;
      cudaMalloc((void **) &d_A, size);
      cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
      cudaMalloc((void **) &d_B, size);
      cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
      cudaMalloc((void **) &d_C, size);
      vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
      cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
      cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

FIGURE 18.1

CUDA vector addition from Chapter 3.

```
1  #include <amp.h>
2  using namespace concurrency;
3
4  void vecAdd(float* A, float* B, float* C, int n)
5  {
6       array_view<const float,1> AV(n,A), BV(n,B);
7       array_view<float,1> CV(n,C);
8       CV.discard_data();
9       parallel_for_each(CV.extent, [=](index<1> i) restrict(amp)
10      {
11          CV[i] = AV[i] + BV[i];
12      });
13      CV.synchronize();
14 }
```

FIGURE 18.2

Vector addition in C++ AMP.

executed by a thread running on the host and it contains a data-parallel computation that may be accelerated. The term *host* has the same meaning in C++ AMP documentation as in CUDA. While CUDA uses the term *device* to refer to the execution environment used for accelerated

execution, C++ AMP uses the term *accelerator*, which is discussed more in Section 18.3.

In C++ AMP, the primary vehicle for reading and writing large data collections is the class template `array_view`. An `array_view` provides a multidimensional reference to a rectangular collection of data locations. This is not a new copy of the data but rather a new way to access the existing memory locations. The template has two parameters: the type of the elements of the source data, and an integer that indicates the dimensionality of the `array_view`. Throughout C++ AMP, template parameters that indicate dimensionality are referred to as the *rank* of the type or object. In this example, we have a 1D `array_view` (or an `array_view` of rank 1) of C++ float values.

The constructor for array views of rank 1, such as `CV` on line 7 in Figure 18.2, takes two parameters. The first is an integer value that is the number of data elements. In general, the set of per-dimension lengths is referred to as an *extent*. To represent and manipulate extents, C++ AMP provides a class template, `extent`, with a single-integer template parameter that captures the rank. For objects with a low number of dimensions, various constructors are overloaded to allow specification of an extent as one or more integer values as is done for `CV`. The second parameter to the `CV` constructor is a pointer to the host data. In `vecAdd` the host data is expressed as a C-style pointer to contiguous data. An `array_view` may also overlay STL containers (see Section 16.1) such as `std::vector` when they support a `data` method to access underlying contiguous storage.

The CUDA code explicitly allocates memory (Figure 18.1, lines 9−13) that is accessible by the device and copies host data into it. These actions are implicit in C++ AMP by creating the association between an `array_view` and host data and subsequently accessing the data through the `array_view` on the accelerator. The method `array_view::discard_data` optimizes data transfers for some accelerators and is discussed in the next section. In this example, it is used when existing data values are immaterial because they are about to be overwritten.

Line 9 in Figure 18.2 illustrates the `parallel_for_each` construct that is the C++ AMP code pattern for a data-parallel computation. This corresponds to the kernel launch in CUDA (Figure 18.1, line 14). In CUDA terminology (as in Figure 3.3), the `parallel_for_each` creates a *grid of threads*. In C++ AMP the set of elements for which a computation is performed is called the *compute domain* and is defined by an `extent` object. Like in CUDA, each thread will invoke the same function for every point and threads are distinguished only by their location in the domain (grid).

Unlike CUDA, this domain need not be treated as an array of thread blocks (as in Figure 3.12). The `index` parameter combines information needed for common cases from the separate CUDA keyword `blockIdx.x`, `blockDim.x`, and `threadIdx.x`.

Similar to the standard C++ STL algorithm `for_each`, the `parallel_for_each` function template specifies a function to be applied to a collection of values. The first argument to a `parallel_for_each` is a C++ AMP extent object that describes the domain over which a data-parallel computation is performed. In this example, we perform an operation over every element in an `array_view` and so the `extent` passed into the `parallel_for_each` is the extent of the `CV` array view. In the example, this is accessed through the `extent` property of the `array_view` type. This is a 1D extent and the domain of the computation consists of integer values $0..(n - 1)$.

The second argument to a `parallel_for_each` is a C++ function object (or functor). In these examples we use the C++ 11 lambda syntax as a convenient way to build such an object. The core semantics of a `parallel_for_each` is to invoke the function defined by the second parameter exactly once for every element in the compute domain defined by the extent argument.

The leading [=] indicates that variables declared inside the containing function but referenced inside the lambda are "captured" and copied into data members of the function object built for the lambda. In this case, this will be the three `array_view` objects. The function invoked has a single parameter that is initialized to the location of a thread within the compute domain. This is again represented by a class template, `index`, which represents a short vector of integer values. The rank of an index is the length of this vector and is the same as the rank of the extent. The index parameter conveys the same information as the explicitly computed value `i` in the CUDA code (see Figure 18.1, line 3). These index values can be used to select elements in an array view as illustrated on line 11 of Figure 18.2.

A key extension to C++ is shown in this example: the `restrict(amp)` modifier. In C++ AMP, the existing C99 keyword `restrict` is borrowed and allowed in a new context: it may trail the formal parameter list of a function (including lambda functions). The `restrict` keyword is then followed by a parenthesized list of one or more restriction specifiers. While other uses are possible, in C++ AMP there are only two such specifiers defined: `amp` and `cpu`.

The function object passed to `parallel_for_each` must have its call operator annotated with a `restrict(amp)` specification. Any function called

from the body of that operator must similarly be restricted. The `restrict (amp)` specification is analogous to the __device__ keyword in CUDA. It identifies functions that may be invoked on a hardware accelerator. Analogously, `restrict(cpu)` corresponds to the CUDA __host__ keyword and indicates functions that may be invoked on the host. When no restriction is specified, the default is `restrict(cpu)`. C++ AMP has no need for an analog to the CUDA __global__ keyword. A function may have both restrictions, `restrict(cpu,amp)`, in which case it may be called in either host or accelerator contexts and must satisfy the restrictions of both contexts.

The `restrict` modifier allows a subset of C++ to be defined for use in a body of code. In the first release of C++ AMP, the restrictions reflect current common limitations of GPUs when used as accelerators of data-parallel code. The set of restrictions includes:

- No reference may be made to global or `static` variables except when they have a `const` type qualification and can be reduced to an integer literal value that is only used as an `rvalue`.
- A lambda expression used in a `parallel_for_each` must capture most variables by value with the exception of C++ AMP `array` and `texture` objects, each described later.
- Targets of function calls may not be virtual methods, pointers to functions, or pointers to member functions.
- Functions may not be recursively invoked and must be inlineable.
- Only `bool`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, and `void` may be used as C++ primitive types.
- C++ compound user-defined types are generally permitted but may not have `virtual` base classes or bit fields, and all data members and base classes must be 4-byte aligned.
- No use of `dynamic_cast` or `typeid` is permitted.
- No use of `goto` statements is permitted.
- No use of `asm` statements is permitted.
- No use of `try`, `catch`, or `throw` is permitted.

These restrictions reflect a common set of limitations for the GPU-based accelerators broadly available today. Over time we expect these restrictions to be lifted, and the open specification for C++ AMP includes a possible roadmap of future versions that are less restrictive. The `restrict(cpu)` specifier, of course, permits all of the capabilities of C++ but, because some functions that are part of C++ AMP are accelerator-specific, they do not have `restrict(cpu)` versions and so they may only be used in `restrict(amp)` code.

The restriction specifiers for a function are part of the type of the function, and function names may be overloaded when they have different restrictions. Thus, two functions may have identical signatures except one has the `restrict(amp)` specification and the other has the `restrict(cpu)` specification. This allows context-specific implementations of functions to be created. A function that has two overloads, one for each context, may be called from a `restrict(amp,cpu)` function, and the appropriate overload will be invoked that corresponds to whether the function is being invoked on the host or on an accelerator. In particular, this capability is used within C++ AMP to allow context-specific implementations of mathematic operations, but it is also available to application and library developers.

Inside the body of the `restrict(amp)` lambda (Figure 18.2, lines 10-12), there are references to the `array_view` objects declared in the containing scope. These are "captured" into the function object that is created to implement the lambda. Other variables from the function scope may also be captured by value. Each of these other values is made available to each invocation of the function executed on the accelerator. As for any C++ 11 nonmutable lambda, variables captured by value may not be modified in the body of the lambda. However, the elements of an `array_view` may be modified and those modifications will be reflected back to the host. In this example, any changes to `CV` made inside the `parallel_for_each` will be reflected in the host data `C` before the function `vecAdd` returns.

The final statement on line 13 in Figure 18.2 uses the `array_view::synchronize` method to ensure the underlying host data structure is updated with any changes. This is also discussed in the next section. This operation is not needed if the host accesses the data through the array view `CV`, but is needed to reliably access the data through the host pointer `C`. The central purpose of the `array_view` is to allow coherent access to data from both the host and the accelerator without the need for explicit synchronization or data copies.

Figure 18.3 is a more complex example borrowed from Chapter 12 and Figure 12.3. It performs a calculation on a slice of a 3D data structure. We use it to illustrate the handing of higher-dimensional `array_view` objects and compute domains. The function interface is essentially identical to the source where the CUDA `dim3` type is replaced with a C++ AMP `extent<3>` for the `grid` parameter. The contiguous data pointed to by `energygrid` is overlaid with a 3D `array_view` (named `energygrid_view`). C++ AMP follows a row-major storage layout so higher-numbered dimensions are less

```
#include <amp_math.h>
void cenergy_2(      float * energygrid, extent<3> grid,
                float gridspacing, float z, int k,
                const float * atoms, int numatoms) {
    array_view<float,3> energygrid_view(grid, energygrid);
    array_view<float,2> energy_slice = energygrid_view(k);
    energy_slice.discard_data();
    array_view<const float,2> atom_view(numatoms,4,atoms);
    parallel_for_each(energy_slice.extent, [=](index<2> ji)
    restrict(amp) {
        float y = gridspacing * float(ji[0]);
        float x = gridspacing * float(ji[1]);
        float energy = 0.0f;
        for(int n =0; n < numatoms; n++) {
            float dx = x - atom_view(n,0);
            float dy = y - atom_view(n,1);
            float dz = z - atom_view(n,2);
            energy + = atom_view(n,3)/
                        precise_math::sqrtf(dx*dx + dy*dy+dz*dz);
        }
        energy_slice[ji] = energy;
    });
    energy_slice.synchronize();
}
```

FIGURE 18.3

Base coulomb potential calculation code for a 2D slice.

significant in the linear storage order. C++ AMP has mechanisms to create an array_view that is a section of another array_view and also to project down to select a lower-dimensional slice. This operation is used on line 6 of Figure 18.3 to select the portion of the data actually defined by the kernel. As before, we use the discard_data method to avoid copying the immaterial existing values to the GPU. We overlay the atoms data with the 2D array_view named atom_view to simplify the expression of the accesses. This does not fundamentally change how the actual addressing arithmetic is performed, but seems to model the problem more accurately.

The data-parallel computation is then over the extent of the slice where the original sequential loop indices j and i are translated into the index < 2 > ji. Except for the indexing of atom_view, and the indexing into energy_slice, the body of the loop is largely unchanged.

C++ AMP provides a set of basic math operations for use in restrict(amp) contexts. These functions are accessed by including amp_math.h (which is not shown). The concurrency::fast_math and concurrency::precise_math names spaces respectively declare faster and more precise versions of functions. In the example, we chose to use

precise_math::sqrtf for illustration. In restrict(cpu) code, both of these namespaces establish aliases to std:: implementations of these functions, so a function that is declared restrict(cpu,amp) can still reference math functions and get the best implementation for the target.

To summarize this section, the core C++ AMP concepts include an array_view, which provides a multidimensional view into rectangular data; an extent, which is the shape of such a view and also the shape of a data-parallel computation; an index, which is used to select elements of an array_view or a data-parallel computation; the parallel_for_each, which launches a data-parallel computation; and restrict(amp) modified functions, which are evaluated at each point in that computation.

18.2 **DETAILS OF THE C++ AMP EXECUTION MODEL**

The core C++ AMP features noted in the previous section focus on expressing data parallelism essentially as a concurrent invocation of a collection of threads that access multidimensional arrays of data. Many accelerators today run in a separate memory and cannot directly access host data. Furthermore, these accelerators run concurrently with the continuing execution of host code. While minimizing the impact of these concerns, these aspects are part of the execution model of C++ AMP.

Explicit and Implicit Data Copies

C++ AMP provides the class template array to allocate storage on an accelerator. Similar to an array_view and with a nearly identical interface, an array has element type and rank template parameters. The constructor includes extent information. Unlike an array_view, an array allocates new storage on an accelerator. The data elements of an array may only be accessed from that accelerator and all operations that copy data between an array and host memory are explicit.

To illustrate this, consider Figure 18.4, which rewrites Figure 18.2 to use explicit array operations. Each array_view is replaced with an array declaration of the same extent. Lines 5 and 6 show explicit copies from host data to an array using the C++ AMP copy function template. The lambda is changed slightly to capture array variables by reference rather than the default mode of capturing variables by value as in the other examples. C++ AMP array objects must be captured by reference while array_view objects must be captured by value for the lambda used in a

```
1  void vecAdd(float* A, float* B, float* C, int n)
2  {
3      array<float,1> AA(n), BA(n);
4      array<float,1> CA(n);
5      copy(A,AA);
6      copy(B,BA);
7      parallel_for_each(CA.extent,
8              [&AA,&BA,&CA](index<1> i) restrict(amp)
9      {
10         CA[i] = AA[i] + BA[i];
11     });
12     copy(CA,C);
13 }
```

FIGURE 18.4

Explicit memory and copy management.

`parallel_for_each`. Line 12 specifies the data to be copied back to the host after completion of the computation.

On an accelerator that cannot access host memory, all of the operations in Figure 18.4 also happen for the code in Figure 18.2 but they are performed transparently either when the `parallel_for_each` is launched or when `array_view::synchronize` is called. The intended use of the explicit mechanisms is to provide more control of memory management and allow copy operations to be initiated earlier and overlapped with other computations (although overlapped copies can be achieved through other means).

When an `array_view` overlays storage on the host but is accessed on the accelerator, the data is copied to an unnamed `array` on that accelerator and the access is made to that `array`. This copy of the host data may persist for the remainder of the lifetime of the `array_view`. This allows the C++ AMP runtime to avoid redundant copies of the same data to the accelerator. C++ AMP provides operations to influence how and when data is copied between these implicit copies and the source storage. Line 8 of Figure 18.2 shows the use of `array_view::discard_data`. This method is an assertion that the values stored in the host storage are immaterial, for example, because they are about to be overwritten. The effect of this assertion is that when the `array_view` is subsequently used in a `parallel_for_each`, no copy is performed from the source data to the implicit `array` created for accelerator access.

When an unnamed array is created to hold a copy of data associated with an `array_view`, and that array may be modified, the C++ AMP
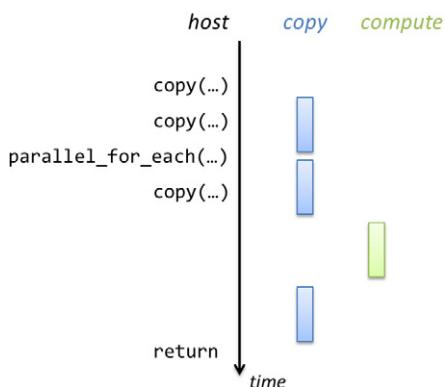
runtime system is permitted to copy the values back to the host storage immediately or leave them on the accelerator. If the `array_view` is destructed or an element is accessed on the host, then values will be copied promptly to make sure host accesses get the most recent definition. The method `array_view::synchronize` is available to force any such copies to be performed by a particular program point. The method `array_view::refresh` indicates to the C++ AMP runtime that all cached copies of the host data should be discarded. Generally, this method would be used when the underlying host data is modified directly without accessing through the `array_view`. This coherence between implicit cached copies and the underlying host data is the responsibility of the programmer.

An `array_view` may also refer to an `array`. This allows data allocated on an accelerator to be accessed by the host. Again, where necessary, this may involve creating copies of the data that are accessible by the host. The copies of data values between the source storage on the accelerator and the copies on the host are controlled using the same mechanisms and functions as before.

Asynchronous Operation

Most C++ AMP operations that initiate work on an accelerator, including operations to copy data to the accelerator, are *asynchronous.* This means that the host operation returns and the host thread continues to the next statement before the work completes. We illustrate this in Figure 18.5, which shows three strands of concurrent activity where time logically flows from the top to the bottom of the figure. On the left is the sequence of host operations that initiate accelerator operations. In the middle, we indicate three copy operations that take some duration each. On the right, we show the actual data-parallel computation that begins after the two copies to the accelerator complete and finishes before the final copy back to the host begins. On the host, the final copy-out is called before the data is ready and that operation blocks until the copy completes. When it returns, the `return` statement executes and the function returns with updated host data.

To provide finer-grain notification on which operation on the accelerator is complete, C++ AMP provides the `completion_future` class. This class is analogous to `std::shared_future`, the C++ standard method for coordination with asynchronous operations. In particular, it provides the `completion_future::get` method that blocks the calling thread until the asynchronous operation completes. C++ AMP has variants of the

FIGURE 18.5

Concurrent host/accelerator execution.

```
1 parallel_for_each(CV.extent, [=](index<1> i) restrict(amp)
2 {
3         CV[i] = AV[i] + BV[i];
4 });
5 completion_future done = CV.synchronize_async();
6 otherProcessing(A,B);
7 done.get();
```

FIGURE 18.6

Overlapped accelerator and host processing.

methods discussed here that are nonblocking and return a `completion_future`. In particular there are `array_view::synchronize_async` and various overloads of `copy_async`. These will initiate the data transfer implied and return a synchronization object immediately rather than blocking the thread until the operation has completed. Figure 18.6 provides a simple illustration where we assume that following the vector add computation there is some other computation involving the unmodified host data `A` and `B`. Upon completion of that other processing, the host then waits for the results from the `parallel_for_each` to be available on the host by using the `completion_future::get` call on the object returned from the `array_view::synchronize_async` method. After the `get` call returns, the host vector `C` will hold the results.

As discussed in Chapter 3, CUDA has an explicit notion of global memory, which is accessible by all threads in a kernel. In C++ AMP this concept is only available by having `array` objects associated with an accelerator. C++ AMP does not provide a facility for having file-scope objects accessible by functions running on the accelerator the way CUDA interprets `__device__` as a qualification on file-scope object declarations. Similarly, C++ AMP does not expose a concept of constant memory although values captured in the top-level lambda passed to a `parallel_for_each` may be stored in constant memory. The differences between CUDA and C++ AMP represent conscious design choices for C++ AMP to simplify the programming model. Some elements of CUDA reflect specifics of current GPU architectures that are not necessarily present in other forms of accelerators or may be significantly less common in the future. C++ AMP chose to leave these as implementation details rather than part of the model.

Section Summary
In this section we have discussed the features of C++ AMP that support a discrete accelerator that does not share memory with the host and runs concurrently with host computations. The key features are the `array` data container, explicit `copy` operations, and explicit asynchronous work mechanisms. We also indicated when and where such copies are made when the more flexible `array_view` is used when targeting discrete accelerators. We discussed the relationship of CUDA memory types with that of C++ AMP.

18.3 MANAGING ACCELERATORS
A computer system may include multiple accelerators suitable for implementing C++ AMP data-parallel computations. This includes both specialized hardware accelerators such as GPUs and simply the use of multicore CPUs with SIMD instructions. A system may also have multiple GPUs that may or may not have similar hardware characteristics. C++ AMP has mechanisms to enumerate available accelerators and to manage how work is mapped to those accelerators.

The class `accelerator` is the C++ AMP abstraction used for a specific mechanism for implementing data parallelism. As shown in Figure 18.7, the `accelerator::get_all` static method returns a vector of

```
 1  accelerator find_accelerator() {
 2      vector<accelerator> accs = accelerator::get_all();
 3      auto result =
 4          find_if(accs.begin(), accs.end(), [](const accelerator& acc)
 5          {
 6              return acc.supports_double_precision &&
 7                     !acc.has_display;
 8          });
 9      if(result == accs.end())
10          throw std::string("No suitable accelerator found");
11      return *result;
12  }
```

FIGURE 18.7

Example of finding an accelerator.

available accelerators in the system. A few properties associated with each accelerator may be used to select one when special requirements are required. For example, support of double-precision data types is an optional feature. For compute-intensive applications, it may be desirable to avoid placing work on the GPU that is used to drive an interactive display. Other properties include the amount of memory dedicated to the accelerator (accelerator::dedicated_memory) and a std::wstring that uniquely identifies the device (accelerator::device_path). The example uses the STL std::find algorithm to capture this search.

In addition to finding a specific accelerator, a system may support multiple suitable accelerators. C++ AMP enables off-loading work from one or more host threads to multiple accelerators. All such accelerator instances are returned by the call to accelerator::get_all and they may be used concurrently by an application.

In C++ AMP, an accelerator_view is an object that refers to a specific underlying accelerator and can be used to specify that accelerator for the purpose of indicating where an array is allocated and where work for a particular parallel_for_each should be executed. Similar to a CUDA stream, (cudaStream_t), various operations performed against a particular accelerator_view are performed in order but operations on different accelerator_views have no defined order.

In C++ AMP there is a default accelerator that is automatically selected by the runtime but can be explicitly set using the accelerator:: set_default static method, which takes a device path string parameter. Each accelerator has a default accelerator_view (accelerator::

default_view). The default view of the default accelerator is used for allocating an array when none is specified. A parallel_for_each may also have an explicit accelerator_view. Figure 18.8 is a variant of the vector add sample that makes use of defaults explicit. It is not necessary to use explicit arrays to direct work using an accelerator_view. Even when all data is accessed with array_view objects that overlay host data, a parallel_for_each may have an explicit accelerator_view indicating where the work should be performed.

