Programming Massively Parallel Processors

A Hands-on Approach

B978-0-12-381472-2.00014-3, 00014

Programming Massively Parallel Processors

A Hands-on Approach

David Kirk and Wen-mei Hwu

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA

This book is printed on acid-free paper.

Notices

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

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

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

Library of Congress Cataloging-in-Publication Data
Application Submitted

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-381472-2

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.elsevierdirect.com

Printed in United States of America
10 11 12 13 14 5 4 3 2 1

Contents

Preface

WHY WE WROTE THIS BOOK

Mass-market computing systems that combine multi-core CPUs and many-core GPUs have brought terascale computing to the laptop 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 unprecedented level of scale, 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.

During the Christmas holiday of 2006, we were frantically working on the lecture slides and lab assignments. David was working the system trying to pull the early GeForce 8800 GTX GPU cards from customer shipments to Illinois, which would not succeed until a few weeks after the semester began. It also became clear that CUDA would not become public until a few weeks after the start of the semester. We had to work out the legal agreements so that we can offer the course to students under NDA for the first few weeks. We also needed to get the words out so that students would sign up since the course was not announced until after the pre-enrollment period.

We gave our first lecture on January 16, 2007. Everything fell into place. David commuted weekly to Urbana for the class. We had 52 students, a couple more than our capacity. We had draft slides for most of the first 10 lectures. Wen-mei's graduate student, John Stratton, graciously volunteered as the teaching assistant and set up the lab. All students signed NDA so that we can proceed with the first several lectures until CUDA became public. We recorded the lectures but did not release them on the Web until February. We had graduate students from physics, astronomy, chemistry, electrical engineering, mechanical engineering as well as computer science and computer engineering. The enthusiasm in the room made it all worthwhile.

Since then, we have taught the course three times in one-semester format and two times in one-week intensive format. The ECE498AL course has become a permanent course known as ECE408 of the University of Illinois, Urbana-Champaign. We started to write up some early chapters of this book when we offered ECE498AL the second time. We tested these

chapters in our spring 2009 class and our 2009 Summer School. The first four chapters were also tested in an MIT class taught by Nicolas Pinto in spring 2009. We also shared these early chapters on the web and received valuable feedback from numerous individuals. We were encouraged by the feedback we received and decided to go for a full book. Here, we humbly present our first edition to you.

TARGET AUDIENCE

The target audience of this book is graduate and undergraduate students from all science and engineering disciplines the computational thinking and parallel programming skills needed to use pervasive terascale computing hardware to achieve breakthroughs in their own disciplines. We assume that the reader has at least some basic C programming experience and thus are more advanced programmers, both within and outside of the field of Computer Science. We especially target computational scientists in fields such as mechanical engineering, civil engineering, electrical engineering, bio-engineering, physics, and chemistry, who use computation to further their field of research. As such, these scientists are both experts in their domain as well as advanced programmers. The book takes the approach of building on basic C programming skills, to teach parallel programming in C. We use C for CUDA™, a parallel programming environment that is supported on NVIDIA GPUs, and emulated on less parallel CPUs. There are approximately 200 million of these processors in the hands of consumers and professionals, and more than 40,000 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 ECE498AL using the material is this book.

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 naïve parallel matrix multiplication code in a couple of hours.

Phase 2: The next phase is a series of ten lectures that give students the *conceptual* understanding of the CUDA memory model, the CUDA threading model, GPU hardware performance features, modern computer system architecture, and the common data-parallel programming patterns needed to develop a high-performance parallel application. These lectures are based on Chapters 4 through 7. 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 scan through this period.

Phase 3: Once the students have established solid CUDA programming skills, the remaining lectures cover computational thinking, a broader range of parallel execution models, and parallel programming principles. These lectures are based on Chapters 8 through 11. (The voice and video recordings of these lectures are available on-line (http://courses.ece.illinois.edu/ece498/al).