Figure 18.9 is another illustration of explicit use of an accelerator_view. Here we provide a modified vector add operation that is parameterized by an accelerator_view that identifies where the work should be performed. The function determines the memory available on the accelerator, converted from kilobytes to bytes and used to determine the largest block size (block) where three blocks may be stored concurrently. Line 8 then loops over the input vectors in chunks of this size. For each chunk, a computation is launched as was done in Figure 18.2 but here the accelerator is explicitly specified by the first parameter, acc, to the parallel_for_each. On line 17, we initiate an asynchronous transfer of the results back to the host data structure. The completion_future returned by this operation is moved into a vector of such results. After all operations are started, lines 19 and 20 iterate over the vector of results using C++ STL methods and wait for each one to complete by calling the get method before the function returns to the caller.

```
1   void vecAdd (float* A, float* B, float* C, intn)
2   {
3         accelerator acc;
4         accelerator_viewview(acc.default_view);
5         array<float,1> AA(n,view), BA(n,view);
6         array<float,1> CA(n,view);
7         copy(A,AA);
8         copy(B,BA);
9         parallel_for_each(view, CA.extent,
10              [&AA,&BA,&CA](index<1> i) restrict(amp)
11        {
12           CA[i] = AA[i] + BA[i];
13        });
14        copy(CA,C);
15  }
```

FIGURE 18.8

Explicit accelerator use.

```
1  using std::vector;
2  void vecAddLong(float *A, float *B, float *C, int n,
3                  accelerator_view acc)
4  {
5      int block = (acc.accelerator.dedicated_memory * 1024)
6                          /(3*sizeof(float));
7      vector<completion_future> results;
8      for(int i = 0; i < n; i += block) {
9          int m = min(n-i,block);
10         array_view<const float,1> AV(m,A+i), BV(m,B+i);
11         array_view<float,1> CV(m,C+i);
12         CV.discard_data();
13         parallel_for_each(acc, CV.extent, [=](index<1> idx) restrict(amp)
14         {
15             CV[idx] = AV[idx] + BV[idx];
16         });
17         results.push_back(CV.synchronize_async());
18     }
19     std::for_each(results.begin(), results.end(),
20                 [](completion_future f) { f.get(); });
21 }
```

FIGURE 18.9

Explicit accelerator with asynchronous transfers.

18.4 TILED EXECUTION

This section touches on a topic important for some scenarios. We discuss a "tiled" version of data parallelism and the additional tools for optimizing memory available in that model.

As described earlier, a data-parallel computation has an associated computational domain defined by a C++ AMP `extent` object. A computational domain of rank 3 or less may also be blocked into regular, rectangular subdomains called *tiles*. The widths of these tiles must be compile-time constants. The threads that are associated with the same tile may share variables and participate in barrier synchronization. In CUDA, the term *block* is used to describe these groups of threads. A new storage class is also added to C++ AMP, `tile_static`, to indicate a variable that has a single instance per-tile that is shared by all threads (in CUDA this is indicated with the `__shared__` keyword). Chapter 5 discusses the motivation for using tiling and tile-shared variables to optimize memory bandwidth. Objects with this storage class may only be accessed in `restrict (amp)` code.

We illustrate tiling as was done in Chapter 5 by using matrix multiplication. Figure 5.12 shows a CUDA kernel that we expand here into a host

function (Figure 18.10) containing the kernel, as well as assuming host pointers are used to refer to dense arrays following the interface from Chapter 5. As before, we overlay `array_view` objects on top of the host data and discard the output data that is about to be overwritten so it is not copied to the accelerator.

A `tiled_extent` is a form of `extent` that captures tile dimensions as template parameters. C++ AMP only supports tiling for one, two, and three dimensions, and the rank of a `tiled_extent` object is inferred from the number of tile dimensions specified. In this case, the `tiled_extent` has rank 2 (line 6).

The `parallel_for_each` method has an overload for `tiled_extent`. The structure is the same as before and the lambda function will be invoked once for each element in the compute domain. C++ AMP requires that the extent of the compute domain must be evenly divisible by the tile size. In this example, `Width` must be multiples of `TILE_WIDTH`. When this condition is not met, a runtime exception is thrown.

```
1  void MatrixMul(float * M, float * N, float *P, int Width) {
2      extent<2> dims(Width,Width);
3      array_view<const float,2> d_M(dims,M), d_N(dims,N);
4      array_view<float,2> d_P(dims,P);
5      d_P.discard_data();
6      tiled_extent<TILE_WIDTH,TILE_WIDTH> tiled(dims);
7      parallel_for_each(tiled,
8          [=](tiled_index<TILE_WIDTH,TILE_WIDTH> t_idx) restrict(amp){
9              tile_static float Mds[TILE_WIDTH][TILE_WIDTH];
10             tile_static float Nds[TILE_WIDTH][TILE_WIDTH];
11             int tx  = t_idx.local[0],  ty  = t_idx.local[1];
12             int Row = t_idx.global[0], Col = t_idx.global[1];
13             float Pvalue = 0;
14             for (int m = 0; m < Width/TILE_WIDTH; ++m) {
15                 Mds[tx][ty] = d_M(m*TILE_WIDTH+tx, Row);
16                 Nds[tx][ty] = d_N(Col, m*TILE_WIDTH+ty);
17                 t_idx.barrier.wait();
18                 for(int k = 0; k < TILE_WIDTH; k++)
19                     Pvalue += Mds[tx][k] * Mds[k][ty];
20                 t_idx.barrier.wait();
21             }
22             d_P(Row,Col) = Pvalue;
23      });
24      d_P.synchronize();
25  }
```

FIGURE 18.10

Tiled matrix multiplication; compare Figure 5.12.

In the case of a `parallel_for_each` for a `tiled_extent`, the parameter to the lambda must be a `tiled_index` instead of an `index`. The `tiled_index` is a class template where again the tile sizes are captured as template parameters. The `tiled_index` (t_idx in Figure 18.10) provides both a mapping for each thread into the compute domain (`t_idx.global`) as well as the relative position of a thread within its tile (`t_idx.local`).

Line 9 declares a `tile_static` array named `Mds` that is shared by all threads in a tile. It will hold a copy of the values in `M` that are needed to perform a sub-block matrix multiplication computation for all of the threads in the tile. Similarly, line 10 declares analogous `Nds` to hold sub-blocks of `N`.

As in Figure 5.12, the loop on Figure 18.10, line 14, multiplies a block-row times a block column in tile-size chunks. The variable `Width` is used uniformly by all threads and is captured from the containing function scope for reuse in the lambda automatically. The threads in the tile cooperatively copy blocks of `M` and `N` into `tile_static` storage. Line 17 is the barrier synchronization point where all threads in the tile wait for the stores into shared variables to complete. A second barrier on line 20 makes sure all of the reads from shared variables are completed before writes on the next iteration begin. In C++ AMP, the object of type `tile_index` includes a `tile_barrier` object as a data member and that object provides methods to perform barriers. C++ AMP provides different forms of barriers that indicate whether the barrier applies to just `tile_static` data, global data, or both. Here we only need to protect `tile_static` data and so could use `wait_with_tile_static_memory_fence`, but chose to use the `wait` method to match the source from Chapter 5.

Figure 18.11 illustrates some details of C++ AMP tiling. It shows a 20×20 compute domain as a grid of small squares and the variable `e` in the code fragment. Rows (dimension 0) are shown as numbered from top to bottom and columns (dimension 1) from left to right. This domain might be blocked into 8×8 tiles. These tiles are illustrated with the larger black squares and the variable `te` or alternately the variable `te2`, which shows the `extent::tile` method template for creating a `tiled_extent`. We also illustrate the use of C++ 11 `auto` keyword to infer types of variables from their initializers.

Note that the tile size in this example does not evenly divide the dimensions of the compute domain. A tiled `parallel_for_each` requires the extent be a multiple of the tile size in each dimension, and the developer must explicitly handle the boundary cases when this is not the case.

```
                                    extent<2> e(20,20);
                                    tiled_extent<8,8> te(e);
                                    auto te2 = e.tile<8,8>();
                                    auto pte = te.pad();
                                    auto tte = te.truncate();
                                    parallel_for_each(pte,
                                        [](tiled_index<8,8> t_idx)
                                        restrict(amp)
                                    {
                                        t_idx.global; // 9,6 for example
                                        t_idx.tile_extent ; // 3,3
                                        t_idx.tile    ; // 1,0
                                        t_idx.tile_origin ; // 8,0
                                        t_idx.local   ; // 1,6
                                    });
```
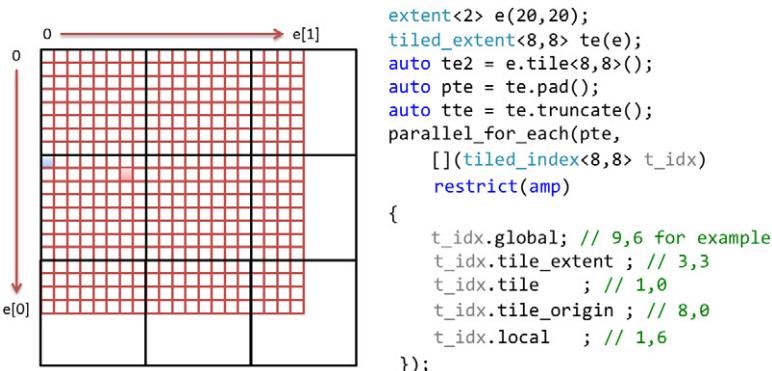
FIGURE 18.11

Illustration of tiling 20 × 20 compute domain.

The `tiled_extent` class template provides methods to either pad or truncate the underlying extent. In the example, variable `pte` corresponds to the padded exetent, `extent<2>(24,24)`, while the variable `tte` corresponds to the truncated extent, `extent<2>(16,16)`.

The `tiled_index` parameter supports a variety of members to facilitate tiled computations. The `global` member is an `index<2>` holding the position in the underlying compute domain. The solid square in the figure cooresponds to position (9,6) in the compute domain. The set of tiles (large squares) forms a domain, `extent<2>(3,3)` in this case, which is returned by the `tile_extent` member. The `tile` member is an `index<2>` holding the position of a point projected into this domain. The highlighed point (9,6) is in tile (1,0). The single lightly shaded square at the left edge is the first element in each dimension in the same tile as point (9,6). This is available as `tile_origin` and in this example corresponds to the global index (8,0). Finally, the points within a tile can be thought of as a small domain and the `local` member returns the position in this space (1,6) formed basically by subtracting `tile_origin` from `global`.

18.5 C++ AMP GRAPHICS FEATURES

The primary motivation for C++ AMP is to support data parallelism as an important algorithm pattern for general computing. Rendering and imaging processing are very important mainstream workloads for which C++ AMP includes some more specialized support, discussed briefly in

this section. These facilities include normalized floating points, short vector types, textures, and, optionally on Microsoft platforms, interoperations with DirectX. Many of these features are segregated into a separate namespace, `concurrency::graphics`. Figure 18.12 illustrates some of the types defined in that namespace and discussed in this section.

C++ AMP provides two types, `norm` and `unorm`, which provide arithmetic that is floating point in nature but of bounded range. The `norm` type holds signed values with magnitude no more than one while the `unorm` type holds non-negative values with magnitude no more than one. Common arithmetic operations are defined on these types where result values that would exceed the range are forced to the extreme value ("clamped"). These types may be mixed with C++ types and convert to `float`. They may also be used as element types for C++ AMP composite types `array`, `array_view`, and the `texture` objects described in the following.

Graphics programs frequently manipulate short vectors of primitive types. C++ AMP supports graphics programming by including definitions of these. For C++ AMP types, `int`, `unsigned int` (as `uint`), `float`, `double`, `norm`, and `unorm`, and for each vector length 2, 3, and 4, there exist types such as `int_2`, `uint_3`, and `float_4`. Each of these holds a number of component values that are accessed by name. The names supported are `x`, `y`, `z`, and `w`, or alternately `r`, `g`, `b`, and `w`. Thus, given the declarations in Figure 18.12, we might access a component `f4.z`, which is

```
1  #include <amp_graphics.h>
2  using namespace graphics;
3  …
4          norm n;          // normalized types
5          unorm u;
6          float_2 f2;
7          float_4 f4;      // short vector types;
8          int_2 i2;
9          norm_2 n2;
10         f2 = f4.xy + i2.x*f4.zw;
11         // usable in arrays, array_views
12         extent<2> e(1024,1024);
13         array<norm_2,2> an2(e);
14         // and in textures
15         texture<unorm_4,2> tu2(e, data, e.size() * 16U, 16U);
16         writeonly_texture_view<unorm_4,2> wotv(tu2);
```

FIGURE 18.12

Example of types from `concurrency::graphics`.

a single float that can be used as either an `rvalue` or an `lvalue`. Certain compound patterns are also supported, such as `f4.xy`, which corresponds to a short vector of suitable length, `float_2` in this case, that may be used as either an `rvalue` or `lvalue`. Assignment and arithmetic on short vectors is done in a component-wise style with scalar arguments promoted to vectors with that value in each component.

A `texture` is a special form of array that allows data-parallel code to access values that are stored using reduced precision. This is a common representation for image data and is the only method in the first version of C++ AMP to access partial word data types in a `restrict(amp)` context. Like an `array`, a `texture` is a class template that is parameterized by an element type and a rank. The set of allowed element types is constrained to be a subset of the `restrict(amp)` compatible primitive types and their short vector variants.

When a `texture` is constructed, in addition to the extent and a data source, a final unsigned integer argument indicates the number of bits per primitive data value used to store the value. Line 15 shows an example texture with a four-wide vector of unsigned normalized floating-point values. The `16U` passed to the constructor indicates each of these values is stored with only 16 bits of information. Not all combinations of data type, vector length, and storage width are supported (details in the specification are listed in the C++ AMP open specification, *http://blogs. msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx*).

A texture is a storage container like an `array` and may be associated with a particular `accelerator_view`. A texture is also indexed like an `array` with overloads of the index operator with an `index` instance of suitable rank as a parameter. As for `array`, these operations are `restrict (amp)` and may not be used in the host code. Overloads of the function template `copy` support transfers to and from host data structures.

A subset of textures may be written to directly and this is done explicitly via a `texture::set` method. For texture formats for which writing is not directly supported by hardware accelerators, C++ AMP provides the `writeonly_texture_view` class template illustrated with the variable named `wotv` (line 16 of Figure 18.12). The `set` method on this object may be used in a `restrict(amp)` context that is defining values in a texture.

Beyond support for these types, C++ AMP on Microsoft platforms includes specific features to enable interoperation with the DirectX framework. These interfaces are available in two namespaces : `concurrency:: direct3d` contains make_array, get_buffer and create_accelerator_view

while concurrency::graphics::direct3d contains make_texture. They include the following capabilities:

- Treating an existing Direct 3D device interface pointer as a C++ AMP acclerator_view.
- Treating an existing Direct 3D buffer interface pointer as a C++ AMP array.
- Treating an existing Direct 3D texture interface pointer as a C++ AMP texture.

These capabilities allow C++ AMP to provide a C++ language solution for GPU compute scenarios that integrates smoothly with the DirectX rendering framework.

Figure 18.13 illustrates the interop features. Function my_rotate consumes a vector of vertex data that is located on the host. Parameter d3ddevice is the existing DirectX interface that is used to first construct an accelerator_view and then an array. The parallel_for_each performs a rotation of the vertices where the result is left on the accelerator. Since the array instance vertices is located on a particular accelerator_view, the parallel_for_each will be executed on that same accelerator_view. We extract the underlying buffer object (typed only as IUnknown) and return this to the caller for subsequent use in scene rendering.

```
1   struct Vertex2D { float_2 Pos; };
2   IUnknown * my_rotate(ID3D11Device* d3ddevice, float THETA,
3                        int num_elements, const float_2 * data)
4   {
5       // copy data into a DX buffer
6       accelerator_view acc = create_accelerator_view(d3ddevice);
7       array<Vertex2D,1> vertices(num_elements, data, acc);
8       parallel_for_each(vertices.extent,
9                         [=, &vertices] (index<1> idx) restrict(amp) {
10          // Rotate the vertex by angle THETA
11          float_2 pos = vertices[idx].Pos;
12          vertices[idx].Pos.y = pos.y * cos(THETA) - pos.x * sin(THETA);
13          vertices[idx].Pos.x = pos.y * sin(THETA) + pos.x * cos(THETA);
14      });
15      // return the DX buffer use of transformed data.
16      return get_buffer(vertices);
17  }
```

FIGURE 18.13

Example DirectX interop—rotate vertex list.

18.6 SUMMARY

This chapter has presented an overview of C++ AMP, a small extension to C++ 11 to support hardware acceleration of data-parallel computations. The discussion is not complete, but the full specification is available at *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx*. The focus of C++ AMP is to create features that integrate well into modern C++ and leverage features such as templates, lambdas, and futures to provide a highly productive set of abstractions that compose with other aspects of C++ and parallelism. The features are layered to allow use by a very broad set of developers with limited knowledge of computer architecture, as well as providing access to the rich execution model needed for the most performance-critical scenarios. Lowering the barrier to expressing data parallelism and ensuring portability across hardware platforms will help more applications deliver the benefits of hardware acceleration and heterogeneous computing.

18.7 EXERCISES

18.1. Translate the simple, untiled version of matrix multiplication into C++ AMP. The CUDA kernel is shown in Figure 4.7. Write a host function that applies this computation to three `array_view<float,2>` inputs. Rather than implementing C = A*B, accumulate in the output and implement C+= A*B.

18.2. Given an array view of rank 2, X, `index<2>` ij, and `extent<2>` e, the operation X.section(ij,e) returns a new `array_view` that overlays the same data as X. If we denote this new view as S, then for all valid indices `idx` of S we have S[idx] is the same location as X[idx + ij].

Assume now there are three `array_view<float,2>` objects, A, B, and C. Assume they will not fit simultaneously in the `dedicated_memory` of the `accelerator` in the system. Use the `array_view::section` method, explicit array objects, and the matrix multiply building block from the first exercise to implement matrix multiplication for the large arrays.

18.3. Assume `std::vector gpu` holds two elements of type `accelerator_view` that refer to different but similar GPUs in a

system. Modify the solution to Exercise 18.3 to use both accelerators to implement the work.

18.4. Translate the tiled version of matrix transpose from Exercise 4.2 into C++ AMP.

18.5. The inner loop in Figure 18.3 redundantly loads data through `atom_view` that is used in multiple threads and these references are not coalesced (see Section 6.2). Rewrite the function in Figure 18.3 to use `tile_static` memory to improve the memory efficiency for accessing the data in `atom_view`.

Programming a Heterogeneous Computing Cluster

19

With Special Contributions from Isaac Gelado and Javier Cabezas

CHAPTER OUTLINE

So far we have focused on programming a heterogeneous computing system with one host and one device. In high-performance computing (HPC), many applications require the aggregate computing power of a cluster of computing nodes. Many of the HPC clusters today have one or more hosts and one or more devices in each node. Historically, these clusters have been programmed predominately with the Message Passing Interface (MPI). In this chapter, we will present an introduction to joint MPI/CUDA programming. Readers should be able to easily extend the material to joint MPI/OpenCL, MPI/OpenACC, and so on. We will only present the key MPI concepts that programmers need to understand to scale their heterogeneous applications to multiple nodes in a cluster environment. In particular, we will focus on domain partitioning, point-to-point communication, and collective communication in the context of scaling a CUDA kernel into multiple nodes.

19.1 BACKGROUND

While there was practically no top supercomputer using GPUs before 2009, the need for better energy efficiency has led to fast adoption of GPUs in recent years. Many of the top supercomputers in the world today use both CPUs and GPUs in each node. The effectiveness of this approach is validated by their high rankings in the Green 500 list, which reflects their high energy efficiency.

The dominating programming interface for computing clusters today is MPI [Gropp1999], which is a set of API functions for communication between processes running in a computing cluster. MPI assumes a distributed memory model where processes exchange information by sending messages to each other. When an application uses API communication functions, it does not need to deal with the details of the interconnect network. The MPI implementation allows the processes to address each other using logical numbers, much the same way as using phone numbers in a telephone system: telephone users can dial each other using phone numbers without knowing exactly where the called person is and how the call is routed.

In a typical MPI application, data and work are partitioned among processes. As shown in Figure 19.1, each node can contain one or more processes, shown as clouds within nodes. As these processes progress, they may need data from each other. This need is satisfied by sending and receiving messages. In some cases, the processes also need to synchronize with each other and generate collective results when collaborating on a large task. This is done with collective communication API functions.

19.2 A RUNNING EXAMPLE

We will use a 3D stencil computation as a running example. We assume that the computation calculates heat transfer based on a finite difference
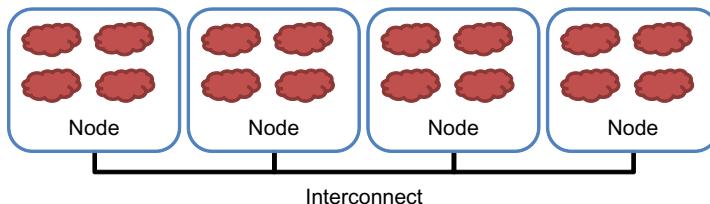
FIGURE 19.1

Programmer's view of MPI processes.

method for solving a partial differential equation that describes the physical laws of heat transfer. In each step, the value of a grid point is calculated as a weighted sum of neighbors (north, east, south, west, up, down) and its own value from the previous time step. To achieve high numerical stability, multiple indirect neighbors in each direction are also used in the computation of a grid point. This is referred to as a *higher-order stencil* computation. For the purpose of this chapter, we assume that four points in each direction will be used. As shown in Figure 19.2, there are a total of 24 neighbor points for calculating the next step value of a grid point. As shown in Figure 19.2, each point in the grid has an x, y, and z coordinate. For a grid point where the coordinate value is $x = i$, $y = j$, and $z = k$, or (i,j,k), its 24 neighbors are $(i-4,j,k)$, $(i-3,j,k)$, $(i-2,j,k)$, $(i-1,j,k)$, $(i+1,j,k)$, $(i+2,j,k)$, $(i+3,j,k)$, $(i+4,j,k)$, $(i,j-4,k)$, $(i,j-3,k)$, $(i,j-2,k)$, $(i,j-1,k)$, $(i,j+1,k)$, $(i,j+2,k)$, $(i,j+3,k)$, $(i,j+4,k)$, $(i,j,k-4)$, $(i,j,k-3)$, $(i,j,k-2)$, $(i,j,k-1)$, $(i,j,k+1)$, $(i,j,k+2)$, $(i,j,k+3)$, and $(i,j,k+4)$. Since the next data value of each grid point is calculated based on the current data values of 25 points (24 neighbors and itself), the type of computation is often called *25-stencil computation*.

We assume that the system is modeled as a structured grid, where spacing between grid points is constant within each direction. This allows us to use a 3D array where each element stores the state of a grid point. The physical distance between adjacent elements in each dimension can be represented by a spacing variable. Figure 19.3 illustrates a 3D array that represents a rectangular ventilation duct, with x and y dimensions as the cross-sections of the duct and the z dimension the direction of the heat flow along the duct.
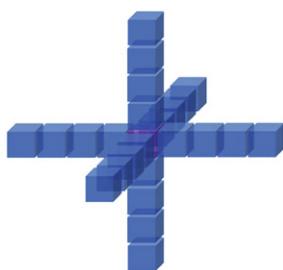
FIGURE 19.2

A 25-stencil computation example, where the neighbors are in the x, y, and z directions.

We assume that the data is laid out in the memory space and that x is the lowest dimension, y is the next, and z is the highest. That is, all elements with $y = 0$ and $z = 0$ will be placed in consecutive memory locations according to their x coordinate. Figure 19.4 shows a small example of the grid data layout. This small example has only 16 data elements in the grid: two elements in the x dimension, two in the y dimension, and four in the z dimension. Both x elements with $y = 0$ and $z = 0$ are placed in memory first. They are followed by all elements with $y = 1$ and $z = 0$. The next group will be elements with $y = 0$ and $z = 1$.

When one uses a cluster, it is common to divide the input data into several partitions, called *domain partitions*, and assign each partition to a node in the cluster. In Figure 19.3, we show that the 3D array is divided into four domain partitions: D1, D2, D3, and D4. Each of the partitions will be assigned to an MPI compute process.

The domain partition can be further illustrated with Figure 19.4. The first section, or slice, of four elements ($z = 0$) in Figure 19.4 is in the first partition, the second section ($z = 1$) the second partition, the third section ($z = 2$) the third partition, and the fourth section ($z = 3$) the fourth partition. This is obviously a toy example. In a real application, there are typically hundreds or even thousands of elements in each dimension. For the rest of this chapter, it is useful to remember that all elements in a z slice are in consecutive memory locations.

19.3 MPI BASICS

Like CUDA, MPI programs are based on the SPMD (single program, multiple data) parallel execution model. All MPI processes execute the same
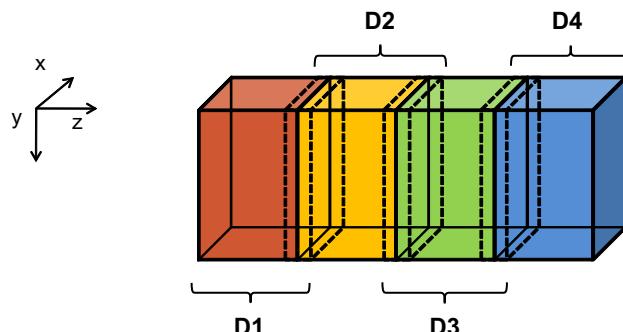
FIGURE 19.3

3D grid array for modeling the heat transfer in a duct.

program. The MPI system provides a set of API functions to establish communication systems that allow the processes to communicate with each other. Figure 19.5 shows five essential API functions that set up and tear down communication systems for an MPI application. Figure 19.6 shows a simple MPI program that uses these API functions. A user needs to supply the executable file of the program to the `mpirun` command or the `mpiexec` command in a cluster.

Each process starts by initializing the MPI runtime with a `MPI_Init()` call. This initializes the communication system for all the processes running the application. Once the MPI runtime is initialized, each process calls two functions to prepare for communication. The first function is `MPI_Comm_rank()`, which returns a unique number to calling each process, called an *MPI rank* or *process ID*. The numbers received by the processes vary from 0 to the number of processes minus 1. MPI rank for a process is equivalent to the expression `blockIdx.x*blockDim.x + threadIdx.x` for a CUDA thread. It uniquely identifies the process in a communication, similar to the phone number in a telephone system.

The `MPI_Comm_rank()` takes two parameters. The first one is an MPI built-in type `MPI_Comm` that specifies the scope of the request. Values of the `MPI_Comm` are commonly referred to as a *communicator*. `MPI_Comm` and other MPI built-in types are defined in a `mpi.h` header file that should be included in all C program files that use MPI. This is similar to the `cuda.h` header file for CUDA programs. An MPI application can create one or more *intracommunicators*. Members of each intracommunicator are MPI processes. `MPI_Comm_rank()` assigns a unique ID to each process in an intracommunicator. In Figure 19.6, the parameter value passed is `MPI_COMM_WORLD`, which means that the intracommunicator includes all MPI processes running the application.

The second parameter to the `MPI_Comm_rank()` function is a pointer to an integer variable into which the function will deposit the returned rank value. In Figure 19.6, a variable `pid` is declared for this purpose. After the `MPI_Comm_rank()` returns, the `pid` variable will contain the unique ID for the calling process.
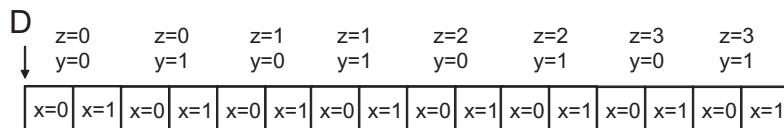
FIGURE 19.4

A small example of memory layout for the 3D grid.

- int MPI_Init (int*argc, char***argv)
  – Initialize MPI
- int MPI_Comm_rank (MPI_Comm comm, int *rank)
  – Rank of the calling process in group of comm
- int MPI_Comm_size (MPI_Comm comm, int *size)
  – Number of processes in the group of comm
- int MPI_Comm_abort (MPI_Comm comm)
  – Terminate MPI comminication connection with an error flag
- int MPI_Finalize ( )
  – Ending an MPI application, close all resources

FIGURE 19.5

Five basic MPI—API functions for establishing and closing a communication system.

```
#include "mpi.h"

int main(int argc, char *argv[]) {
    int pad = 0, dimx = 480+pad, dimy = 480, dimz = 400, nreps = 100;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np< 3) {
        if(0 == pid) printf("Needed 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_process(dimx, dimy, dimz/ (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz);

    MPI_Finalize();
    return 0;
 }
```

FIGURE 19.6

A simple MPI main program.

The second API function is MPI_Comm_size(), which returns the total number of MPI processes running in the intracommunicator. The MPI_Comm_size() function takes two parameters. The first one is an MPI

built-in type MPI_Comm that gives the scope of the request. In Figure 19.6, the scope is MPI_COMM_WORLD. Since we use MPI_COMM_WORLD, the returned value is the number of MPI processes running the application. This is requested by a user when the application is submitted using the mpirun command or the mpiexec command. However, the user may not have requested a sufficient number of processes. Also, the system may or may not be able to create all the processes requested. Therefore, it is good practice for an MPI application program to check the actual number of processes running.

The second parameter is a pointer to an integer variable into which the MPI_Comm_size() function will deposit the return value. In Figure 19.6, a variable np is declared for this purpose. After the function returns, the variable np contains the number of MPI processes running the application. In Figure 19.6, we assume that the application requires at least three MPI processes. Therefore, it checks if the number of processes is at least three. If not, it calls the MPI_Comm_abort() function to terminate the communication connections and return with an error flag value of 1.

Figure 19.6 also shows a common pattern for reporting errors or other chores. There are multiple MPI processes but we need to report the error only once. The application code designates the process with pid = 0 to do the reporting.

As shown in Figure 19.5, the MPI_Comm_abort() function takes two parameters. The first is the scope of the request. In Figure 19.6, the scope is all MPI processes running the application. The second parameter is a code for the type of error that caused the abort. Any number other than 0 indicates that an error has happened.

If the number of processes satisfies the requirement, the application program goes on to perform the calculation. In Figure 19.6, the application uses np-1 processes (pid from 0 to np-2) to perform the calculation and one process (the last one of which the pid is np-1) to perform an input/output (I/O) service for the other processes. We will refer to the process that performs the I/O services as the data server and the processes that perform the calculation as compute processes. In Figure 19.6, if the pid of a process is within the range from 0 to np-2, it is a compute process and calls the compute_process() function. If the process pid is np-1, it is the data server and calls the data_server() function.

After the application completes its computation, it notifies the MPI runtime with a call to the MPI_Finalize(), which frees all MPI communication resources allocated to the application. The application can then exit with a return value 0, which indicates that no error occurred.

19.4 MPI POINT-TO-POINT COMMUNICATION TYPES

MPI supports two major types of communication. The first is the point-to-point type, which involves one source process and one destination process. The source process calls the MPI_Send() function and the destination process calls the MPI_Recv() function. This is analogous to a caller dialing a call and a receiver answering a call in a telephone system.

Figure 19.7 shows the syntax for using the MPI_Send() function. The first parameter is a pointer to the starting location of the memory area where the data to be sent can be found. The second parameter is an integer that gives that number of data elements to be sent. The third parameter is of an MPI built-in type MPI_Datatype. It specifies the type of each data element being sent. The MPI_Datatype is defined in mpi.h and includes MPI_DOUBLE (double precision, floating point), MPI_FLOAT (single precision, floating point), MPI_INT (integer), and MPI_CHAR (character). The exact sizes of these types depend on the size of the corresponding C types in the host processor. See the MPI programming guild for more sophisticated uses of MPI types [Gropp 1999].

The fourth parameter for MPI_Send is an integer that gives the MPI rank of the destination process. The fifth parameter gives a tag that can be used to classify the messages sent by the same process. The sixth parameter is a communicator that selects the processes to be considered in the communication.

Figure 19.8 shows the syntax for using the MPI_Recv() function. The first parameter is a pointer to the area in memory where the received data should be deposited. The second parameter is an integer that gives the maximal number of elements that the MPI_Recv() function is allowed to receive. The third parameter is an MPI_Datatype that specifies the type

- int MPI_Send (void *buf, int count,
  MPI_Datatype datatype, int dest, int tag,
  MPI_Comm comm)
    - Buf: starting address of send buffer (pointer)
    - Count: Number of elements in send buffer (nonnegative integer)
    - Datatype: Datatype of each send buffer element (MPI_Datatype)
    - Dest: Rank of destination (integer)
    - Tag: Message tag (integer)
    - Comm: Communicator (handle)

FIGURE 19.7

Syntax for the MPI_Send() function.

(size) of each element to be received. The fourth parameter is an integer that gives the process ID of the source of the message.

The fifth parameter is an integer that specifies the particular tag value expected by the destination process. If the destination process does not want to be limited to a particular tag value, it can use MPI_ANY_TAG, which means that the receiver is willing to accept messages of any tag value from the source.

We will first use the data server to illustrate the use of point-to-point communication. In a real application, the data server process would typically perform data input and output operations for the compute processes. However, input and output have too much system-dependent complexity. Since I/O is not the focus of our discussion, we will avoid the complexity of I/O operations in a cluster environment. That is, instead of reading data from a file system, we will just have the data server initialize the data with random numbers and distribute the data to the compute processes. The first part of the data server code is shown in Figure 19.9.

The data server function takes four parameters. The first three parameters specify the size of the 3D grid: number of elements in the *x* dimension is dimx, the number of elements in the *y* dimension is dimy, and the number of elements in the *z* dimension is dimz. The fourth parameter specifies the number of iterations that need to be done for all the data points in the grid.

In Figure 19.9, line 1 declares variable np that will contain the number of processes running the application. Line 2 calls MPI_Comm_size(), which will deposit the information into np. Line 3 declares and initializes several helper variables. The variable num_comp_procs contains the number of compute processes. Since we are reserving one process as the data server, there are np-1 compute processes. The variable first_proc gives the

- int **MPI_Recv** (void *buf, int count,
  MPI_Datatype datatype, int source, int tag,
  MPI_Comm comm, MPI_Status *status)
  - buf: starting address of receive buffer (pointer)
  - Count: Maximum number of elements in receive buffer (integer)
  - Datatype: Datatype of each receive buffer element (MPI_Datatype)
  - Source: Rank of source (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)
  - Status: Status object (Status)

FIGURE 19.8

Syntax for the MPI_Recv() function.

process ID of the first compute process, which is 0. The variable last_proc gives the process ID of the last compute process, which is np-2. That is, line 3 designates the first np-1 processes, 0 through np-2, as compute processes. This reflects the design decision and the process with the largest rank serves as the data server. This decision will also be reflected in the compute process code.

Line 4 declares and initializes the num_points variable that gives the total number of grid data points to be processed, which is simply the product of the number of elements in each dimension, or dimx * dimy * dimz. Line 5 declares and initializes the num_bytes variable that gives the total number of bytes needed to store all the grid data points. Since each grid data point is a float, this value is num_points * sizeof(float).

Line 6 declares two pointer variables: input and output. These two pointers will point to the input data buffer and the output data buffer, respectively. Lines 7 and 8 allocate memory for the input and output buffers and assign their addresses to their respective pointers. Line 9 checks if the memory allocations were successful. If either of the memory allocations fail, the corresponding pointer will receive a NULL pointer from the malloc() function. In this case, the code aborts the application and reports an error.

```
void data_server(int dimx, int dimy, int dimz, int nreps) {
1.   int np,
/* Set MPI Communication Size */
2.   MPI_Comm_size(MPI_COMM_WORLD, &np);

3.   num_comp_nodes = np – 1, first_node = 0, last_node = np - 2;
4.   unsigned int num_points = dimx * dimy * dimz;
5.   unsigned int num_bytes  = num_points * sizeof(float);
6.   float *input=0, *output=0;
         /* Allocate input data */
7.   input = (float *)malloc(num_bytes);
8.   output = (float *)malloc(num_bytes);
9.   if(input == NULL || output == NULL) {
         printf("server couldn't allocate memory\n");
         MPI_Abort( MPI_COMM_WORLD, 1 );
     }
     /* Initialize input data */
10.  random_data(input, dimx, dimy ,dimz , 1, 10);
     /* Calculate number of shared points */
11.  int edge_num_points = dimx * dimy * (dimz / num_comp_nodes + 4);
12.  int int_num_points  = dimx * dimy * (dimz / num_comp_nodes + 8);
13.  float *send_address = input;
```

FIGURE 19.9

Data server process code (part 1).

Lines 11 and 12 calculate the number of grid point array elements that should be sent to each compute process. As shown in Figure 19.3, there are two types of compute processes. The first process (process 0) and the last process (process 3) compute an "edge" partition that has neighbors only on one side. Partition 0 assigned to the first process has a neighbor only on the right side (partition 1). Partition 3 assigned to the last process has a neighbor only on the left side (partition 2). We call the compute processes that compute edge partitions the *edge processes*.

Each of the rest of the processes computes an internal partition that has neighbors on both sizes. For example, the second process (process 1) computes a partition (partition 1) that has a left neighbor (partition 0) and a right neighbor (partition 2). We call the processes that compute internal partitions *internal processes*.

Recall that each calculation step for a grid point needs the values of its immediate neighbors from the previous step. This creates a need for halo cells for grid points at the left and right boundaries of a partition, shown as slices defined by dotted lines at the edge of each partition in Figure 19.3. Note that these halo cells are similar to those in the convolution pattern presented in Chapter 8. Therefore, each process also needs to receive one slice of halo cells that contains all neighbors for the boundary grid points of its partition. For example, in Figure 19.3, partition D2 needs a halo slice from D1 and a halo slice from D3. Note that a halo slice for D2 is a boundary slice for D1 or D3.

Recall that the total number of grid points is `dimx*dimy*dimz`. Since we are partitioning the grid along the *z* dimension, the number of grid points in each partition should be `dimx*dimy*(dimz / num_comp_procs)`. Recall that we will need four neighbor slices in each direction to calculate values within each slice. Because we need to send four slices of grid points for each neighbor, the number of grid points that should be sent to each internal process should be `dimx*dimy*(dimz/num_comp_procs + 8)`. As for an edge process, there is only one neighbor. Like in the case of convolution, we assume that zero values will be used for the ghost cells and no input data needs to be sent for them. For example, partition D1 only needs the neighbor slice from D2 on the right side. Therefore, the number of grid points to be sent to an edge process should be `dimx*dimy*(dimz/ num_comp_procs + 4)`. That is, each process receives four slices of halo grid points from the neighbor partition on each side.

Line 13 of Figure 19.9 sets the `send_address` pointer to point to the beginning of the input grid point array. To send the appropriate partition

to each process, we need to add the appropriate offset to this beginning address for each `MPI_Send()`. We will come back to this point later.

We are now ready to complete the code for the data server, shown in Figure 19.10. Line 14 sends process 0 its partition. Since this is the first partition, its starting address is also the starting address of the entire grid, which was set up in line 13. Process 0 is an edge process and it does not have a left neighbor. Therefore, the number of grid points to be sent is the value `edge_num_points`, that is, `dimx*dimy*(dimz/num_comp_procs +4)`. The third parameter specifies that the type of each element is an `MPI_FLOAT`, which is a C `float` (single precision, 4 bytes). The fourth parameter specifies that the value of `first_node` (i.e., 0) is the MPI rank of the destination process. The fifth parameter specifies 0 for the MPI tag. This is because we are not using tags to distinguish between messages sent from the data server. The sixth parameter specifies that the communicator to be used for sending the message should be all MPI processes for the current application.

Line 15 of Figure 19.10 advances the `send_address` pointer to the beginning of the data to be sent to process 1. From Figure 19.3, there are `dimx*dimy*(dimz/num_comp_procs)` elements in partition D1, which means D2 starts at a location that is `dimx*dimy*(dimz/num_comp_procs)` elements from the starting location of `input`. Recall that we also need to send the halo cells from D1 as well. Therefore, we adjust the starting address for the `MPI_Send()` back by four slices, which results in the expression for advancing the `send_address` pointer in line 15: `dimx*dimy* (dimz/num_comp_procs-4)`.

```
    /* Send data to the first compute node */
14. MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
              0, MPI_COMM_WORLD );

15. send_address += dimx * dimy * (dimz / num_comp_nodes - 4);
    /* Send data to "internal" compute nodes */
16. for(int process = 1; process < last_node; process++) {
17.     MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
              0, MPI_COMM_WORLD);
18.     send_address += dimx * dimy * (dimz / num_comp_nodes);
    }

    /* Send data to the last compute node */
19. MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
              0, MPI_COMM_WORLD);
```

FIGURE 19.10

Data server process code (part 2).

Line 16 is a loop that sends out the MPI messages to process 1 through process `np-3`. In our small example for four compute processes, `np` is 5. The loop sends the MPI messages to processes 1, 2, and 3. These are internal processes that need to receive halo grid points for neighbors on both sides. Therefore, the second parameter of the `MPI_Send()` in line 17 uses `int_num_nodes`, that is, `dimx*dimy*(dimz/num_comp_procs + 8)`. The rest of the parameters are similar to that for the `MPI_Send()` in line 14 with the obvious exception that the destination process is specified by the loop variable `process`, which is incremented from 1 to `np-3` (`last_node` is `np-2`).

Line 18 advances the send address for each internal process by the number of grid points in each partition: `dimx*dimy*dimz/num_comp_nodes`. Note that the starting locations of the halo grid points for internal processes are `dimx*dimy*dimz/num_comp_procs` points apart. Although we need to pull back the starting address by four slices to accommodate halo grid points, we do so for every internal process so the net distance between the starting locations remains as the number of grid points in each partition.

Line 19 sends the data to the process `np-2`, the last compute process that has only one neighbor to the left. Readers should be able to reason through all the parameter values used. Note that we are not quite done with the data server code. We will come back later for the final part of the data server that collects the output values from all compute processes.

We now turn our attention to the compute processes that receive the input from the data server process. In Figure 19.11, lines 1 and 2 establish the process ID for the process and the total number of processes for the application. Line 3 establishes that the data server is process `np-1`. Lines 4 and 5 calculate the number of grid points and the number of bytes that should be processed by each internal process. Lines 6 and 7 calculate the number of grid points and the number of bytes in each halo (four slices).

Lines 8-10 allocate the host memory and device memory for the input data. Note that although the edge processes need less halo data, they still allocate the same amount of memory for simplicity; part of the allocated memory will not be used by the edge processes. Line 10 sets the starting address of the host memory for receiving the input data from the data server. For all compute processes except process 0, the starting receiving location is simply the starting location of the allocated memory for the input data. However, we adjust the receiving location by four slices. This is because for simplicity, we assume that the host memory for receiving the input data is arranged the same way for all compute processes: four slices of halo from the left neighbor followed by the partition, followed by

four slices of halo from the right neighbor. However, we showed in line 4 of Figure 19.10 that the data server will not send any halo data from the left neighbor to process 0. That is, for process 0, the MPI message from the data server only contains the partition and the halo from the right neighbor. Therefore, line 10 adjusts the starting host memory location by four slices so that process 0 will correctly interpret the input data from the data server.

Line 12 receives the MPI message from the data server. Most of the parameters should be familiar. The last parameter reflects any error condition that occurred when the data is received. The second parameter specifies that all compute processes will receive the full amount of data from the data server. However, the data server will send less data to process 0 and process np-2. This is not reflected in the code because MPI_Recv() allows the second parameter to specify a larger number of data points than what is actually received, and will only place the actual number of bytes received from the sender into the receiving memory. In the case of process 0, the input data from the data server contains only the partition and the halo from the right neighbor. The received input will be placed by skipping the first four slices of the allocated memory, which should correspond to the halo for the (nonexistent) left neighbor. This effect is achieved with

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
      int np, pid;
1.    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
2.    MPI_Comm_size(MPI_COMM_WORLD, &np);
3.    int server_process = np - 1;


4.    unsigned int num_points       = dimx * dimy * (dimz + 8);
5.    unsigned int num_bytes        = num_points * sizeof(float);
6.    unsigned int num_halo_points = 4 * dimx * dimy;
7.    unsigned int num_halo_bytes  = num_halo_points * sizeof(float);


      /* Alloc host memory */
8.    float *h_input  = (float *)malloc(num_bytes);
          /* Alloca device memory for input and output data */
9.    float *d_input = NULL;
10.   cudaMalloc((void **)&d_input,  num_bytes );
11.   float *rcv_address = h_input + num_halo_points * (0 == pid);
12.   MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status );
13.   cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```

FIGURE 19.11

Compute process code (part 1).

the term `num_halo_points*(pid = = 0)` in line 11. In the case of process `np-2`, the input data contains the halo from the left neighbor and the partition. The received input will be placed from the beginning of the allocated memory, leaving the last four slices of the allocated memory unused.