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 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 final project is available at the ECE498AL web site (http://courses.ece.illinois.edu/ece498/al), 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 recruit several major computational science research groups to propose problems and serve as mentors. The mentors are asked to contribute a one-to-two-page project specification sheet that briefly describes the significance of the application, what the mentor would like to accomplish with the student teams on the application, the technical skills (particular type of Math, Physics, Chemistry courses) required to understand and work on the application, and a list of web and traditional resources that students can draw upon for technical background, general

information, and building blocks, along with specific URLs or ftp paths to particular implementations and coding examples. These project specification sheets also provide students with learning experiences in defining their own research projects later in their careers. (Several examples are available at the ECE498AL 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. We the instructors attempt to facilitate the collaborative relationship between students and their mentors, making it a very valuable experience for both mentors and students.

The 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/she can simply attend the presentations, participate in the discussions, and join one of the project teams. Students are not graded during the workshops, in order to keep the atmosphere non-threatening and enable them to focus on a meaningful dialog with the instructor(s), teaching assistants, and the rest of the class.

The workshop schedule is designed so the instructor(s) 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 pre-loaded into a PC in order 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 instructor(s) and TAs must make a commitment to attend all the presentations and to give useful feedback. Students typically need most help in answering the following questions. First, are the projects too big or too small for the amount of time available? Second, is there existing work in the field that the project can benefit from? Third, are the computations being targeted for parallel execution appropriate for the CUDA programming model?

The 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; (2) The best CDUA parallel code in terms of performance. This version is the main output of the project; (3) A version of CPU sequential code that is based on the same algorithm as version 3, using single precision. 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 precision 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.

The 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/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 [where are they available?].

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 on-line material that accompanies this book, which is available at the Publisher's Web site, [www.elsevierdirect.com/9780123814722].

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 on-line material. Of course, we also like to know what you liked about the book. We look forward to hearing from you.

David B. Kirk and Wen-mei W. Hwu

Acknowledgements

We would like to especially acknowledge Ian Buck, the father of CUDA and John Nicolls, the lead architect of Tesla GPU Computing Architecture. Their teams laid created an excellent infrastructure for this course. Ashutosh Rege and the NVIDIA DevTech team contributed to the original slides and contents used in ECE498AL course. Bill Bean, Simon Green, Mark Harris, Manju Hedge, Nadeem Mohammad, Brent Oster, Peter Shirley, Eric Young and Cyril Zeller provided review comments and corrections to the manuscripts. Nadeem Mohammad organized the NVIDIA review efforts and also helped plan Chapter 11 and Appendix B. Calisa Cole helped with cover. Nadeem's heroic efforts have been critical to the completion of this book.

We would like to especially thank Jensen Huang for providing a great amount of financial and human resources for developing the course. Tony Tamasi's team contributed heavily to the review and revision of the book chapters. Jensen also took the time to read the early drafts of the chapters and gave us valuable feedback. David Luebke has facilitated the GPU computing resources for the course. Jonah Alben has provided valuable insight. Michael Shebanow and Michael Garland have given guest lectures and offered materials.

John Stone and Sam Stone in Illinois contributed much of the base material for the case study and OpenCL chapters. John Stratton and Chris Rodrigues contributed some of the base material for the computational thinking chapter. I-Jui "Ray" Sung, John Stratton, Xiao-Long Wu, Nady Obeid contributed to the lab material and helped to revise the course material as they volunteered to serve as teaching assistants on top of their research. Laurie Talkington and James Hutchinson helped to dictate early lectures that served as the base of the first five chapters. Mike Showerman helped build two generations of GPU computing clusters for the course. Jeremy Enos worked tirelessly to ensure that students have a stable, user-friendly GPU computing cluster to work on their lab assignments and projects.

We would like to acknowledge Dick Blahut who challenged us to create the course in Illinois. His constant reminder that we needed to write the book helped keep us going. Beth Katsinas arranged a meeting between Dick Blahut and NVIDIA Vice President Dan Vivoli. Through that gathering, Blahut was introduced to David and challenged David to come to Illinois and create the course with Wen-mei.