Line 13 copies the received input data to the device memory. In the case of process 0, the left halo points are not valid. In the case of process `np-2`, the right halo points are not valid. However, for simplicity, all compute nodes send the full size to the device memory. The assumption is that the kernels will be launched in such a way that these invalid portions will be correctly ignored. After line 13, all the input data is in the device memory.
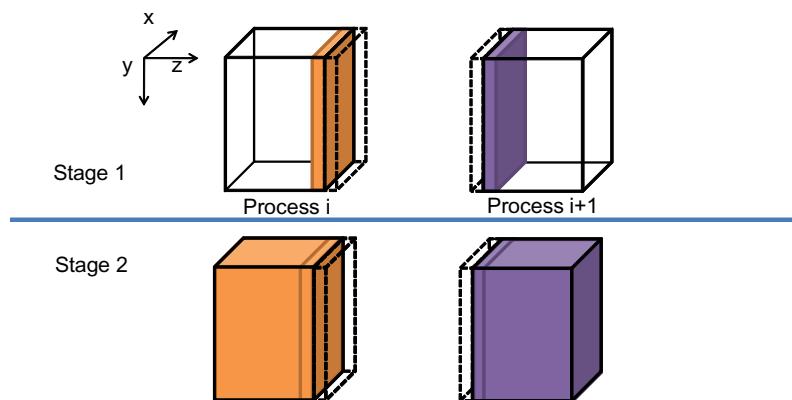
Figure 19.12 shows part 2 of the compute process code. Lines 14-16 allocate host memory and device memory for the output data. The output data buffer in the device memory will actually be used as a ping-pong buffer with the input data buffer. That is, they will switch roles in each iteration. We will return to this point later.

We are now ready to present the code that performs computation steps on the grid points.

19.5 OVERLAPPING COMPUTATION AND COMMUNICATION

A simple way to perform the computation steps is for each compute process to perform a computation step on its entire partition, exchange halo data with the left and right neighbors, and repeat. While this is a very simple strategy, it is not very effective. The reason is that this strategy forces the system to be in one of the two modes. In the first mode, all compute processes are performing computation steps. During this time, the communication network is not used. In the second mode, all compute processes exchange halo data with their left and right neighbors. During this time, the computation hardware is not well utilized. Ideally, we would like to achieve better performance by utilizing both the communication network and computation hardware all the time. This can be achieved by dividing the computation tasks of each compute process into two stages, as illustrated in Figure 19.13.

During the first stage (stage 1), each compute process calculates its boundary slices that will be needed as halo cells by its neighbors in the next iteration. Let's continue to assume that we use four slices of halo data. Figure 19.13 shows that the collection of four halo slices as a dashed transparent piece and the four boundary slices as a solid piece. Note that the solid piece of process `i` will be copied into the dashed piece of process

FIGURE 19.12

A two-stage strategy for overlapping computation with communication.

i + 1 and vice versa during the next communication. For process 0, the first phase calculates the right four slices of boundary data for four computation steps. For an internal node, it calculates the left four slices and the right four slices of its boundary data. For process n-2, it calculates the right four pieces of its boundary data. The rationale is that these boundary slices are needed by their neighbors for the next iteration. By calculating these boundary slices first, the data can be communicated to the neighbors while the compute processes calculate the rest of its grid points.

During the second stage (stage 2), each compute process performs two parallel activities. The first is to communicate its new boundary values to its neighbor processes. This is done by first copying the data from the device memory into the host memory, followed by sending MPI messages to the neighbors. As we will discuss later, we need to be careful that the data received from the neighbors is used in the next iteration, not the current iteration. The second activity is to calculate the rest of the data in the partition. If the communication activity takes a shorter amount of time than the calculation activity, we can hide the communication delay and fully utilize the computing hardware all the time. This is usually achieved by having enough slices in the internal part of each partition allow each compute process to perform computation steps in between communications.

To support the parallel activities in stage 2, we need to use two advanced features of the CUDA programming model: *pinned memory allocation* and *streams*. A pinned memory allocation requests that the memory

```
14. float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
15. float *h_output = (float *)malloc(num_bytes);
16. cudaMalloc((void **)&d_output, num_bytes );

17. float *h_left_boundary = NULL, *h_right_boundary = NULL;
18. float *h_left_halo = NULL, *h_right_halo = NULL;

    /* Alloc host memory for halo data */
19. cudaHostAlloc((void **)&h_left_boundary, num_halo_bytes, cudaHostAllocDefault);
20. cudaHostAlloc((void **)&h_right_boundary,num_halo_bytes, cudaHostAllocDefault);
21. cudaHostAlloc((void **)&h_left_halo,     num_halo_bytes, cudaHostAllocDefault);
22. cudaHostAlloc((void **)&h_right_halo,    num_halo_bytes, cudaHostAllocDefault);

    /* Create streams used for stencil computation */
23. cudaStream_t stream0, stream1;
24. cudaStreamCreate(&stream0);
25. cudaStreamCreate(&stream1);
```

FIGURE 19.13

Compute process code (part 2).

allocated will not be paged out by the operating system. This is done with the `cudaHostAlloc()` API call. Lines 19-22 allocate memory buffers for the left and right boundary slices and the left and right halo slices. The left and right boundary slices need to be sent from the device memory to the left and right neighbor processes. The buffers are used as a host memory staging area for the device to copy data into, and then used as the source buffer for `MPI_Send()` to neighbor processes. The left and right halo slices need to be received from neighbor processes. The buffers are used as a host memory staging area for `MPI_Recv()` to use as a destination buffer and then copied to the device memory.

Note that the host memory allocation is done with the `cudaHostAlloc()` function rather than the standard `malloc()` function. The difference is that the `cudaHostAlloc()` function allocates a *pinned memory* buffer, sometimes also referred to as *page-locked memory* buffer. We need to present a little more background on the memory management in operating systems to fully understand the concept of pinned memory buffers.

In a modern computer system, the operating system manages a virtual memory space for applications. Each application has access to a large, consecutive address space. In reality, the system has a limited amount of physical memory that needs to be shared among all running applications. This sharing is performed by partitioning the virtual memory space into pages and mapping only the actively used pages into physical memory. When there is much demand for memory, the operating system needs to "page out" some of the pages from the physical memory to mass storage such as disks. Therefore, an application may have its data paged out any time during its execution.

The implementation of `cudaMemcpy()` uses a type of hardware called a direct memory access (DMA) device. When a `cudaMemcpy()` function is called to copy between the host and device memories, its implementation uses a DMA device to complete the task. On the host memory side, the DMA hardware operates on physical addresses. That is, the operating system needs to give a translated physical address to the DMA device. However, there is a chance that the data may be paged out before the DMA operation is complete. The physical memory locations for the data may be reassigned to another virtual memory data. In this case, the DMA operation can be potentially corrupted since its data can be overwritten by the paging activity.

A common solution to this corruption problem is for the CUDA runtime to perform the copy operation in two steps. For a host-to-device copy, the CUDA runtime first copies the source host memory data into a "pinned" memory buffer, which means the memory locations are marked so that the operating paging mechanism will not page out the data. It then uses the DMA device to copy the data from the pinned memory buffer to the device memory. For a device-to-host copy, the CUDA runtime first uses a DMA device to copy the data from the device memory into a pinned memory buffer. It then copies the data from the pinned memory to the destination host memory location. By using an extra pinned memory buffer, the DMA copy will be safe from any paging activities.

There are two problems with this approach. One is that the extra copy adds delay to the `cudaMemcpy()` operation. The second is that the extra complexity involved leads to a synchronous implementation of the `cudaMemcpy()` function. That is, the host program cannot continue to execute until the `cudaMemcpy()` function completes its operation and returns. This serializes all copy operations. To support fast copies with more parallelism, CUDA provides a `cudaMemcpyAsync()` function.

To use the `cudaMemcpyAsync()` function, the host memory buffer must be allocated as a pinned memory buffer. This is done in lines 19-22 for the host memory buffers of the left boundary, right boundary, left halo, and right halo slices. These buffers are allocated with a special `cudaHostAlloc()` function, which ensures that the allocated memory is pinned or page-locked from paging activities. Note that the `cudaHostAlloc()` function takes three parameters. The first two are the same as `cudaMalloc()`. The third specifies some options for more advanced usage. For most basic use cases, we can simply use the default value `cudaHostAllocDefault`.

The second advanced CUDA feature is *streams*, which supports the managed concurrent execution of CUDA API functions. A stream is an ordered sequence of operations. When a host code calls a `cudaMemcpyAsync()` function or launches a kernel, it can specify a stream as one of its parameters. All operations in the same stream will be done sequentially. Operations from two different streams can be executed in parallel.

Line 23 of Figure 19.13 declares two variables that are of CUDA built-in type `cudaStream_t`. Recall that the CUDA built-in types are declared in `cuda.h`. These variables are then used in calling the `cudaStreamCreate()` function. Each call to the `cudaStreamCreate()` creates a new stream and deposits a pointer to the stream into its parameter. After the calls in lines 24 and 25, the host code can use either `stream0` or `stream1` in subsequent `cudaMemcpyAsync()` calls and kernel launches.

Figure 19.14 shows part 3 of the compute process. Lines 27 and 28 calculate the process ID of the left and right neighbors of the compute process. The `left_neighbor` and `right_neighbor` variables will be used by compute processes as parameters when they send messages to and receive messages from their neighbors. For process 0, there is no left neighbor, so line 27 assigns an MPI constant `MPI_PROC_NULL` to `left_neighbor` to note this fact. For process `np-2`, there is no right neighbor, so line 28 assigns `MPI_PROC_NULL` to `right_neighbor`. For all the internal processes, line 27 assigns `pid-1` to `left_neighbor` and `pid + 1` to `right_neighbor`.

Lines 31-33 set up several offsets that will be used to launch kernels and exchange data so that the computation and communication can be overlapped. These offsets define the regions of grid points that will need to be calculated at each stage of Figure 19.12. They are also visualized in Figure 19.15. Note that the total number of slices in each device memory is four slices of left halo points (dashed white), plus four slices of left boundary points, plus `dimx*dimy*(dimz-8)` internal points, plus four slices of boundary points, and four slices of right halo points (dashed white). Variable `left_stage1_offset` defines the starting point of the slices that are needed to calculate the left boundary slices. This includes 12 slices of data: 4 slices of left neighbor halo points, 4 slices of boundary points, and 4 slices of internal points. These slices are the leftmost in the partition so the offset value is set to 0 in line 31. Variable `right_stage2_offset` defines the starting point of the slices that are needed for calculating the right boundary slices. This also includes 12 slices: 4 slices of internal points, 4 slices of right boundary points, and 4 slices of right halo cells. The beginning point of these 12 slices can be derived by subtracting the

```
26. MPI_Status status;
27. int left_neighbor  = (pid > 0)    ? (pid - 1) : MPI_PROC_NULL;
28. int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

    /* Upload stencil cofficients */
    upload_coefficients(coeff, 5);

29. int left_halo_offset   = 0;
30. int right_halo_offset  = dimx * dimy * (4 + dimz);
31. int left_stage1_offset = 0;
32. int right_stage1_offset = dimx * dimy * (dimz - 4);
32. int stage2_offset      = num_halo_points;

34. MPI_Barrier( MPI_COMM_WORLD );
35. for(int i=0; I < nreps; i++) {
        /* Compute boundary values needed by other nodes first */
36.     launch_kernel(d_output + left_stage1_offset,
            d_input + left_stage1_offset, dimx, dimy, 12, stream0);
37.     launch_kernel(d_output + right_stage1_offset,
            d_input + right_stage1_offset, dimx, dimy, 12, stream0);

        /* Compute the remaining points */
38.     launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                      dimx, dimy, dimz, stream1);
```

FIGURE 19.14
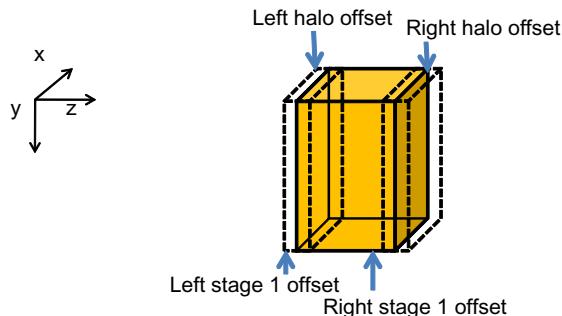
Compute process code (part 3).

total number of slices dimz + 8 by 12. Therefore, the starting offset for these 12 slices is dimx*dimy*(dimz-4).

Line 35 is an MPI barrier synchronization, which is similar to the CUDA_syncthreads(). An MPI barrier forces all MPI processes specified by the parameter to wait for each other. None of the processes can continue their execution beyond this point until everyone has reached this point. The reason why we want a barrier synchronization here is to make sure that all compute nodes have received their input data and are ready to perform the computation steps. Since they will be exchanging data with each other, we would like to make them all start at about the same time. This way, we will not be in a situation where a few tardy processes delay all other processes during the data exchange. MPI_Barrier() is a *collective communication* function. We will discuss more details about collective communication API functions in the next section.

Line 35 starts a loop that performs the computation steps. For each iteration, each compute process will perform one cycle of the two-stage process in Figure 19.12.

Line 36 calls a function that will perform four computation steps to generate the four slices of the left boundary points in stage 1. We assume

FIGURE 19.15

Device memory offsets used for data exchange with neighbor processes.

that there is a kernel that performs one computation step on a region of grip points. The `launch_kernel()` function takes several parameters. The first parameter is a pointer to the output data area for the kernel. The second parameter is a pointer to the input data area. In both cases, we add the `left_stage1_offset` to the input and output data in the device memory. The next three parameters specify the dimensions of the portion of the grid to be processed, which is 12 slices in this case. Note that we need to have four slices on each side. Line 37 does the same for the right boundary points in stage 1. Note that these kernels will be launched within `stream0` and will be executed sequentially.

Line 38 launches a kernel to generate the `dimx*dimy*(dimz-8)` internal points in stage 2. Note that this also requires four slices of input boundary values on each side so the total number of input slices is `dimx*dimy*dimz`. The kernel is launched in `stream1` and will be executed in parallel with those launched by lines 36 and 37.

Figure 19.16 shows part 4 of the compute process code. Line 39 copies the four slices of left boundary points to the host memory in preparation for data exchange with the left neighbor process. Line 40 copies the four slices of the right boundary points to the host memory in preparation for data exchange with the right neighbor process. Both are asynchronous copies in `stream0` and will wait for the two kernels in `stream0` to complete before they copy data. Line 40 is a synchronization that forces the process to wait for all operations in `stream0` to complete before it can continue. This makes sure that the left and right boundary points are in the host memory before the process proceeds with data exchange.

During the data exchange phase, we will have all MPI processes send their boundary points to their left neighbors. That is, all processes will have their right neighbors sending data to them. It is, therefore, convenient to have an MPI function that sends data to a destination and receives data from a source. This reduces the number of MPI function calls. The MPI_Sendrecv() function in Figure 19.17 is such a function. It is essentially a combination of MPI_Send() and MPI_Recv(), so we will not further elaborate on the meaning of the parameters.

Figure 19.18 shows part 5 of the compute process code. Line 42 sends four slices of left boundary points to the left neighbor and receives four slices of right halo points from the right neighbors. Line 43 sends four slices of right boundary points to the right neighbor and receives four slices of left halo points from the left neighbor. In the case of process 0, its left_neighbor has been set to MPI_PROC_NULL in line 27, so the MPI runtime will not send out the message in line 42 or receive the message in line 43 for process 0. Likewise, the MPI runtime will not receive the message in line 42 or send out the message in line 43 for process np-2. Therefore, the conditional assignments in lines 27 and 28 eliminate the need for special if-the-else statements in lines 42 and 43.

After the MPI messages have been sent and received, lines 44 and 45 transfer the newly received halo points to the d_output buffer of the device memory. These copies are done in stream0 so they will execute in parallel with the kernel launched in line 38.

Line 46 is a synchronize operation for all device activities. This call forces the process to wait for all device activities, including kernels and data copies to complete. When the cudaDeviceSynchronize() function returns, all d_output data from the current computation step are in place: left halo data from the left neighbor process, boundary data from the kernel launched in line 36, internal data from the kernel launched in line 38, right boundary data from the kernel launched in line 37, and right halo data from the right neighbor.

```
     /* Copy the data needed by other nodes to the host */
 39. cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
             num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
 40. cudaMemcpyAsync(h_right_boundary,
                d_output + right_stage1_offset + num_halo_points,
                num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
 41. cudaStreamSynchronize(stream0);
```

FIGURE 19.16

Compute process code (part 4).

- int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
    - Sendbuf: Initial address of send buffer (choice)
    - Sendcount: Number of elements in send buffer (integer)
    - Sendtype: Type of elements in send buffer (handle)
    - Dest: Rank of destination (integer)
    - Sendtag: Send tag (integer)
    - Recvcount: Number of elements in receive buffer (integer)
    - Recvtype: Type of elements in receive buffer (handle)
    - Source: Rank of source (integer)
    - Recvtag: Receive tag (integer)
    - Comm: Communicator (handle)
    - Recvbuf: Initial address of receive buffer (choice)
    - Status: Status object (Status). This refers to the receive operation.

FIGURE 19.17

Syntax for the `MPI_Sendrecv()` function.

```
        /* Send data to left, get data from right */
42.  MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
                 left_neighbor,  i, h_right_halo,
                 num_halo_points, MPI_FLOAT, right_neighbor, i,
                 MPI_COMM_WORLD, &status );
        /* Send data to right, get data from left */
43.  MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
                 right_neighbor, i, h_left_halo,
                 num_halo_points, MPI_FLOAT, left_neighbor,  i,
                 MPI_COMM_WORLD, &status );

44.  cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
                 num_halo_bytes, cudaMemcpyHostToDevice, stream0);
45.  cudaMemcpyAsync(d_output+right_ghost_offset, h_right_ghost,
                 num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
46.  cudaDeviceSynchronize();

47.  float *temp = d_output;
48.  d_output = d_input; d_input = temp;
   }
```

FIGURE 19.18

Compute process code (part 5).

Lines 47 and 48 swap the d_input and d_output pointers. This changes the output of the d_ouput data of the current computation step into the d_input data of the next computation step. The execution then proceeds to the next computation step by going to the next iteration of the loop of line 35. This will continue until all compute processes complete the number of computations specified by the parameter nreps.

Figure 19.19 shows part 6, the final part, of the compute process code. Line 46 is a barrier synchronization that forces all processes to wait for each other to finish their computation steps. Lines 50-52 swap d_output with d_input. This is because lines 47 and 48 swapped d_output with d_input in preparation for the next computation step. However, this is unnecessary for the last computation step. So, we use lines 50-52 to undo the swap. Line 53 copies the final output to the host memory. Line 54 sends the output to the data server. Line 55 waits for all processes to complete. Lines 56-59 free all the resources before returning to the main program.

Figure 19.20 shows part 3, the final part, of the data server process code, which continues from Figure 19.10. Line 20 waits for all compute nodes to complete their computation steps and send their outputs. This barrier corresponds to the barrier at line 55 of the compute process. Line 22 receives the output data from all the compute processes. Line 23 stores the output into an external storage. Lines 24 and 25 free resources before returning to the main program.

```
    /* Wait for previous communications */
49. MPI_Barrier(MPI_COMM_WORLD);

50. float *temp = d_output;
51. d_output = d_input;
52. d_input = temp;

    /* Send the output, skipping halo points */
53. cudaMemcpy(h_output, d_output, num_bytes, cudaMemcpyDeviceToHost);
    float *send_address = h_output + num_ghost_points;
54. MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
            server_process, DATA_COLLECT, MPI_COMM_WORLD);
55. MPI_Barrier(MPI_COMM_WORLD);

    /* Release resources */
56. free(h_input); free(h_output);
57. cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
58. cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
59. cudaFree( d_input ); cudaFree( d_output );
}
```

FIGURE 19.19

Compute process code (part 6).

19.6 MPI COLLECTIVE COMMUNICATION

The second type of MPI communication is collective communication, which involves a group of MPI processes. We have seen an example of the second type of MPI communication API: `MPI_Barrier`. The other commonly used group collective communication types are broadcast, reduction, gather, and scatter.

Barrier synchronization `MPI_Barrier()` is perhaps the most commonly used collective communication function. As we have seen in the stencil example, barriers are used to ensure that all MPI processes are ready before they begin to interact with each other. We will not elaborate on the other types of MPI collective communication functions but encourage readers to read up on the details of these functions. In general, collective communication functions are highly optimized by the MPI runtime developers and system vendors. Using them usually leads to better performance as well as readability and productivity.

19.7 SUMMARY

We covered basic patterns of joint CUDA/MPI programming in this chapter. All processes in an MPI application run the same program. However, each process can follow different control flow and function call paths to specialize their roles, as is the case of the data server and the compute

```
      /* Wait for nodes to compute */
20.   MPI_Barrier(MPI_COMM_WORLD);

      /* Collect output data */
21.   MPI_Status status;
22.   for(int process = 0; process < num_comp_nodes; process++)
         MPI_Recv(output + process * num_points / num_comp_nodes,
            num_points / num_comp_nodes, MPI_REAL, process,
            DATA_COLLECT, MPI_COMM_WORLD, &status );

      /* Store output data */
23.   store_output(output, dimx, dimy, dimz);

      /* Release resources */
24.   free(input);
25.   free(output);
   }
```

FIGURE 19.20

Data server process code (part 3).

processes in our example in this chapter. We also presented a common pattern where compute processes exchange data. We presented the use of CUDA streams and asynchronous data transfers to enable the overlap of computation and communication. We would like to point out that while MPI is a very different programming system, all major MPI concepts that we covered in this chapter—SPMD, MPI ranks, and barriers—have counterparts in the CUDA programming model. This confirms our belief that by teaching parallel programming with one model well, our students can quickly pick up other programming models easily. We would like to encourage readers to build on the foundation from this chapter and study more advanced MPI features and other important patterns.

19.8 EXERCISES

19.1. For vector addition, if there are 100,000 elements in each vector and we are using three compute processes, how many elements are we sending to the last compute process?

    a. 5

    b. 300

    c. 333

    d. 334

19.2. If the MPI call `MPI_Send(ptr_a, 1000, MPI_FLOAT, 2000, 4, MPI_COMM_WORLD)` resulted in a data transfer of 40,000 bytes, what is the size of each data element being sent?

    a. 1 byte

    b. 2 bytes

    c. 4 bytes

    d. 8 bytes

19.3. Which of the following statements is true?

    a. `MPI_Send()` is blocking by default.

    b. `MPI_Recv()` is blocking by default.

    c. MPI messages must be at least 128 bytes.

      d. MPI processes can access the same variable through shared memory.

19.4. Use the code base in Appendix A and examples in Chapters 3, 4, 5, and 6 to develop an OpenCL version of the matrix−matrix multiplication application.

Reference

Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message Passing Interface* (2nd ed.Cambridge, MA: MIT Press, Scientific and Engineering Computation Series.

This page intentionally left blank

CUDA Dynamic Parallelism

20

CUDA dynamic parallelism is an extension to the CUDA programming model enabling a CUDA kernel to create new thread grids by launching new kernels. Dynamic parallelism is introduced with the Kepler architecture, first appearing in the GK110 chip. In previous CUDA systems, kernels can only be launched from the host code. Algorithms that involved recursion, irregular loop structures, time-space variation, or other constructs that do not fit a flat, single level of parallelism needed to be implemented with multiple kernel launches, which increases burden on the host and amount of host-device communication. The dynamic parallelism support allows algorithms that dynamically discover new work to prepare and launch kernels without burdening the host. This chapter describes the extended capabilities of the CUDA architecture that enables dynamic parallelism, including the modifications and additions to the CUDA programming model necessary to take advantage of these, as well as guidelines and best practices for exploiting this added capacity.

20.1 **BACKGROUND**

Many real-world applications employ algorithms that dynamically vary the amount of work performed. For example, Figure 20.1 shows a turbulence simulation example where the level of required modeling details varies across space and time. As the combustion flow moves from left to right, the level of activities and intensity increases. The level of details required to model the right side of the model is much higher than that for the left side of the model. On one hand, using a fixed fine grid would incur too much work for no gain for the left side of the model. On the other hand, using a fixed coarse grid would sacrifice too much accuracy for the right side of the model. Ideally, one should use fine grids for the parts of the model that require more details and coarse grids for those that do not require as many details.

Previous CUDA systems require all kernels to be launched from the host code. The amount of work done by a thread grid is predetermined during kernel launch. With the SPMD programming style for the kernel code, it is tedious if not extremely difficult to have thread blocks to use different grid spacing. This limitation favors the use of fixed-grid systems
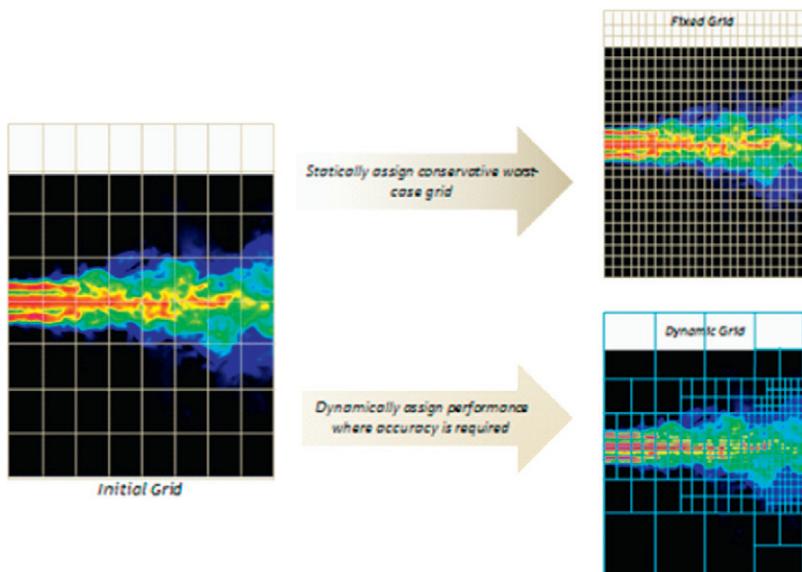
FIGURE 20.1

Fixed versus dynamic grids for a turbulence simulation model.

as we discussed in Chapter 12. To achieve the desired accuracy, such a fixed-grid approach, as illustrated in Figure 20.1, typically needs to accommodate the most demanding parts of the model and perform unnecessary extra work in parts that do not require as much detail.

A more desirable approach is shown as the dynamic grid in the lower right portion of Figure 20.1. As the simulation algorithm detects fast-changing simulation quantities in some areas of the model, it refines the grid in those areas to achieve the desired level of accuracy. Such refinement does not need to be done for the areas that do not exhibit such intensive activity. This way, the algorithm can dynamically direct more computation work to the areas of the model that benefit from the additional work.

Figure 20.2 shows a conceptual comparison between the original CUDA and the dynamic parallelism version with respect to the simulation model in Figure 20.1. Without dynamic parallelism, the host code must launch all kernels. If new work is discovered, such as refining the grid of an area of the model during the execution of a kernel, it needs to report back to the host code and have the host code to launch a new kernel. This is illustrated in Figure 20.2(a), where the host launches a wave of kernels, receives information from these kernels, and launches the next level of kernels for any new work discovered by the completed kernels.

Figure 20.2(b) shows that with dynamic parallelism, the threads that discover new work can just go ahead and launch kernels to do the work. In our example, when a thread discovers that an area of the model needs

(a) Without Dynamic Parallelism   (b) With Dynamic Parallelism
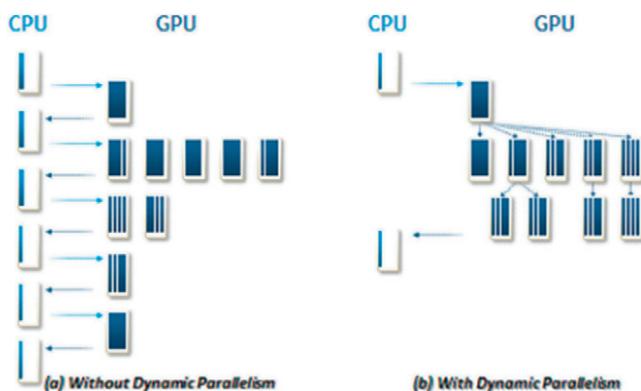
FIGURE 20.2

Kernel launch patterns for algorithms with dynamic work variation: (a) without dynamic parallelism and (b) with dynamic parallelism.

to be refined, it can launch a kernel to perform the computation step on the refined grid area without the overhead of terminating the kernel, reporting back to the host, and having the host to launch new kernels.

20.2 DYNAMIC PARALLELISM OVERVIEW

From the programmer's perspective dynamic parallelism means that he or she can write a kernel launch statement in a kernel. In Figure 20.3, the main function (host code) launches three kernels, A, B, and C. These are kernel launches in the original CUDA model. What is different is that one of the kernels, B, launches three kernels X, Y, and Z. This would have been illegal in previous CUDA systems.

The syntax for launching a kernel from a kernel is the same as that for launching a kernel from host code:

```
kernel_name <<< Dg, Db, Ns, S >>> ([kernel arguments])
```

- Dg is of type dim3 and specifies the dimensions and size of the grid.
- Db is of type dim3 and specifies the dimensions and size of each thread block.
- Ns is of type size_t and specifies the number of bytes of shared memory that are dynamically allocated per thread block for this call, which is in addition to the statically allocated shared memory. Ns is an optional argument that defaults to 0.
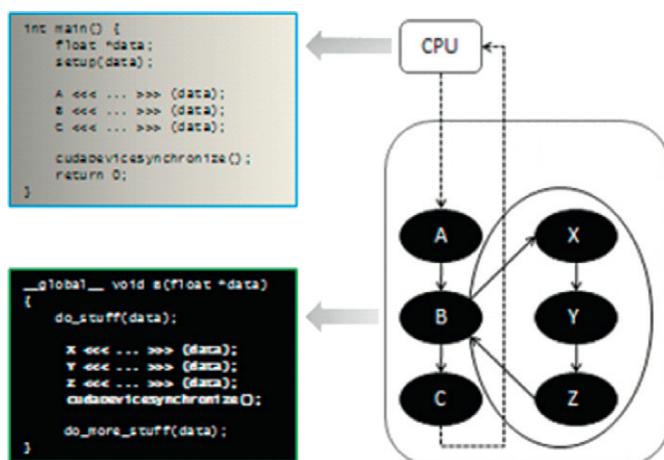
FIGURE 20.3

A simple example of a kernel (B) launching three kernels (X, Y, and Z).

- S is of type cudaStream_t and specifies the stream associated with this call. The stream must have been allocated in the same thread block where the call is being made. S is an optional argument that defaults to 0.

20.3 IMPORTANT DETAILS

Although the syntax for launching a kernel from a kernel is similar to that for launching a kernel from the host code, there are several important differences that must be clearly understood by programmers.

Launch Environment Configuration

All device configuration settings (e.g., shared memory and L1 cache size as returned from cudaDeviceGetCacheConfig(), and device limits as returned from cudaDeviceGetLimit()) will be inherited from the parent. That is, if the parent is configured for 16 K bytes of shared memory and 48 K bytes of L1 cache, then the child's execution settings will be configured identically. Likewise, a parent's device limits such as stack size will be passed as-is to its children.

API Errors and Launch Failures

Like CUDA API function calls in host code, any CUDA API function called within a kernel may return an error code. The last error code returned is recorded and may be retrieved via the cudaGetLastError() call. Errors are recorded on a per-thread basis, so that each thread can identify the most recent error that it has generated. The error code is of type cudaError_t, which is a 32-bit integer value.

Events

Only the interstream synchronization capabilities of CUDA events are supported in kernel functions. Events within individual streams are currently not supported in kernel functions. This means cudaStreamWaitEvent() is supported, but cudaEventSynchronize(), timing with cudaEventElapsedTime(), and event query via cudaEventQuery() are not. *These may be supported in a future version.*

To ensure that this restriction is clearly seen by the user, dynamic parallelism cudaEvents must be created via cudaEventCreateWithFlags(),

which currently only accepts the `cudaEventDisableTiming` flag value when called from a kernel.

Event objects may be shared between the threads within the CUDA thread-block that created them, but are local to that block and should not be passed to child/parent kernels. Event handles are not guaranteed unique between blocks, so using an event handle within a block that did not allocate it will result in undefined behavior.

Streams
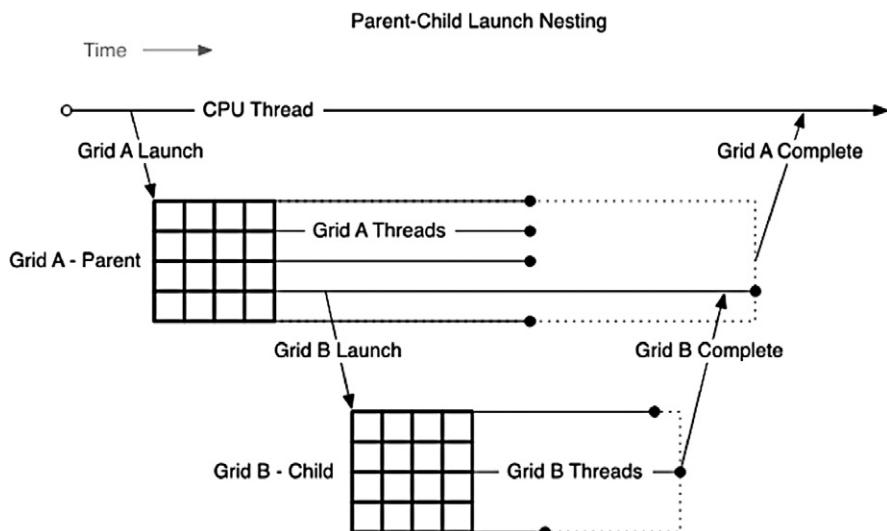
Both named and unnamed (NULL) streams are available under dynamic parallelism. Named streams may be used by any thread within a thread block, but stream handles should not be passed to other blocks or child/ parent kernels. In other words, a stream should be treated as private to the block in which it is created. Stream handles are not guaranteed to be unique between blocks, so using a stream handle within a block that did not allocate it will result in undefined behavior.

Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that require concurrency between child kernels are ill-formed and will have undefined behavior.

The host-side NULL stream's global synchronization semantic is not supported under dynamic parallelism. To explicitly indicate this behavior change all streams must be created using the `cudaStreamCreateWithFlags()` API with the `cudaStreamNonBlocking` flag in a kernel. Calls to `cudaStreamCreate()` will fail with a compiler "unrecognized function call" error, so as to make clear the different stream semantic under dynamic parallelism.

The `cudaStreamSynchronize()` API is not available within a kernel; only `cudaDeviceSynchronize()` can be used to wait explicitly for launched work to complete. This is because the underlying system software implements only a block-wide synchronization call, and it is undesirable to offer an API with incomplete semantics (i.e., the synchronize guarantees one stream synchronizes, but coincidentally provides a full barrier as a side effect).

A thread that is part of an executing grid and configures and launches a new grid belongs to the parent grid, and the grid created by the launch is the child grid. As shown in Figure 20.4, the creation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed.

Parent-Child Launch Nesting

FIGURE 20.4

Completion sequence for parent and child grids.

Even if the parent threads do not explicitly synchronize on the child grids launched, the runtime guarantees an implicit synchronization between the parent and child by forcing the parent to wait for all its children to exit execution before it can exit execution.

Synchronization Scope

A thread in the parent grid may only perform synchronization on the grids launched by that thread (e.g., using cudaDeviceSynchronize()), other threads in the thread block (e.g., using __synchthreads()), or on streams created within the same thread block (e.g., using cudaStreamWaitEvent()). Streams created by a thread within a grid exist only within the thread's thread block scope and have undefined behavior when used outside of the thread block where they were created. Streams created within a thread block are implicitly synchronized when all threads in the thread block exit execution. The behavior of operations on a stream that has been modified outside of the thread block scope is undefined. Streams created on the host have undefined behavior when used within any kernel, just as streams created by a parent grid have undefined behavior if used within a child grid.

20.4 MEMORY VISIBILITY

Global Memory

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There are two points in the execution of a child grid when its view of memory is fully consistent with the parent thread: (1) when the child grid is created by the parent, and (2) when the child grid completes as signaled by a synchronization API call in the parent thread.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. All memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion.

Zero-Copy Memory

Zero-copy system memory has identical coherence and consistency guarantees as global memory, and follows the semantics just detailed. A kernel may not allocate or free zero-copy memory, however, but may use pointers passed in from the host code.

Constant Memory

Constants are immutable and may not be written to by a kernel, even between dynamic parallelism kernel launches. That is, the value of all `__constant__` variables must be set from the host prior to launch of the first kernel. Constant memory variables are globally visible to all kernels, and so must remain constant for the lifetime of the dynamic parallelism launch tree invoked by the host code.

Taking the address of a constant memory object from within a thread has the same semantics as for non-dynamic parallelism programs, and passing that pointer from parent to child or from a child to parent is fully supported.

Local Memory

Local memory is private storage for a thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child will be undefined. For example, the

following is illegal, with undefined behavior if `x_array` is accessed by `child_launch`:

```
int x_array[10];  // Creates x_array in parent's local memory
child_launch<< < 1, 1 >> >(x_array);
```

It is sometimes difficult for a programmer to be aware of when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with `malloc()` or `new()` or by declaring `__device__` storage at the global scope. For example, Figure 20.5(a) shows a valid kernel launch where a pointer to a global memory variable is passed as an argument into the child kernel. Figure 20.5(b) shows an invalid code where a pointer to a local memory (register) variable is passed into the child kernel.

The NVIDIA compiler will issue a warning if it detects that a pointer to local memory is being passed as an argument to a kernel launch. However, such detections are not guaranteed.

Shared Memory
Shared memory is private storage for an executing thread block, and data is not visible outside of that thread block. Passing a pointer to shared memory to a child kernel either through memory or as an argument will result in undefined behavior.

Texture Memory
Texture memory accesses (read only) are performed on a memory region that may be aliased to the global memory region that is writable. Coherence for texture memory is enforced at the invocation of a child grid and when a child

```
__device__ int value;

__device__ void x() {

   value = 5;

   child<<< 1, 1 >>>(&value);

}
```

```
__device__ void y() {

   int value = 5;

   child<<< 1, 1 >>>(&value);

}
```

(a) Valid – "value" is global storage

(b) Invalid– "value" is local storage

FIGURE 20.5

Passing a pointer as an argument to a child kernel: (a) valid (`value` is global storage) and (b) invalid (`value` is local storage).

grid completes. This means that writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Also, writes to memory by a child will be reflected in the texture memory accesses by a parent, after the parent synchronizes on the child's completion.

Concurrent texture memory access and writes to global memory objects that alias the texture memory objects between a parent and its children or between multiple children will result in undefined behavior.

20.5 A SIMPLE EXAMPLE

In this section, we provide a simple example of coding in each of two styles—first in the original CUDA style, and second in the dynamic parallelism style. The example problem is extracted from the divergent phase of a hypothetical parallel algorithm. It does not compute useful results but provides a conceptually simple calculation that can be easily verified. It serves to illustrate the difference between the two styles and how one can use the dynamic parallelism style to reduce control flow divergence when the amount of work done by each thread in an algorithm can vary dynamically.

Line 22 of Figure 20.6 shows the host code main function for the example coded without dynamic parallelism. It allocates the foo variable on the device (line 25) and initializes it to 0 (line 26). It then launches the diverge_cta() kernel to perform a calculation on foo (line 27). The kernel is launched with a grid of K (set to 2 in line 5) blocks of $32 \times M$ (M set to 32 in line 4) threads each. Therefore, in this example, we are launching two blocks of 1,024 threads each.

In the diverge_cta() kernel, threads of which the threadIdx.x values are not a multiple of 32 will return immediately. In our example, only the threads with threadIdx.x values of 0, 32, 64, ..., 960, 992 will continue to execute. In line 16, all remaining M threads of each block will call the entry() function, which will increment the foo variable N (set to 128 in line 3) times. This is done by the for loop in line 8. The atomic operation in line 9 is necessary because there are multiple blocks calling the entry() function at the same time. The atomic operation ensures that increments by one of the blocks are not trampled by those of other blocks. In our case, the atomic operation ensures that all increments by both thread blocks are properly reflected in the variable foo.

After all blocks have completed their increments, the value of foo should be $K \times M \times N$, since there are K blocks and each block has M active threads each incrementing the foo variable N times. In line 17, thread 0 of each block initializes a shared memory variable x (declared in line 13) to value 5, which

```
1.   #include <stdio.h>
2.   #include <cuda.h>

3.   #define N 128
4.   #define M 32
5.   #define K 2

6.   __device__ volatile int vint = 0;

7.   __device__ void entry( volatile int* foo )
     {
8.     for (int i = 0; i < N; ++i) {
9.          atomicAdd((int*)foo, 1);
       }
     }
10.  extern "C"
11.  __global__ void
12.  diverge_cta( volatile int *foo )
{
13.      __shared__ int x;
14.      if ((threadIdx.x%32) != 0) {
15.      return;
           }
16.      entry(foo);

17.      if (threadIdx.x == 0) {
18.        x = 5;
19.        return;
       }
20.       __syncthreads();

21.     atomicAdd((int*)foo, x);
     }

22. int main( int argc, char **argv )
{
23.     int *foo;
24.     int h_foo;

25.     cudaMalloc((void**)&foo, sizeof(int));
26.     cudaMemset(foo, 0, sizeof(int));
27.     printf("foo addr: 0x%x\n", (unsigned)(size_t)foo);

28.     diverge_cta<<<K,M*32>>>( foo );
29.     cudaDeviceSynchronize();
30.     cudaMemcpy(&h_foo, foo, sizeof(int), cudaMemcpyDe viceToHost);
31.     if (h_foo == K*(M*N+5*(M-1))) {
32.         printf("simple_scan_test test PASSED\n");
33.     }
34.     else {
35.         printf("Result: %d\n", h_foo);
36.         printf("simple_scan_test test FAILED\n");
37.      }

38.      return 0;
39. }
```

FIGURE 20.6

A simple example of the divergent phase of a hypothetical parallel algorithm
coded in CUDA without dynamic parallelism.

is visible to all threads in the same block. Thread 0 then terminates. After barrier synchronization (line 20), all remaining `M-1` threads in each block will perform an atomic operation on variable `foo` (line 21). The increment amount of the atomic operation is the value of `x` (5). Since there are only `M-1` threads executing (all of which the `threadIdx.x` values are multiples of 32), all threads in a block should jointly add `5*(M-1)` to the value of `foo`. With a total of `K` blocks in the grid, the total contribution due to line 21 among all blocks is `K*(5*(M-1))`.

After the kernel terminates (line 29), the host copies the value of `foo` into its variable `h_foo` (line 30). The host then performs a test and checks if the total value in `h_foo` is the expected value of `K*N*M + K*(5*(M-1))`, which is `K*(N*M + 5*(M-1))` (line 31).

Figure 20.7 shows a version of the source code based on dynamic parallelism. The main function is identical to that of Figure 20.6 and is not shown. Also, we only assign line numbers to the lines that are different from Figure 20.6. In this version, instead of having thread 0 of each block to call the device function `entry()`, we will have each of them to launch `entry()` as a kernel. In line 2, the device function `entry()` in Figure 20.6 is now declared as a kernel function.