We would also like to thank Thom Dunning of the University of Illinois and Sharon Glotzer of the University of Michigan, Co-Directors of the multi-university Virtual School of Computational Science and Engineering, for graciously hosting the summer school version of the course. Trish Barker, Scott Lathrop, Umesh Thakkar, Tom Scavo, Andrew Schuh, and Beth McKown all helped organize the summer school. Robert Brunner, Klaus Schulten, Pratap Vanka, Brad Sutton, John Stone, Keith Thulborn, Michael Garland, Vlad Kindratenko, Naga Govindaraj, Yan Xu, Arron Shinn, and Justin Haldar contributed to the lectures and panel discussions at the summer school.

Nicolas Pinto tested the early versions of the first chapters in his MIT class and assembled an excellent set of feedback comments and corrections. Steve Lumetta and Sanjay Patel both taught versions of the course and gave us valuable feedback. John Owens graciously allowed us to use some of his slides. Tor Aamodt, Dan Connors, Tom Conte, Michael Giles, Nacho Navarro and numerous other instructors and their students worldwide have provided us with valuable feedback.

We would like to especially thank our colleagues Kurt Akeley, Al Aho, Arvind, Dick Blahut, Randy Bryant, Bob Colwell, Ed Davidson, Mike Flynn, John Hennessy, Pat Hanrahan, Nick Holonyak, Dick Karp, Kurt Keutzer, Dave Liu, Dave Kuck, Yale Patt, David Patterson, Bob Rao, Burton Smith, Jim Smith and Mateo Valero who have taken the time to share their insight with us over the years.

We are humbled by the generosity and enthusiasm of all the great people who contributed to the course and the book.

David B. Kirk and Wen-mei W. Hwu

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

CHAPTER

c0005 Introduction 1

CHAPTER CONTENTS

s0010 INTRODUCTION

p0045 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 giga (billion) floating-point operations per second (GFLOPS) to the desktop and hundreds of GFLOPS to cluster servers. This relentless drive of 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 cycle for the computer industry.

p0050 During the drive, most software developers have relied on the advances in hardware to increase the speed of their applications under the hood; the same software simply runs faster as each new generation of processors is introduced. This drive, however, has slowed since 2003 due to energy-consumption and heat-dissipation issues that have 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. 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

Programming Massively Parallel Processors: A Hands-on Approach. DOI: 10.1016/B978-0-12-381472-2.00001-5

processing power. This switch has exerted a tremendous impact on the software developer community [Sutter 2005].

p0055 Traditionally, the vast majority of software applications are written as sequential programs, as described in 1945 by von Neumann in his seminal report [von Neumann 1945]. The execution of these programs can be understood by a human sequentially stepping through the code. Historically, computer users have 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 any 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, thus reducing the growth opportunities of the entire computer industry.

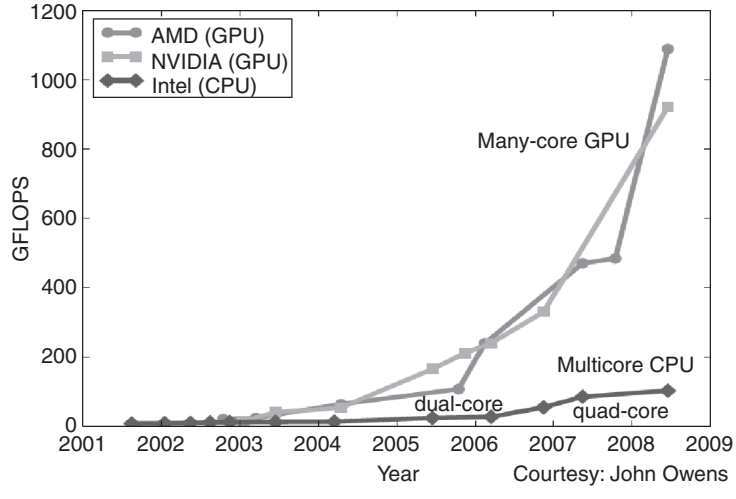
p0060 Rather, the applications software that will continue to enjoy performance improvement with each new generation of microprocessors will be parallel programs, in which multiple threads of execution cooperate to complete the work faster. This new, dramatically escalated incentive for parallel program development has been referred to as the *concurrency revolution* [Sutter 2005]. 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 must 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.