In line 3, the `diverge_cta()` kernel launches the `entry()` kernel with only one block, which contains the `M` thread. `K` (set to 2) kernel launches are done. In our example, one is launched by thread 0 of block 0 and one by thread 0 of block 1. Instead of having each of the remaining `M` threads of a block to call `entry()` as a device function, we use thread 0 of each block to launch `entry()` as a kernel with `M` threads.

Note that the effect on the `foo` value remains the same. The `entry()` kernel is launched `K` times. For each launch, there are `M` threads executing the `entry()` kernel, and each thread increments the `foo` value by `N`. Therefore, the total changes due to all threads are `K*M*N`. However, amount of divergence changes. The original kernel still has divergence. However, the increments are now done by the `entry()` kernel where all neighboring threads are taking the same control flow path. The amount of time the code spends in control-divergent execution decreases.

20.6 RUNTIME LIMITATIONS
Memory Footprint

Memory is allocated as the backing-store for the parent kernel state to be used when synchronizing on a child launch. Conservatively, this memory

```
                #include <stdio.h>
                #include <cuda.h>
                #include <cuos.h>

                #define N 100
                #define M 32
                #define K 2

                __device__ volatile int vint = 0;

        1.   __global__ void
                entry( volatile int* foo )
                {
                  for (int i = 0; i < N; ++i) {
                    atomicAdd((int*)foo, 1);
                  }
                }

                extern "C"
                __global__ void
                diverge_cta( volatile int *foo )
                {
                  __shared__ int x;
                  if ((threadIdx.x%32) != 0) {
                      return;
                  }
                  if (threadIdx.x == 0) {
        2.          entry<<<1,M>>>( foo );
        3.          cudaDeviceSynchronize();
                      x = 5;
                      return;
                  }
                   __syncthreads();

                  atomicAdd((int*)foo, x);
                }
```

FIGURE 20.7

The diverge_cta() kernel revised using dynamic parallelism.

must support storing of state for the maximum number of live threads pos-
sible on the GPU. This in turn means that each level of nesting requires
∼ 150 MB of device memory in a current generation device, which will be
unavailable for program use even if it is not all consumed. The dynamic
parallelism runtime system detects if the parent exits without calling
cudaDeviceSynchronize(). In this case, the runtime does not save the
parent's state and the memory footprint required for the program will be
much less than the conservative maximum.

In addition to the thread backing-store, more memory is used by the system software, for example, to store launch queues and events. The total memory footprint of dynamic parallelism is difficult to specify exactly, but may be queried at runtime.

Nesting Depth

Under dynamic parallelism, one kernel may launch another kernel, and that kernel may launch another, and so on. Each subordinate launch is considered a new "nesting level," and the total number of levels is the "nesting depth" of the program.

The maximum nesting depth is limited in hardware to 64, but in software it may be limited to 63 or less. Practically speaking, the real limit will be the amount of memory required by the system for each new level (see the preceding "Memory Footprint" section). The number of levels to be supported must be configured before the top-level kernel is launched from the host, to guarantee successful execution of a nested program.

Memory Allocation and Lifetime

Currently, cudaMalloc and cudaFree have slightly modified semantics between the host and device environments (Table 20.1). Within the device environment the total allocatable memory is limited to the device malloc() heap size, which may be smaller than the available unused device memory. Also, it is an error to invoke cudaFree from the host program on a pointer that was allocated by cudaMalloc on the device, or to invoke cudaFree from the device program on a pointer that was allocated by cudaMalloc on the host. These limitations may be removed in a future version.

Table 20.1 Memory allocation and deallocation from host and device.		
	cudaMalloc() **on Host**	cudaMalloc() **on Device**
cudaFree() on host	Supported	Not supported
cudaFree() on device	Not supported	Supported
Allocation limit	Free device memory	cudaLimitMallocHeapSize

ECC Errors

No notification of ECC errors is available to code within a CUDA kernel. ECC errors are only reported at the host side. Any ECC errors that arise during execution of a dynamic parallelism kernel will either generate an exception or continue execution (depending on error and configuration).

Streams

Unlimited named streams are supported per block, but the maximum concurrency supported by the platform is limited. If more streams are created than can support concurrent execution, some of these may serialize or alias with each other. In addition to block-scope named streams, each thread has an unnamed (NULL) stream, but named streams will not synchronize against it (indeed, all named streams must be created with a flag explicitly preventing this).

Events

Unlimited events are supported per block, but these consume device memory. Owing to resource limitations, if too many events are created (exact number is implementation-dependent), then GPU-launched grids may attain less concurrency than might be expected. Correct execution is guaranteed, however.

Launch Pool

When a kernel is launched, all associated data is added to a slot within the launch pool, which is tracked until the kernel completes. Launch pool storage may be virtualized by the system, between device and host memory; however, device-side launch pool storage has improved performance. The amount of device memory reserved for device-side launch pool storage is configurable prior to the initial kernel launch from the host.

20.7 A MORE COMPLEX EXAMPLE

We now show an example that is a more interesting and useful case of recursive, adaptive subdivision of spline curves. This illustrates a variable amount of child kernel launches, according to the workload. The example is to calculate Bezier curves [Wiki_Bezier 2012], which are frequently

used in computer graphics to draw smooth, intuitive curves that are defined by a set of *control points*, which are typically defined by a user.

Mathematically, a Bezier curve is defined by a set of control points $\mathbf{P}_0$ through $\mathbf{P}_n$, where n is called its order ($n = 1$ for linear, 2 for quadratic, 3 for cubic, etc.). The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve.

Linear Bezier Curves

Given two control points $\mathbf{P}_0$ and $\mathbf{P}_1$, a linear Bezier curve is simply a straight line connecting between those two points. The coordinates of the points on the curve are given by the following linear interpolation formula:

$$B(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + tP_1, \; t \in [0, \, 1]$$

Quadratic Bezier Curves

A quadratic Bezier curve is defined by three control points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$. The points on a quadratic curve are defined as a linear interpolation of corresponding points on the linear Bezier curves from $\mathbf{P}_0$ to $\mathbf{P}_1$ and from $\mathbf{P}_1$ to $\mathbf{P}_2$, respectively. The calculation of the coordinates of points on the curve is expressed in the following formula:

$$B(t) = (1 - t)[(1 - t)P_0 + tP_1] + t[(1 - t)P_1 + tP_2], \; t \in [0, \, 1]$$

which can be simplified into the following formula:

$$B(t) = (1-t)^2 P_0 + 2(1 - t)t P_1 + t^2 P_2, \; t \in [0, \, 1].$$

Bezier Curve Calculation (Predynamic Parallelism)

Figure 20.8 shows a CUDA C program that calculates the coordinates of points on a Bezier curve. The first part of the code defines several operators (operator +, operator −, operator*, length) for 2D coordinates that will be used in the kernel code. They should be quite self-explanatory so we will not elaborate on them.

The main function (line 20) initializes a set of control points to random values (lines 22, 23, and 24). In a real application, these control points are

```
      #include <stdio.h>
      #include <cuda.h>

      //Some inline vector math functions
      __forceinline__ __device__ float2 operator+(float2 a, float2 b){
        float2 c;
        c.x = a.x + b.x;    c.y = a.y + b.y;
        return c;
      }

      __forceinline__ __device__ float2 operator-(float2 a, float2 b){
        float2 c;
        c.x = a.x - b.x;    c.y = a.y - b.y;
        return c;
      }

      __forceinline__ __device__ float2 operator*(float a, float2 b){
        float2 c;
        c.x = a * b.x;    c.y = a * b.y;
        return c;
      }

      __forceinline__ __device__ float length(float2 a){
        return sqrtf(a.x*a.x + a.y*a.y);
      }

      #define MAX_TESS_POINTS 32

1.    struct BezierLine //A structure containing all the parameters we
        need to tessellate a Bezier line
      {
        float2 CP[3];                     //Control points for the line
        float2 vertexPos[MAX_TESS_POINTS]; //Vertex position array to
                tessellate into
        int nVertices;                    //Number of tessellated
                vertices
      };

      __global__ void computeBezierLines(BezierLine *bLines, int nLines)
      {
        int bidx = blockIdx.x;
        if(bidx < nLines){
        //Compute the curvature of the line
2.      float curvature = length(bLines[bidx].CP[1] - 0.5f*
            (bLines[bidx].CP[0] + bLines[bidx].CP[2]))/length
            (bLines[bidx].CP[2] - bLines[bidx].CP[0]);
        //From the curvature, compute the number of tessellation points
3.      int nTessPoints = min(max((int)(curvature*16.0f),4),32);
4.      bLines[bidx].nVertices = nTessPoints;

        //Loop through the vertices to be tessellated, incrementing by
            blockDim.x
5.       for(int inc = 0; inc < nTessPoints; inc += blockDim.x){
```

FIGURE 20.8

Bezier curve calculation without dynamic parallelism.

```
6.        int idx = inc + threadIdx.x;  //Compute a unique index for
            this point
7.        if(idx < nTessPoints){
8.         float u = (float)idx/(float)(nTessPoints-1);  //Compute u
            from idx
9.         float omu = 1.0f - u;   //pre-compute one minus u

10.        float B3u[3]; //Compute quadratic Bezier coefficients
11.        B3u[0] = omu*omu;
12.        B3u[1] = 2.0f*u*omu;
13.        B3u[2] = u*u;

14.        float2 position = {0,0};  //Set position to zero
15.        for(int i = 0; i < 3; i++){
            //Add the contribution of the i'th control point to position
16.         position = position + B3u[i] * bLines[bidx].CP[i];
           }
           //Assign the value of the vertex position to the correct
           array element
17.        bLines[bidx].vertexPos[idx] = position;
         }
        }
       }
      }

18. #define N_LINES 256
19. #define BLOCK_DIM 32

20. int main( int argc, char **argv )
     {
21. BezierLine *bLines_h = new BezierLine[N_LINES]; //Allocate array
         of lines in host memory

    float2 last = {0,0};    //Set initial point to zero (last is the
         last point in the previous segment).
22.  for(int i = 0; i < N_LINES; i++){
23.   bLines_h[i].CP[0] = last; //Set first point of this line to last
         point of previous line
24.    for(int j = 1; j < 3; j++){
        bLines_h[i].CP[j].x = (float)rand()/(float)RAND_MAX;  //Assign
         random coordinate between 0 and 1
        bLines_h[i].CP[j].y = (float)rand()/(float)RAND_MAX;  //Assign
         random coordinate between 0 and 1
       }
       last = bLines_h[i].CP[2];   //keep the last point of this line
       bLines_h[i].nVertices = 0;  //Set number of tessellated
         vertices to zero
      }

25.  BezierLine *bLines_d; //Pointer to array of Bezier lines in
     device memory
26.  cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
         //Allocate device memory for array of Bezier lines
```

FIGURE 20.8

(continued)

```
27.    cudaMemcpy(bLines_d, bLines_h, N_LINES*sizeof(BezierLine),
           cudaMemcpyHostToDevice);

28.    computeBezierLines<<<N_LINES, BLOCK_DIM>>>(bLines_d, N_LINES);
           //Call the kernel to tessellate the lines

       //Do something to draw the lines here

       cudaFree(bLines_d); //Free the array of lines in device memory
```

FIGURE 20.8

(continued)

most likely inputs from a user. The control points are part of the `bLines_h` array of which the element type `BezierLine` is declared in line 1. The storage for the `bLines_h` array is allocated in line 21. The host code then allocates the corresponding device memory for the `bLines_d` array and copies the initialized data to `bLines_d` (lines 26−28). It then calls the `computeBezierLine()` kernel to calculate the coordinates of the Bezier curve.

The `computeBezierLine()` kernel is designed to use a thread block to calculate the curve points for a set of three control points (of the quadratic Bezier formula). Each thread block first computes a measure of the curvature of the curve defined by the three control points. Intuitively, the larger the curvature, the more the points it takes to draw a smooth quadratic Bezier curve for the three control points. This defines the amount of work to be done by each thread block. This is reflected in lines 3 and 4, where the total number of points to be calculated by the current thread block is proportional to the curvature value.

In the `for` loop in line 5, all threads calculate a consecutive set of Bezier curve points in each iteration. The detailed calculation in the loop body is based on the formula we presented earlier. The key point is that the number of iterations taken by threads in a block can be very different from that taken by threads in another block. Depending on the scheduling policy, such variation of the amount of work done by each thread block can result in decreased utilization of streaming multiprocessors and thus reduced performance.

Bezier Curve Calculation (with Dynamic Parallelism)

Figure 20.9 shows a Bezier curve calculation code using dynamic parallelism. It breaks the `computeBezierLine()` kernel in Figure 20.8 into two kernels. The first part, `computeBezierLineCDP()`, discovers the amount of work to be

```
1. struct BezierLine
   {
    float2 CP[3];      //Control points for the line
    float2 *vertexPos; //Vertex position array to tessellate into
    int nVertices;     //Number of tessellated vertices
   };

2. __global__ void computeBezierLinePositions(int lidx, BezierLine*
      bLines, int nTessPoints)
   {
3. int idx = threadIdx.x + blockDim.x*blockIdx.x; //Compute an
      index unique to this vertex
4. if(idx < nTessPoints){
5.   float u = (float)idx/(float)(nTessPoints-1); //Compute u from idx
     float omu = 1.0f - u;   //Pre-compute one minus u

     float B3u[3];   //Compute quadratic Bezier coefficients
     B3u[0] = omu*omu;
     B3u[1] = 2.0f*u*omu;
     B3u[2] = u*u;

     float2 position = {0,0};  //Set position to zero
     for(int i = 0; i < 3; i++){
       //Add the contribution of the i'th control point to position
        position = position + B3u[i] * bLines[lidx].CP[i];
     }

     bLines[lidx].vertexPos[idx] = position; //Assign the value of the
        vertex position to the correct array element
    }
   }
   __global__ void computeBezierLinesCDP(BezierLine *bLines, int
      nLines)
   {
6.  int lidx = threadIdx.x + blockDim.x*blockIdx.x;  //Compute a
      unique index for each Bezier line

7.  if(lidx < nLines){
      //Compute the curvature of the line
      float curvature = length(bLines[lidx].CP[1] - 0.5f*(bLines[lidx]
             .CP[0] + bLines[lidx].CP[2]))/length(bLines[lidx].CP[2]-
             bLines[lidx].CP[0]);
      //From the curvature, compute the number of tessellation points
      bLines[lidx].nVertices = min(max((int)(curvature*16.0f),4),
      MAX_TESS_POINTS);

8.    cudaMalloc((void**)&bLines[lidx].vertexPos, bLines[lidx]
      .nVertices*sizeof(float2));
      //Call the child kernel to compute the tessellated points for
      each line
9.    computeBezierLinePositions<<<ceil((float)bLines[lidx]
      .nVertices/32.0f), 32>>>(lidx, bLines,bLines[lidx].nVertices);
    }
}
```

FIGURE 20.9

Bezier calculation with dynamic parallelism.

```
    __global__ void freeVertexMem(BezierLine *bLines, int nLines)
    {
    int lidx = threadIdx.x + blockDim.x*blockIdx.x;   //Compute a unique
        index for each Bezier line
10. if(lidx < nLines)
11. cudaFree(bLines[lidx].vertexPos);   //Free the vertex memory for
        this line
    }

     #define N_LINES 256
    #define BLOCK_DIM 64

12. int main( int argc, char **argv )
    {
      BezierLine *bLines_h = new BezierLine[N_LINES]; //Allocate array
        of lines in host memory

      float2 last = {0,0};    //Set last point to zero
       for(int i = 0; i < N_LINES; i++){
         bLines_h[i].CP[0] = last;   //Set first point of this line to
                last point of previous line
          for(int j = 1; j < 3; j++){
           bLines_h[i].CP[j].x = (float)rand()/(float)RAND_MAX;  //Assign
                random coordinate between 0 and 1
           bLines_h[i].CP[j].y = (float)rand()/(float)RAND_MAX; //Assign
                random coordinate between 0 and 1
           }
      last = bLines_h[i].CP[2];   //keep the last point of this line
      bLines_h[i].vertexPos = NULL;   //Set the vertex position array
                to NULL
      bLines_h[i].nVertices = 0;  //Set number of tessellated vertices
                to zero
      }

      BezierLine *bLines_d;   //Pointer to array of Bezier lines in
        device memory
      cudaMalloc((void**)&bLines_d, N_LINES*sizeof(BezierLine));
      cudaMemcpy(bLines_d, bLines_h, N_LINES*sizeof(BezierLine),
        cudaMemcpyHostToDevice);

13. computeBezierLinesCDP<<<ceil((float)N_LINES/(float)BLOCK_DIM),
        BLOCK_DIM>>>(bLines_d, N_LINES);

      //Do something to draw the lines here

14. freeVertexMem<<<ceil((float)N_LINES/(float)BLOCK_DIM),
        BLOCK_DIM>>>(bLines_d, N_LINES);
    cudaFree(bLines_d);   //Free the array of lines in device memory
    delete[] bLines_h;    //Free the array of lines in host memory
    }
```

FIGURE 20.9

(continued)

done for each control point. The second part, computeBezierLinePositions(), performs the calculation.

With the new organization, the amount of work done for each set of control points by the computeBezierLinesCDP() kernel is much smaller than the original computeBezierLines() kernel. Therefore, we use one thread to do this work in computeBezierLinesCDP(), as opposed to using one block in computeBezierLinesPossitions(). In line 13, we only need to launch one thread per set of control points. This is reflected by dividing the N_LINES by BLOCK_DIM to form the number of blocks in the kernel launch configuration.

There are two key differences between the computeBezierLinesCDP() kernel and the computeBezierLines() kernel. First, the index used to access the control points is formed on a thread basis (line 6) rather than a block basis. This is because the work for each control point is done by a thread rather than a block as we mentioned before. Second, the memory for storing the calculated Bezier curve points is dynamically determined and allocated in line 8. This allows the code to assign just enough memory to each set of control points in the BezierLine type. Note that in Figure 20.8, each BezierLine element is declared with a maximal possible number of points. On the other hand, the declaration in Figure 20.9 has only a pointer to a dynamically allocated storage. Allowing a kernel to call the cudaMalloc() function can lead to substantial reduction of memory usage for situations where the curvature of control points varies significantly.

Once a thread of the computeBezierLinesCDP() kernel determines the amount of work needed by its set of control points, it launches the computeBezierPositions() kernel to do the work (line 9). In our example, every thread from the parent grid creates a new grid for its assigned set of control points. This way, the work done by each thread block is balanced. The amount of work done by each child grid varies.

After the computeBezierLinesCDP() kernel terminates, the main function can copy the data back and draw the curve on an output device. It can also call a kernel to free all storage allocated to the bLines_d storage in parallel (line 14). This can be faster than sequentially calling the cudaFree() function in a loop.

20.8 SUMMARY

CUDA dynamic parallelism extends the CUDA programming model to allow kernels to launch kernels. This allows each thread to dynamically

discover work and launch new grids according to the amount of work. It also supports dynamic allocation of device memory by threads. As we show in the Bezier curve calculation example, these extensions can lead to better work balance across threads and blocks as well as more efficient memory usage.

Reference

Bezier Curves, Available at: <*http://en.wikipedia.org/wiki/B%C3%A9zier_curve*>, 2012.

This page intentionally left blank

Conclusion and Future Outlook

21

CHAPTER OUTLINE

You made it! We have arrived at the finishing line. In this final chapter, we will briefly review the goals that we have achieved through this book. Instead of drawing a conclusion, we will offer our vision for the future evolution of massively parallel processor architectures and how the advancements will impact parallel application development.

21.1 GOALS REVISITED

As we stated in Chapter 1, our primary goal is to teach you, the readers, how to program massively parallel processors. We promised that it would become easy once you develop the right insight and go about it the right way. In particular, we promised to focus on *computational thinking* skills that would enable you to think about problems in ways that are amenable to parallel computing.

We delivered on these promises through an introduction to performance considerations for CUDA (Chapter 6), three parallel patterns (Chapters 8, 9, and 10), two detailed application case studies (Chapters 11 and 12), and a chapter dedicated to computational thinking skills (Chapter 13). Through this process, we introduced the pertinent computer

architecture knowledge needed to understand the hardware limitations that must be addressed in high-performance parallel programming. In particular, we focused on the memory bandwidth limitations that will remain as the primary performance limiting factor in massively parallel computing systems (Chapters 4, 5, 6, 8, 9, 10, 11, 12, and 13). We also introduced the concept of floating-point precision/accuracy and numerical stability, and how they relate to parallel algorithms (Chapter 7). With these insights, high-performance parallel programming becomes a manageable process, rather than a black art.

We stated that our second goal was to teach high-performance parallel programming styles that naturally avoid subtle correctness issues. To deliver on this promise, we showed that the simple data-parallel CUDA programming model (Chapters 3 and 4) based on barrier synchronization can be used to develop very high-performance applications. This disciplined way of parallel programming naturally avoids the subtle race conditions that plague many other parallel programming systems.

We promised to teach parallel programming styles that transparently scale across future hardware generations, which will be more and more parallel. With the CUDA threading model (Chapter 4), a massive number of thread blocks can be executed in any order relative to each other. Your application will be able to benefit from more parallel hardware coming in the future. We also presented algorithm techniques, such as tiling and cutoff, that allow your application to scale naturally to very large data sets (Chapters 8, 9, 10, 11, 12, and 13).

We promised to teach the programming skills in such a way that you will be able to apply them to other programming models and languages. To help you branch out to other programming models, we introduced OpenCL (Chapter 14), OpenACC (Chapter 15), Thrust (Chapter 16), CUDA FORTRAN (Chapter 17), C++ AMD (Chapter 18), and MPI-CUDA (Chapter 19). In each chapter, we explained how the programming model/language relates to CUDA and how you can apply the skills you learned based on CUDA to these models/languages.

We hope that you have enjoyed the book.