s0015 1.1 GPUs AS PARALLEL COMPUTERS

p0065 Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessor [Hwu 2008]. The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began as two-core processors, with the number of cores doubling with each semiconductor process generation. A current exemplar is the recent Intel® Core™ i7 microprocessor, which

has four processor cores, each of which is an out-of-order, multiple-instruction issue processor implementing the full x86 instruction set; the microprocessor supports hyperthreading with two hardware threads and is designed to maximize the execution speed of sequential programs.

p0070 In contrast, the *many-core* trajectory focuses more on the execution throughput of parallel applications. The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation. A current exemplar is the NVIDIA® GeForce® GTX 280 graphics processing unit (GPU) with 240 cores, each of which is a heavily multithreaded, in-order, single-instruction issue processor that shares its control and instruction cache with seven other cores. Many-core processors, especially the GPUs, have led the race of floating-point performance since 2003. This phenomenon is illustrated in Figure 1.1. While the performance improvement of general-purpose microprocessors has slowed significantly, the GPUs have continued to improve relentlessly. As of 2009, the ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 10 to 1. These are not necessarily achievable application speeds but are merely the raw speed that the execution resources can potentially support in these chips: 1 teraflops (1000 gigaflops) vs. 100 gigaflops in 2009.



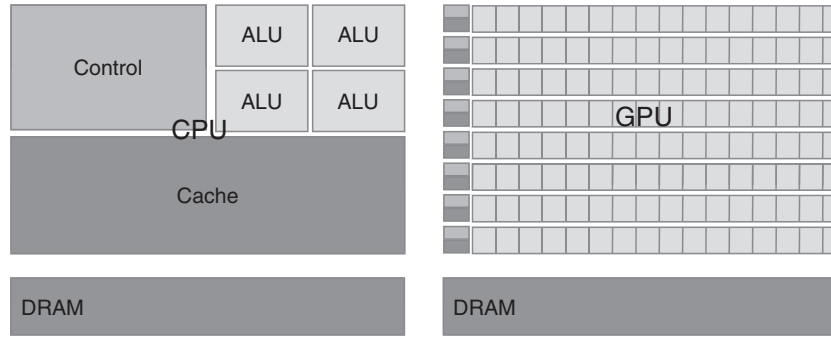
f0010 **FIGURE 1.1**

Enlarging performance gap between GPUs and CPUs.

p0075 Such a large performance gap between parallel and sequential execution has amounted to a significant "electrical potential" buildup, 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 applications 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.

p0080 One might ask why there is such a large performance gap between many-core 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 1.2. 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 of execution 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 2009, the new general-purpose, Au1 multicore microprocessors typically have four large processor cores designed to deliver strong sequential code performance.

p0085 Memory bandwidth is another important issue. Graphics chips have been operating at approximately 10 times the bandwidth of contemporaneously available CPU chips. In late 2006, the GeForce® 8800 GTX, or simply



f0015 **FIGURE 1.2**
CPUs and GPUs have fundamentally different design philosophies.

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 frame buffer requirements and the relaxed memory model—the way various system software, applications, and input/output (I/O) devices expect their memory accesses to work—general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. In contrast, with simpler memory models and fewer legacy constraints, the GPU designers can more easily achieve higher memory bandwidth. The more recent NVIDIA® GT200 chip supports about 150 GB/s. Microprocessor system memory bandwidth will probably not grow beyond 50 GB/s for about 3 years, so CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

p0090 The design philosophy of the GPUs is shaped by the fast growing video game industry, which exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced games. This demand motivates the GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The prevailing solution to date is to optimize for the execution throughput of massive numbers of threads. The hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, thus minimizing the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications so multiple threads that access the same memory data do not need to all go to the DRAM. As a result, much more chip area is dedicated to the floating-point calculations.