Now that we have reviewed our promises, we would like to share our view of the coming evolution of the massively parallel processor architectures and how the advancements will likely impact application development. We hope that these outlooks will help you to peek into the future of parallel programming. Our comments are based on the new features in GPUs based on NVIDIA's Kepler compute architecture that arrived at the market when this book went into press.

21.2 MEMORY MODEL EVOLUTION

Large virtual and physical address spaces. GPUs have traditionally used only a physical address space with up to 32 address bits, which limited the GPU DRAM to 4 gigabytes or less. This is because graphics applications have not demanded more than a few hundred megabytes of frame buffer and texture memory. This is in contrast to the 64-bit virtual space and 40 + bits of physical space that CPU programmers have been taking for granted for many years. However, more recent graphics applications have demanded more.

More recent GPU families such as Fermi and Kepler have adopted CPU-style virtual memory architecture with a 64-bit virtual address space and a physical address space of at least 40 bits. The obvious benefit is that Fermi and Kepler GPUs can incorporate more than 4 gigabytes of DRAM and that CUDA kernels can now operate on very large data sets, whether hosted entirely in on-board GPU DRAM, or by accessing mapped host memory.

The Fermi virtual memory architecture also lays the foundation for a potentially profound enhancement to the programming model. The CPU system physical memory and the GPU physical memory can now be mapped within a single, shared virtual address space [GNS 2009]. A shared global address space allows all variables in an application to have unique addresses. Such memory architecture, when exposed by programming tools and a runtime system to applications, can result in several major benefits.

First, new runtime systems can be designed to allow CPUs and GPUs to access the entire volume of application data under traditional protection models. Such a capability would allow applications to use a single pointer system to access application variables, removing a confusing aspect of the current CUDA programming model where developers must not dereference a pointer to the device memory in host functions.

These variables can reside in the CPU physical memory, the GPU physical memory, or even both. The runtime and hardware can implement data migration and coherence support like the GMAC system [GNS 2009]. If a CPU function dereferences a pointer and accesses a variable mapped to the GPU physical memory, the data access would still be serviced, but perhaps at a longer latency. Such capability would allow the CUDA programs to more easily call legacy libraries that have not been ported to GPUs. In the current CUDA memory architecture, the developer must manually transfer data from the device memory to the host memory to use legacy library functions to process them on the CPU. GMAC is built on a current CUDA

runtime API and gives the developer the option to either rely on the runtime system to service such accesses or to manually transfer data as a performance optimization. However, the GMAC system currently does not have a clean mechanism for supporting multiple GPUs. The new virtual memory capability would enable a much more elegant implementation.

Ultimately, the virtual memory capability will also enable a mechanism similar to the zero-copy feature in CUDA 2.2 to allow the GPU to directly access very large physical CPU system memories. In some application areas such as CAD, the CPU physical memory system may have hundreds of gigabytes of capacity. These physical memory systems are needed because the applications require the entire data set to be "in core." It is currently infeasible for such applications to take advantage of GPU computing. With the ability to directly access very large CPU physical memories, it becomes feasible for GPUs to accelerate these applications.

The second potential benefit is that the shared global address space enables peer-to-peer direct data transfer between devices in a multidevice system. This is supported in CUDA 4.0 and later, using the GPUDirect™ feature. In older CUDA systems, devices must first transfer data to the host memory before delivering them to a peer device. A shared global address space enables the implementation of a runtime system to provide an API to directly transfer data from one device memory to another device memory. Ultimately, a runtime system can be designed to automate such transfers when devices reference data in each other's memory, but still allow the use of explicit data transfer APIs as a performance optimization. In CUDA 5.0, it is possible not only to reference data on other GPUs within a multi-GPU system, but also data on GPUs on other local systems.

The third benefit is that one can implement I/O-related memory transfers directly in and out of the device memory. In older CUDA systems, I/O input data must first be transferred into the host memory before it can be copied into the device memory. The ability to directly transfer data in and out of the device memory can significantly reduce the copying cost and enhance the performance of applications that process large data sets.

Unified device memory space. In early CUDA memory models, constant memory, shared memory, local memory, and global memory form their own separate address spaces. The developer can use pointers into the global memory but not others. Starting with the Fermi architecture, these memories are parts of a unified address space. This makes it easier to abstract which memory contains a particular operand, allowing the programmer to deal with this only during allocation, and making it simpler to pass CUDA data objects into other procedures and functions, irrespective

of which memory area they come from. It makes CUDA code modules much more "composable." That is, a CUDA device function can now accept a pointer that may point to any of these memories. The code would run faster if a function argument pointer points to a shared memory location and slower if it points to a global memory location. The programmer can still perform manual data placement and transfers as a performance optimization. This capability will significantly reduce the cost of building production-quality CUDA libraries.

Configurable caching and scratchpad. The shared memory in early CUDA systems served as programmer-managed scratch memory and increased the speed of applications where key data structures have localized, predictable access patterns. Starting with the Fermi architecture, the shared memory has been enhanced to a larger on-chip memory that can be configured to be partially cache memory and partially shared memory, which allows coverage of both predictable and less predictable access patterns to benefit from on-chip memory. This configurability allows programmers to apportion the resources according to the best fit for their application.

Applications in an early design stage that are ported directly from CPU code will benefit greatly from caching as the dominant part of the on-chip memory. This would further smooth the performance tuning process by increasing the level of "easy performance" when a developer ports a CPU application to a GPU.

Existing CUDA applications and those that have predictable access patterns will have the ability to increase their use of fast shared memory by a factor of three while retaining the same device "occupancy" they had on previous generation devices. For CUDA applications of which the performance or capabilities are limited by the size of the shared memory, the three times increase in size will be a welcome improvement. For example, in stencil computation such as finite volume methods for computational fluid dynamics, the state loaded into the shared memory also includes "halo" elements from neighboring areas.

The relative portion of halo decreases as the size of the stencil increases. In 3D simulation models, the halo cells can be comparable in data size as the main data for current shared memory sizes. This can significantly reduce the effectiveness of the shared memory due to the significant portion of the memory bandwidth spent on loading the halo elements. For example, if the shared memory allows a thread block to load an 8^3 ($= 512$) cell stencil into the shared memory, with one layer of halo elements on every surface, only 6^3 ($= 216$), or less than half of the loaded

cells, are the main data. The bandwidth spent on loading the halo elements is actually bigger than that spent on the main data. A three times increase in shared memory size allows some of these applications to have a more favorable stencil size where the halo accounts for a much lesser portion of the data in shared memory. In our example, the increased size would allow a 11^3 ($= 1,331$) tile to be loaded by each thread block. With one layer of halo elements on each surface, a total of 9^3 ($= 729$) cells, or more than half of the loaded elements, are main data. This significantly improves the memory bandwidth efficiency, and the performance of the application.

Enhanced atomic operations. The atomic operations in Fermi are much faster than those in previous CUDA systems, and the atomic operations in Kepler are still faster. In addition, the Kepler atomic operations are more general. Atomic operations are frequently used in random scatter computation patterns such as histograms. Faster atomic operations reduce the need for algorithm transformations such as prefix sum (Chapter 9) [SHZ 2007] and sorting [SHG 2009] for implementing such random scattering computations. These transformations tend to increase the number of kernel invocations needed to perform the target computation. Faster atomic operations can also reduce the need for involvement of the host CPU in algorithms that do collective operations or where multiple thread blocks update shared data structures, and thus reduce the data transfer pressure between the CPU and the GPU.

Enhanced global memory access. The speed of random memory access is much faster in Fermi and Kepler than earlier CUDA systems. Programmers can be less concerned about memory coalescing. This allows more CPU algorithms to be directly used in the GPU as an acceptable base, further smoothing the path of porting applications that access a diversity of data structures such as ray tracing, and other applications that are heavily object-oriented and may be difficult to convert into perfectly tiled arrays.

21.3 KERNEL EXECUTION CONTROL EVOLUTION

Function calls within kernel functions. Previous CUDA versions did not allow function calls in kernel code. Although the source code of kernel functions can appear to have function calls, the compiler must be able to inline all function bodies into the kernel object so that there is no function calls in the kernel function at runtime. Although this model works reasonably well for performance-critical portions of many applications, it does not support the software engineering practices in more sophisticated

applications. In particular, it does not support system calls, dynamically linked library calls, recursive function calls, and virtual functions in object-oriented languages such as C++.

More recent device architectures such as Kepler support function calls in kernel functions at runtime. This feature is supported in CUDA 5.0 and later. The compiler is no longer required to inline the function bodies. It can still do so as a performance optimization. This capability is partly enabled by cached, fast implementation of massively parallel call frame stacks for CUDA threads. It makes CUDA device code much more "composable" by allowing different authors to write different CUDA kernel components and assemble them all together without heavy redesign costs. In particular, it allows modern object-oriented techniques such as virtual function calls, and software engineering practices such as dynamically linked libraries. It also allows software vendors to release device libraries without source code for intellectual property protection.

Support for function calls at runtime allows recursion and will significantly ease the burden on programmers as they transition from legacy CPU-oriented algorithms toward GPU-tuned approaches for divide-and-conquer types of computation. This also allows easier implementation of graph algorithms where data structure traversal often naturally involves recursion. In some cases, developers will be able to "cut and paste" CPU algorithms into a CUDA kernel and obtain a reasonably performing kernel, although continued performance tuning would still add benefit.

Exception handling in kernel functions. Early CUDA systems did not support exception handling in kernel code. While not a significant limitation for performance-critical portions of many high-performance applications, it often incurs software engineering costs in production-quality applications that rely on exceptions to detect and handle rare conditions without executing code to explicitly test for such conditions. Also, it does not allow kernel functions to utilize operating system services, which is typically avoided in performance-critical portions of the applications except during debugging situations.

With the availability of exception handling and function call support, kernels can now call standard library functions such as `printf()` and `malloc()`, which can lead to system call traps. In our experience, the ability to call `printf()` in the kernel provides a subtle but important aid in debugging and supporting kernels in production software. Many end users are nontechnical and cannot be easily trained to run debuggers to provide developers with more details on what happened before a crash. The ability to execute `printf()` in the kernel allows the developers to add a mode to

the application to dump the internal state so that the end users can submit meaningful bug reports.

Simultaneous execution of multiple kernels. Previous CUDA systems allow only one kernel to execute on each GPU device at any point in time. Multiple kernel functions can be submitted for execution. However, they are buffered in a queue that releases the next kernel after the current one completes execution. Fermi and its successors allow multiple kernels from the same application to be executed simultaneously, which reduces the pressure for the application developer to "batch" multiple kernels into a larger kernel to more fully utilize a device. A typical example of benefit is for parallel cluster applications that segment work into "local" and "remote" partitions, where remote work is involved in interactions with other nodes and resides on the critical path of global progress. In previous CUDA systems, kernels needed to be large to keep the device running efficiently, and one had to be careful not to launch local work such that global work could be blocked. This meant choosing between underutilizing the device while waiting for remote work to arrive, or eagerly starting on local work to keep the device productive at the cost of increased latency for completing remote work units. With multiple kernel execution, the application can use much smaller kernel sizes for launching work, and as a result when high-priority remote work arrives, it can start running with low latency instead of being stuck behind a large kernel of local computation.

In Kepler and CUDA 5.0, the multiple kernel launch facility is extended by the addition of multiple hardware queues, which allow much more efficient scheduling of blocks from multiple kernels including kernels in multiple streams. In addition, the CUDA dynamic parallelism feature allows GPU work creation: GPU kernels can launch child kernels, asynchronously, dynamically, and in a data-dependent or compute load-dependent fashion. This reduces CPU−GPU interaction and synchronization, since the GPU can now manage more complex workloads independently. The CPU is in turn free to perform other useful computation.

Interruptable kernels. Fermi allows the running kernel to be "canceled," which eases the creation of CUDA-accelerated applications that allow the user to abort a long-running calculation at any time, without requiring significant design effort on the part of the programmer. Once software support is available, this will enable implementation of user-level task scheduling systems that can better perform load balance between GPU nodes of a computing system, and allows more graceful handling of cases where one GPU is heavily loaded and may be running slower than its peers [SH 2009].

21.4 **CORE PERFORMANCE**

Double-precision speed. Early devices perform double-precision floating-point arithmetic with significant speed reduction (around eight times slower) compared to single precision. The floating-point arithmetic units of Fermi and its successors have been significantly strengthened to perform double-precision arithmetic at about half the speed of single precision. Applications that are intensive in double-precision floating-point arithmetic benefit tremendously. Other applications that use double precision carefully and sparingly see less performance impact.

In practice, the most significant benefit will likely be obtained by developers who are porting CPU-based numerical applications to GPUs. With the improved double-precision speed, they will have little incentive to spend the effort to evaluate whether their applications or portions of their applications can fit into single precision. This can significantly reduce the development cost for porting CPU applications to GPUs, and addresses a major criticism of GPUs by the high-performance computing community. Some applications that are operating on smaller size input data (8 bits, 16 bits, or single-precision floating point) may continue to benefit from using single-precision arithmetic, due to the reduced bandwidth of using 32-bit versus 64-bit data. Applications such as medical imaging, remote sensing, radio astronomy, seismic analysis, and other natural data frequently fit into this category.

Better control flow efficiency. Fermi adopts a general compiler-driven predication technique [MHM1995] that can more effectively handle control flow than previous CUDA systems. While this technique was moderately successful in VLIW systems, it can provide more dramatic speed improvements in GPU warp-style SIMD execution systems. This capability can potentially broaden the range of applications that can take advantage of GPUs. In particular, major performance benefits can potentially be realized for applications that are very data-driven, such as ray tracing, quantum chemistry visualization [SSH2009], and cellular automata simulation.

21.5 **PROGRAMMING ENVIRONMENT**

Future CUDA compilers will include enhanced support for C++ templates and virtual function calls in kernel functions. Although the hardware enhancements, such as the ability to make function calls at runtime, are in place, enhanced C++ language support in the compiler has been taking

more time. The C++ try/catch features will also likely be fully supported in kernel functions in the near future. With these enhancements, future CUDA compilers will support most mainstream C++ features. The remaining features in kernel functions such as new, delete, constructors, and destructors will likely be available in later compiler releases.

New and evolved programming interfaces will continue to improve the productivity of heterogeneous parallel programmers. As we showed in Chapter 15, OpenACC allows developers to annotate their sequential loops with compiler directives to enable a compiler to generate CUDA kernels. In Chapter 16, we show that one can use the Thrust library of parallel type-generic functions, classes, and iterators to describe their computation and have the underlying mechanism to generate and configure the kernels that implement the computation. In Chapter 17, we presented CUDA FORTRAN that allows FORTRAN programmers to develop CUDA kernels in their familiar language. In particular, this interface offers strong support for indexing into multidimensional arrays. In Chapter 18, we gave an overview of the C++ AMP interface that allow the developers to describe their kernels as parallel loops that operate on logical data structures, such as multidimensional arrays in a C++ application. We fully expect that new innovations will continue to arise to further boost the productivity of developers in this exciting area.

21.6 FUTURE OUTLOOK

The new CUDA 5.0 SDK and the new GPUs based on the Kepler architecture mark the beginning of the fourth generation of GPU computing that places real emphasis on support for developer productivity and modern software engineering practices. With the new capabilities, the range of applications that will be able to get reasonable performance at minimal development cost will expand significantly. We expect that developers will immediately notice the reduction in application development, porting, and maintenance cost compared to previous CUDA systems. The existing applications developed with Thrust and similar high-level tools that automatically generate CUDA code will also likely get an immediate boost in their performance. While the benefit of hardware enhancements in memory architecture, kernel execution control, and compute core performance will be visible in the associated SDK release, the true potential of these enhancements may take years to be fully exploited in the SDKs and runtimes. For example, the true potential of the hardware virtual memory

capability will likely be fully achieved only when a shared global address space runtime that supports direct GPU I/O and peer-to-peer data transfer for multi-GPU systems becomes widely available. We predict an exciting time for innovations from both industry and academia in programming tools and runtime environments for massively parallel computing in the next few years.

Enjoy the ride!

References

Gelado, I., Navarro, N., Stone, J., Patel, S., & Hwu, W. W. (2009). An asymmetric distributed shared memory model for heterogeneous parallel systems, Technical Report, IMPACT Group, University of Illinois, Urbana-Champaign.

Mahlke, S. A., Hank, R. E., MCormick, J. E., August, D. I., & Hwu, W. W. (June 1995). A comparison of full and partial predicated execution support for ILP processors, Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, pp. 138−150.

Stone, J. E., & Hwu, W. W. (2009).WorkForce: A Lightweight Framework for Managing Multi-GPU Computations, Technical Report, IMPACT Group, University of Illinois, Urbana-Champaign.

Satish, N., Harris, M., & Garland. M. (May 2009). Designing efficient sorting algorithms for many core GPUs, Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, Rome, Italy, pp. 177-187.

Sengupta, S., Harris, M., Zhang, Y., & Owens, J. D. (Aug. 2007). Scan Primitives for GPU computing, Proceedings of Graphics Hardware 2007, San Diego, California, pp. 97−106.

Stone, J. E., Saam, J., Hardy, D. J., Vandivort, K. L., Hwu, W. W., & Schulten, K. (March 8, 2009). High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs, the second GPGPU workshop, ACM/IEEE Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS), pp. 9−18.

This page intentionally left blank

Matrix Multiplication Host-Only Version Source Code

A

APPENDIX OUTLINE

This appendix shows a host-only source code that can be used as the base of your CUDA matrix multiplication code. We have already inserted timer calls in key places so that you can use the measurement to isolate the execution time of the function that actually performs the matrix multiplication. It also has the code that you can use to print out the matrix contents and verify the results.

A.1 matrixmul.cu

```
/*****************************************************************************
  File Name  [matrixmul.cu]
  Synopsis   [This file defines the main function to do matrix-
  matrixmultiplication.]
  Description []
*****************************************************************/
//-----------------------------------------------------------
// Included C libraries
//-----------------------------------------------------------
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
```

```
//-----------------------------------------------------------
// Included CUDA libraries
//-----------------------------------------------------------
#include <cutil.h>
//-----------------------------------------------------------
// Included helper functions
//-----------------------------------------------------------
#include "assist.h"
//-----------------------------------------------------------
//Included    host    matrix-matrix    multiplication    function
  prototype
//-----------------------------------------------------------
#include "matrixmul.h"
/*= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/*                                                          */
/* Synopsis   [Main function]                               */
/* Description []                                           */
/*                                                          */
/*= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
   int
main(int argc, char** argv)
{
   bool if_quiet = false;
   unsigned int timer_compute = 0;
   int i, j;
   char * matrix_id = NULL, * input_fn = NULL, * gold_fn = NULL;
   int Mw = 0, Mh = 0, Nw = 0, Nh = 0, Pw = 0, Ph = 0;

   if (argc = = 2) {
     matrix_id = strdup(argv[1]);
   } else {
     fprintf(stderr, "Error: Wrong input parameter
     numbers.\n");
     fprintf(stderr, "Usage:\n"
                     "$> ./lab1.1-matrixmul <8, 128, 512,
                     3072, 4096> \n"
                     "Examples:\n"
                     "   $> ./lab1.1-matrixmul 128\n"
              );
     exit(1);
   }
   Mw = Mh = Nw = Nh = Pw = Ph = atoi(matrix_id);
   input_fn = (char *) malloc(30*sizeof(char));
   gold_fn = (char *) malloc(30*sizeof(char));
   sprintf(input_fn, "matrix_%s.bin", matrix_id);
   sprintf(gold_fn, "matrix_%s.gold", matrix_id);
   if (Pw*Ph > 15*15) {
```

```
      if_quiet = true; // If not display matrix contents
    }
    printf("Input matrix size: %d by %d\n", Mw, Mh);
    //———————————————————————————————————
    // Setup host side
    //———————————————————————————————————
    printf("Setup host side environment:\n");

    // allocate host memory for matrices M and N
    printf(" Allocate host memory for matrices M and N.\n");
    printf("   M: %d x %d\n", Mw, Mh);
    printf("   N: %d x %d\n", Nw, Nh);
    unsigned int size_M = Mw * Mh;
    unsigned int mem_size_M = sizeof(float) * size_M;
    float* hostM = (float*) malloc(mem_size_M);
    unsigned int size_N = Nw * (Nh);
    unsigned int mem_size_N = sizeof(float) * size_N;
    float* hostN = (float*) malloc(mem_size_N);

    // allocate memory for the result on host side
    printf(" Allocate memory for the result on host side.\n");
    unsigned int size_P = Pw * Ph;
    unsigned int mem_size_P = sizeof(float) * size_P;
    float* hostP = (float*) malloc(mem_size_P);

    // Initialize the input matrices.
    printf(" Generate input matrix data for matrix M and N.\n");
    GenMatrixFile(input_fn, Pw, Ph, if_quiet);
    unsigned int * matrix = ReadMatrixFile(input_fn, Pw, Ph,
  true);
    for (i = 0; i < Mw; i++)
      for (j = 0; j < Nw; j++)
        hostM[i * Mw + j] = hostN[i * Mw + j] = (float)
        matrix[i*Mw + j];
    free(matrix); matrix = NULL;
 // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
// Do matrix-matrix multiplication
// = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    printf(" Computing matrix multiplication M x N:\n");
    if (Pw*Ph > 512*512) {
          printf("  (It takes time since matrix is larger than
          512by512.\n");
    }
    CUT_SAFE_CALL(cutCreateTimer(&timer_compute));
    CUT_SAFE_CALL(cutStartTimer(timer_compute));

    float* reference = (float*) malloc(mem_size_P);
    computeGold(reference, hostM, hostN, Mh, Mw, Nw);
    CUT_SAFE_CALL(cutStopTimer(timer_compute));
```

```
  printf(" CPU Processing time : %f (ms)\n",
    cutGetTimerValue(timer_compute));
  CUT_SAFE_CALL(cutDeleteTimer(timer_compute));

  printf(" Matrix data checksum : %g\n", CheckSum(reference,
  Mw, Nw));

  if (!if_quiet) {
    printf(" Matrix data contents :\n");
    printf("  ");
  }
  matrix = (unsigned int *) malloc(Pw * Ph * sizeof(unsigned
  int));
  for (i = 0; i < Ph; i++) {
    for (j = 0; j < Pw; j++) {
      matrix[i*Pw + j] = (unsigned int) reference[i*Pw + j];
      if (!if_quiet) printf("%u ", matrix[i*Pw + j]);
    }
    if (!if_quiet) printf("\n  ");
  }
  if (!if_quiet) printf("\n");

  WriteMatrixFile(gold_fn, matrix, Pw, Ph, 1);
  free(matrix); matrix = NULL;
  free(reference);

  // clean up memory
  free(hostM); free(hostN); free(hostP);
  free(input_fn); free(gold_fn);
  return 0;
}
```