p0095 It should be clear now that GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well; therefore, one should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs. This is why the CUDA™ (Compute Unified Device Architecture) programming model, introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of an application.[1]

fn0010 [1]See Chapter 2 for more background on the evolution of GPU computing and the creation of CUDA.

p0100 It is also important to note that performance is not the only decision factor when application developers choose the processors for running their applications. Several other factors can be even more important. First and foremost, the processors of choice must have a very large presence in the marketplace, referred to as the installation base of the processor. The reason is very simple. The cost of software development is best justified by a very large customer population. Applications that run on a processor with a small market presence will not have a large customer base. This has been a major problem with traditional parallel computing systems that have negligible market presence compared to general-purpose microprocessors. Only a few elite applications funded by government and large corporations have been successfully developed on these traditional parallel computing systems. This has changed with the advent of many-core GPUs. Due to their popularity in the PC market, hundreds of millions of GPUs have been sold. Virtually all PCs have GPUs in them. The G80 processors and their successors have shipped more than 200 million units to date. This is the first time that massively parallel computing has been feasible with a mass-market product. Such a large market presence has made these GPUs economically attractive for application developers.

p0105 Other important decision factors are practical form factors and easy accessibility. Until 2006, parallel software applications usually ran on data-center servers or departmental clusters, but such execution environments tend to limit the use of these applications. For example, in an application such as medical imaging, it is fine to publish a paper based on a 64-node cluster machine, but actual clinical applications on magnetic resonance imaging (MRI) machines are all based on some combination of a PC and special hardware accelerators. The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks of clusters to clinical settings, but this is common in academic departmental settings. In fact, the National Institutes of Health (NIH) refused to fund parallel programming projects for some time; they 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.

p0110 Yet another important consideration in selecting a processor for executing numeric computing applications is the support for the Institute of Electrical and Electronics Engineers (IEEE) floating-point standard. The standard makes it possible to have predictable results across processors from different vendors. While 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 to that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable values as the CPUs. Today, a major remaining issue is that the floating-point arithmetic units of the GPUs are primarily single precision. Applications that truly require double-precision floating point were not suitable for GPU execution; however, this has changed with the recent GPUs, whose double-precision execution speed approaches about half that of single precision, a level that high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications.

p0115 Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphic application programming interface (API) functions to access the processor cores, meaning that OpenGL® or Direct3D® techniques were needed to program these chips. This technique was called GPGPU, short for general-purpose programming using a graphics processing unit. Even with a higher level programming environment, the underlying code is still limited by the APIs. These APIs limit the kinds of applications that one can actually write for these chips. That's why only a few people could master the skills necessary to use these chips to achieve performance for a limited number of applications; consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent results.