A.2 matrixmul_gold.cpp

This "gold" version of the matrix multiplication function can be used to verify the results of your parallel implementation.

```
/****************************************************************************

  File Name  [matrixmul_gold.cpp]

  Synopsis [This file defines the gold-version matrix-matrix
    multiplication.]

  Description []
*****************************************************************************/
#include <stdio.h>
#include "matrixmul.h"
/*= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
/*                                                              */
```

```
/*   Synopsis      [Sequential/Gold version of matrix-matrix
  multiplication.]                                              */
/*                                                              */
/*  Description [This function computes multiplication of two
  matrix M and N,*/
/*   and stores the output to P.]                               */
/*                                                              */
/*= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =*/
  void
computeGold(
  float* P,          // Resultant matrix data
  const float* M,    // Matrix M
  const float* N,    // Matrix N
  int Mh,            // Matrix M height
  int Mw,            // Matrix M width
  int Nw)            // Matrix N width
{
 int i, j, k;
 float sum, a, b;

 for (i = 0; i < Mh; i++)
  for (j = 0; j < Nw; j++)
   {
     sum = 0;
     for (k = 0; k < Mw; k++)
      {
        a = M[i * Mw + k];
        b = N[k * Nw + j];
       //printf ("A[%d] * B[%d]\n", i * Mw + k, k * Nw + j);
        sum + = a * b;
      }
     P[i * Nw + j] = (float)sum;
   }
}
```

A.3 `matrixmul.h`

This file contains the function prototype of the gold-version of matrix-matrix multiplication.

```
/********* *****************************************************************
   File Name  [matrixmul.h]

   Synopsis   [This file defines the function prototype of the
   gold-versionmatrix-matrix multiplication.]
   Description []
   *****************************************************************/
```

```
#ifndef MATRIXMUL_H
#define MATRIXMUL_H
extern "C"
void computeGold(
  float* P, const float* M, const float* N, int Mh, int Mw, int Nw);
#endif
```

A.4 assist.h

This file contains helper functions that assist in reading, writing, and verifying matrix data files to make your implementation easy.

```
/*****************************************************************************
   File Name  [assist.h]
   Synopsis  [This file defines the helper functions to assist
        In file access and result verification in matrix-matrix
  multiplication.]
   Description []
 *****************************************************************************/
   FILE *
OpenFile (
   const char * const fn_p,
   const char * const open_mode_p,
   const int if_silent       // If not show messages
   )
{
   FILE * f_p = NULL;
   if (fn_p = = NULL) {
     printf ("Null file name pointer.");
     exit (-1);
   }
   if (!if_silent) {
     fprintf(stdout,"Opening the file %s ... ", fn_p);
   }
   f_p = fopen(fn_p, open_mode_p);
   if (f_p = = NULL) {
     if (!if_silent) {
       fprintf(stdout,"failed.\n");
     } else {
        fprintf(stdout,"\nOpening the file %s ... failed.\n\n",
        fn_p);
     }
  exit (-1);
   }
   if (!if_silent) fprintf(stdout,"succeeded.\n");
   return (f_p);
```

```
}
   int
GenMatrixFile (
   const char * const matrix_fn_p,
   const unsigned int M_WIDTH,     // matrix width
   const unsigned int M_HEIGHT,    // matrix height
   const int if_silent     // If not show messages
   )
{
   FILE * matrix_fp = NULL;
   const unsigned int M_SIZE = M_WIDTH * M_HEIGHT;
   unsigned int * matrix = NULL;
   unsigned int i = 0, j = 0;

   matrix_fp = OpenFile (matrix_fn_p, "wb", 1);
   matrix = (unsigned int *) malloc (M_SIZE * sizeof
   (unsigned int));
     //if (!if_silent) fprintf (stdout, "Generated contents of
   matrix:\n");
   if (!if_silent) fprintf (stdout, "   ");
   for (i = 0; i < M_HEIGHT; i++) {
    for (j = 0; j < M_WIDTH; j++) {
     matrix[i*M_WIDTH + j] = i + j + 1;
     if (!if_silent) fprintf (stdout, "%u ", matrix
     [i*M_WIDTH + j]);
    }
    if (!if_silent) fprintf (stdout, "\n   ");
   }
   if (!if_silent) fprintf (stdout, "\n");

   fwrite (matrix, 1, M_SIZE * sizeof (unsigned int), matrix_fp);
   fclose (matrix_fp);
   free (matrix); matrix = NULL;
   return (1);
}
   unsigned int *
ReadMatrixFile (
   const char * const matrix_fn_p,
   const unsigned int M_WIDTH,     // matrix width
   const unsigned int M_HEIGHT,    // matrix height
   const int if_silent     // If not show messages
   )
{
   FILE * matrix_fp = NULL;
   const unsigned int M_SIZE = M_WIDTH * M_HEIGHT;
   unsigned int * matrix = NULL;
   unsigned int i = 0, j = 0;
```

```
    matrix_fp = OpenFile(matrix_fn_p, "rb", if_silent);
     matrix = (unsigned int *) malloc(M_SIZE * sizeof (unsigned
  int));
    fread(matrix, 1, M_SIZE * sizeof (unsigned int), matrix_fp);
  if (!if_silent) {
    fprintf (stdout, "Read contents of matrix:\n");
    fprintf (stdout, "   ");
    for (i = 0; i < M_HEIGHT; i++) {
      for (j = 0; j < M_WIDTH; j++) {
        fprintf (stdout, "%u ", matrix[i*M_WIDTH + j]);
      }
      fprintf (stdout, "\n   ");
    }
    fprintf(stdout, "\n");
    }
    fclose (matrix_fp);
    return (matrix);
}
    int
WriteMatrixFile (
   const char * const matrix_fn_p,
   const unsigned int * const matrix,
   const unsigned int M_WIDTH,      // matrix width
   const unsigned int M_HEIGHT,     // matrix height
   const int if_silent     // If not show messages
   )
{
   FILE * matrix_fp = NULL;
   const unsigned int M_SIZE = M_WIDTH * M_HEIGHT;
   unsigned int i = 0, j = 0;
   matrix_fp = OpenFile (matrix_fn_p, "wb", if_silent);
   fwrite (matrix, 1, M_SIZE * sizeof (unsigned int), matrix_fp);

  if (!if_silent) {
    fprintf (stdout, "Written contents of matrix:\n");
    for (i = 0; i < M_HEIGHT; i++) {
      for (j = 0; j < M_WIDTH; j++) {
        fprintf (stdout, "%u ", matrix[i*M_WIDTH + j]);
      }
      fprintf (stdout, "\n");
    }
    }
    fclose (matrix_fp);
    return (1);
}
// Usage:
// CompareMatrixFile ("your output", "golden output", WC, HC, 1);
```

```
     void
CompareMatrixFile (
    const char * const matrix_fn_p1,
    const char * const matrix_fn_p2,
    const unsigned int M_WIDTH,        // matrix width
    const unsigned int M_HEIGHT,       // matrix height
    const int if_silent       // If not show messages
    )
{
    unsigned int i = 0, j = 0, wrong = 0;
    int check_ok = 1;
     unsigned int * m1 = ReadMatrixFile (matrix_fn_p1, M_WIDTH,
    M_HEIGHT, if_silent);
     unsigned int * m2 = ReadMatrixFile (matrix_fn_p2, M_WIDTH,
    M_HEIGHT, if_silent);
      printf (" Comparing file %s with %s ...\n", matrix_fn_p1,
  matrix_fn_p2);
    for (i = 0; i < M_HEIGHT && wrong < 15; i++) {
       for (j = 0; j < M_WIDTH && wrong < 15; j++) {
           //printf ("m1[%d][%d] ? = m2[%d][%d] : %d ? = %d\n",
           //    i,j,i,j, m1[i*M_WIDTH + j], m2[i*M_WIDTH + j]);
          if (m1[i*M_WIDTH + j] ! = m2[i*M_WIDTH + j]) {
            printf ("m1[%d][%d] ! = m2[%d][%d] : %d ! = %d\n",
                i,j,i,j, m1[i*M_WIDTH + j], m2[i*M_WIDTH + j]);
            check_ok = 0; wrong++;
          }
       }
    }
    printf ("   Check ok? ");
    if (check_ok) printf ("Passed.\n");
    else printf ("Failed.\n");
}
    float
CheckSum(const float *matrix, const int width, const int height)
{
    int i, j;
    float s1, s2;
    for (i = 0, s1 = 0; i < width; i++) {
       for (j = 0, s2 = 0; j < height; j++) {
         s2 + = matrix[i * width + j];
       }
       s1 + = s2;
    }
    return s1;
}
```

A.5 **EXPECTED OUTPUT**

This is the expected output when you test your implementation of matrix-matrix multiplication.

```
Input matrix size: 8 by 8
Setup host side environment:
 Allocate host memory for matrices M and N.
  M: 8 × 8
  N: 8 × 8
 Allocate memory for the result on host side.
 Generate input matrix data for matrix M and N.
  1 2 3 4 5 6 7 8
  2 3 4 5 6 7 8 9
  3 4 5 6 7 8 9 10
  4 5 6 7 8 9 10 11
  5 6 7 8 9 10 11 12
  6 7 8 9 10 11 12 13
  7 8 9 10 11 12 13 14
  8 9 10 11 12 13 14 15

 Computing matrix multiplication M x N:
 CPU Processing time : 0.009000 (ms)
 Matrix data checksum : 35456
 Matrix data contents :
  204 240 276 312 348 384 420 456
  240 284 328 372 416 460 504 548
  276 328 380 432 484 536 588 640
  312 372 432 492 552 612 672 732
  348 416 484 552 620 688 756 824
  384 460 536 612 688 764 840 916
  420 504 588 672 756 840 924 1008
  456 548 640 732 824 916 1008 1100
```

GPU Compute Capabilities

B

B.1 GPU COMPUTE CAPABILITY TABLES

As we discussed in Chapters 6-10, maximizing the kernel performance on a particular GPU requires knowledge of the resource limitations in the GPU hardware. Therefore, the main hardware resource provisions in each GPU are typically exposed to applications in a standardized system called *compute capability*. The general specifications and features of a compute device depend on its compute capability. For CUDA, the compute capability starts at Compute 1.0, and at the time of this writing the latest version is Compute 3.5. Each higher level of compute capability indicates a newer generation of GPU devices with a higher number of features. Table B.1 highlights the key features support differences between each of the compute capabilities. Features not listed can be considered supported by all compute capability variations; differences in memory coalescing are discussed in Section B.2. In general, a higher-level compute capability defines a superset of features of those of a lower level.

Table B.2 shows the main dimensions of compute capability specifications and gives the numerical value of each dimension for Compute 3.5. Each higher level of compute capability enhances one more of these dimensions.

Depending on the time of its introduction, each CUDA-enabled device supports up to a particular generation of compute capability. Many CUDA-enabled devices are introduced each year. Readers should refer to *http://developer.nvidia.com/cuda-gpus* for an updated list.

Many device-specific features and sizes can be determined calling runtime CUDA function `cudaGetDeviceProperties()`. See the *CUDA Programmer Guide* for more details.

Table B.1 Key Functional Support Variations Between CUDA Compute Capabilities

Feature Support Differences		Compute Capability					
		1.0	1.1	1.2	1.3	2.x	3.0	3.5
Atomic functions operating on 32-bit integer values in global memory		No		Yes			
`atomicExch()` operating on 32-bit floating-point values in global memory							
Atomic functions operating on 32-bit integer values in shared memory		No			Yes		
Atomic functions operating on 64-bit integer values in global memory							
Warp vote functions							
Double-precision floating-point numbers		No			Yes		
Atomic functions operating on 64-bit integer values in shared memory		No				Yes	
Atomic additions operating on 32-bit floating-point values in global and shared memory							
Enhanced warp vote functions							
Memory fence functions							
Synchronization functions							
3D grid support							
Funnel shift		No						Yes
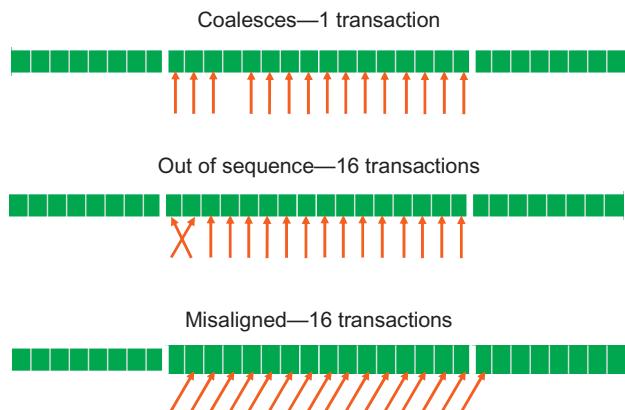
B.2 MEMORY COALESCING VARIATIONS

Each level of compute capability also specifies a different level of hardware memory coalescing capability. Knowing the compute capability, one can determine the number of global memory transactions that a load instruction in a warp will incur. Later compute capabilities such as 2.x and higher substantially reduce the number of memory transactions and occurrence of noncoalesced accesses. In Compute 1.0 and Compute 1.1, memory transactions are done for either memory 64 B or 128 B segments. Coalescing of accesses in a warp requires that the kth thread in a warp access the kth word in a 64 B segment when accessing 32-bit words (or the kth word in two contiguous 128 B segments when accessing 128-bit words). Not all threads need to participate for coalescing to work. In the top of Figure B.1, one of the threads in a warp did not participate and the accesses are still coalesced into one transaction.
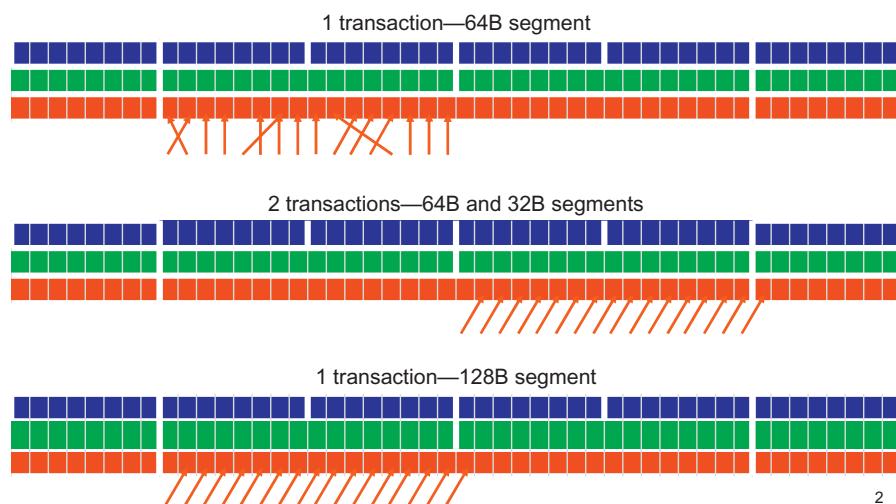
Table B.2 Main Dimensions of Compute Capability and the Attributes of Compute 3.5

Features	Compute 3.5
Number of stream processors per multiprocessor (MP)	192
Max. number of threads per block	1,024
Max. grid dimensions X, Y, Z	$2^{31} - 1,65535,65535$
Max. block dimensions X, Y, Z	1,024, 1,024, 64
Threads in a warp	32
Registers per MP	65,536 (64 K)
Shared memory per MP	49,152 (48 K)
Banks in shared memory	32
Total constant memory	65,536 (64 K)
Cache working set for constants per MP	8,192 (8 K)
Local memory per thread	524,288 (512 K)
Cache working set for texture per MP	6−8 KB
Max. number of active blocks per MP	16
Max. number of active warps per MP	64
Max. number of active threads per MP	2,048
1D texture bound to CUDA array—max. width	65,536
1D texture bound to linear memory—max. width	2^{27}
2D texture bound to linear memory or CUDA array; max. dimensions X, Y	65,000 and 65,536, respectively
3D texture bound to a CUDA array max. dimensions X, Y, Z	$4\,K \times 4\,K \times 4\,K$
Max. width, height, and layers for a cube map—layered texture reference	$16,384 \times 16,384 \times 2,046$
Max. number of textures that can be bound to a kernel	256
1D surface reference bound to a CUDA array—max. width	65,536
1D layered surface reference—max. width and layers	$65,536 \times 2,048$
2D layered surface reference—max. width, height, and layers	$65,536 \times 32,768 \times 2,048$
3D layered surface reference—max. width, height, and depth bound to a CUDA array	$65,536 \times 32,768 \times 2,048$
Max. width, height, and layers for a cube map—layered surface reference	$32,768 \times 32,768 \times 2,046$
Max. number of surfaces	
Max. number of instructions per kernel	512 million microcode instructions
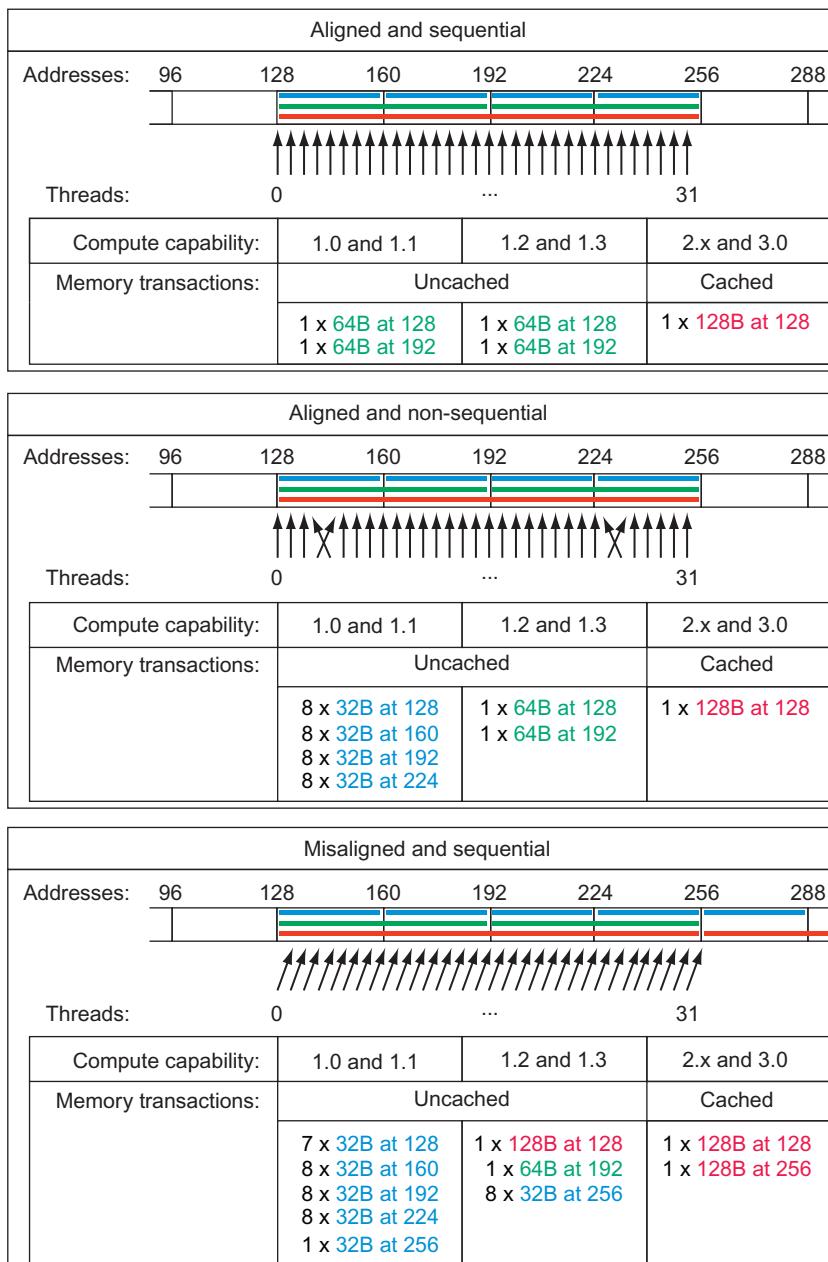
In particular, all accesses must be in sequence. If one or more of the accesses are out of sequence, the accesses will no longer be coalesced. In the middle of Figure B.1, two of the accesses are out of sequence. The accesses are therefore not coalesced; 16 transactions to the global memory are done for the access.

FIGURE B.1

Memory coalescing in compute 1.0 and Compute 1.1.

FIGURE B.2

Memory coalescing in compute 1.2 and higher.

FIGURE B.3

Examples of global memory access and resulting memory transactions for each compute capability.

In Compute 1.2 and higher, the global memory transactions are issued in 32 B, 64 B, or 128 B segments. Having a smaller segment size allows the hardware to reduce waste of global memory bandwidth for some less coherent warp access patterns.

Figure B.2 illustrates improvements in memory coalescing in Compute 1.2. The top part shows that warp accesses within a segment can be out of sequence and still be fully coalesced into one transaction.

The middle part shows that the access can be nonaligned across a 128 B boundary. One extra 32 B segment transaction will be issued and the accesses are still coalesced. The bottom part shows if warp accesses are nonaligned but stay within a 128 B boundary, a single 128 B segment transaction will be used to access all the words involved. In these two cases, the global memory bandwidth consumption is much less than that in Compute 1.0 or Compute 1.1 where 16 transactions of 64 B segments would be used.

Figure B.3 illustrates the improvements introduced in Compute 2.0 resulting in all aligned memory accesses to be considered coalesced and eliminating additional memory transactions.

Index

Note: Page numbers followed by "*f*" and "*b*" refer to figures and boxes respectively.