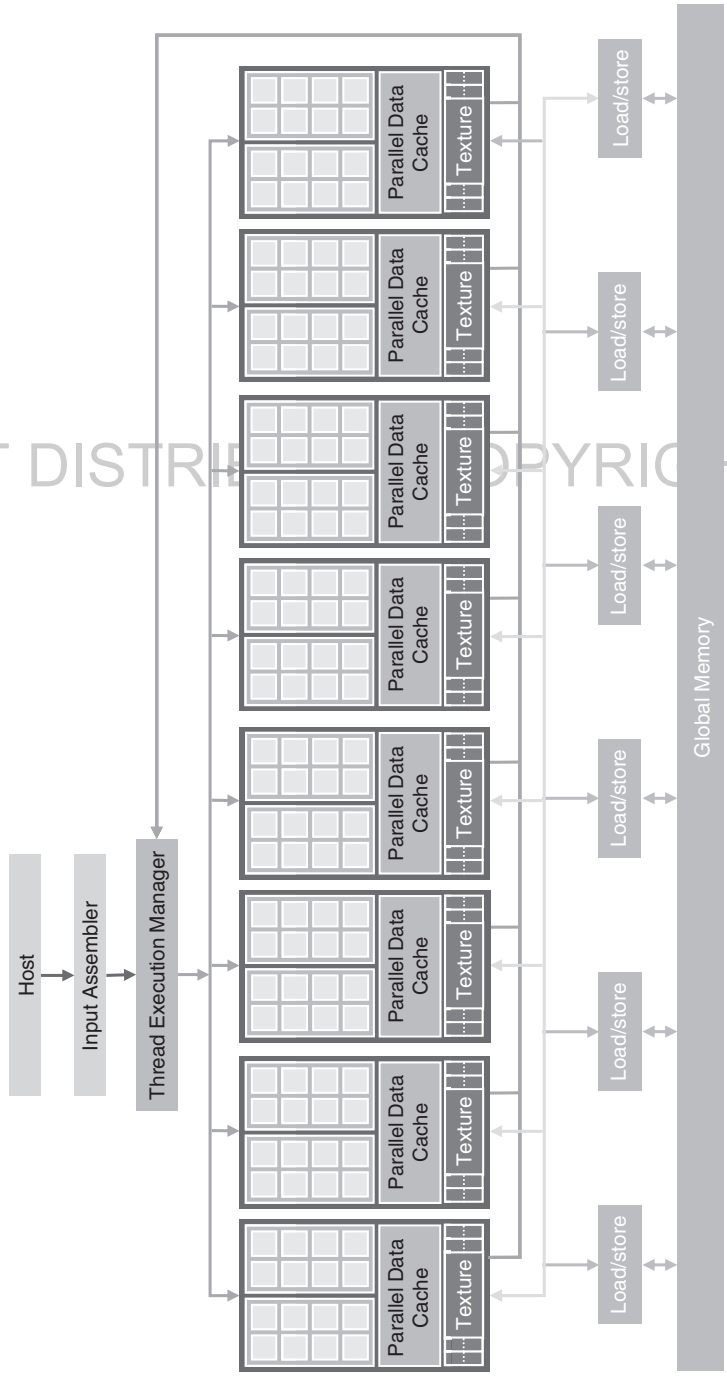
p0120 Everything changed in 2007 with the release of CUDA [NVIDIA 2007]. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming, so this did not represent a change in software alone; additional hardware was added to the chip. In the G80 and its successor chips for parallel 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. Moreover, all of the other software layers were redone, as well, so 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.

s0020 1.2 ARCHITECTURE OF A MODERN GPU

p0125 Figure 1.3 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, each SM in Figure 1.3 has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 4 gigabytes of graphics 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 three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

p0130 The G80 that introduced the CUDA architecture had 86.4 GB/s of memory bandwidth, plus an 8-GB/s communication bandwidth with the CPU. A CUDA application can transfer data from the system 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. The communication bandwidth is much lower than the memory bandwidth and may seem like a limitation; however, the PCI Express® bandwidth is comparable to the CPU front-side bus bandwidth to the system memory, so it's really not the limitation it would seem at first. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

p0135 The massively parallel G80 chip has 128 SPs (16 SMs, each with 8 SPs). Each SP has a multiply–add (MAD) unit and an additional multiply unit. With 128 SPs, that's a total of over 500 gigaflops. In addition, special-function units perform floating-point functions such as square root (SQRT), as well as transcendental functions. With 240 SPs, the GT200 exceeds 1 teraflops. Because each SP is massively threaded, it can run thousands of threads per application. A good application typically runs 5000 to 12,000 threads simultaneously on this chip. For those who are used to simultaneous multithreading, note that Intel CPUs support 2 or 4 threads, depending on the machine model, per core. The G80 chip supports up to 768 threads per SP, which sums up to about 12,000 threads for this chip. The more recent GT200 supports 1024 threads per SP and up to about 30,000 threads



f0020 **FIGURE 1.3**
Architecture of a CUDA-capable GPU.

for the chip. Thus, the level of parallelism supported by GPU hardware is increasing quickly. It is very important to strive for such levels of parallelism when developing GPU parallel computing applications.

1.3 WHY MORE SPEED OR PARALLELISM?

p0140 As we stated in Section 1.1, the main motivation for massively parallel programming is for applications to enjoy a continued increase in speed 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×) speedup over sequential execution. If the application includes what we call *data parallelism*, it is often a simple task to achieve a 10× speedup with just a few hours of work. For anything beyond that, we invite you to keep reading!

p0145 Despite the myriad computing applications in today's world, many exciting mass-market applications of the future will be what we currently consider to be *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. From the 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 with regard to science and medicine.

p0150 For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition television (HDTV) vs. older National Television System Committee (NTSC) television. Once we experience the level of details offered by HDTV, it is very hard to go back to older technology. But, consider all the processing that is necessary 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 that televisions have more computing power.

p0155 Among the benefits offered by greater computing speed are much better user interfaces. Consider the Apple® iPhone® interfaces; the user enjoys a much more natural interface with the touch screen compared to other cell phone devices, even though the iPhone has a limited-size window. Undoubtedly, future versions of these devices will incorporate higher definition, three-dimensional perspectives, voice and computer vision based interfaces, requiring even more computing speed.

p0160 Similar developments are underway in consumer electronic gaming. Imagine driving a car in a game today; the game is, in fact, simply a prearranged set of scenes. If your car bumps into an obstacle, the course of your vehicle does not change; only the game score changes. Your wheels are not bent or damaged, and it is 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 large amounts of computing power.

p0165 All of the new applications that we mentioned involve simulating a concurrent world in different ways and at different levels, with tremendous amounts of data being processed. 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. Techniques for doing so are well known to those who work with such applications on a regular basis. Thus, various granularities of parallelism do exist, but the programming model must not hinder parallel implementation, and the data delivery must be properly managed. CUDA includes such a programming model along with hardware support that facilitates parallel implementation. We aim to teach application developers the fundamental techniques for managing parallel execution and delivering data.

p0170 How many times speedup can be expected from parallelizing these super 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× speedup of the parallel portion will reduce the execution time by 29.7%. The speedup for the entire application will be only 1.4×. In fact, even an infinite amount of speedup in the parallel portion can only slash less 30% off execution time, achieving no more than 1.43× speedup.

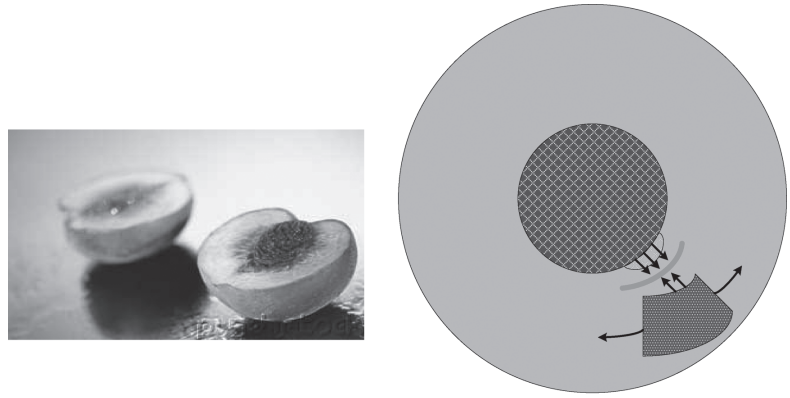
On the other hand, if 99% of the execution time is in the parallel portion, a 100× speedup will reduce the application execution to 1.99% of the original time. This gives the entire application a 50× speedup; therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speedup its execution.

p0175 Researchers have achieved speedups of more than 100× for some applications; however, this is typically achieved only after extensive optimization and tuning after the algorithms have been enhanced so more than 99.9% of the application execution time is in parallel execution. In general, straightforward parallelization of applications often saturates the memory (DRAM) bandwidth, resulting in only about a 10× speedup. The trick is to figure out how to get around memory bandwidth limitations, which involves doing one of many transformations to utilize specialized GPU on-chip memories to drastically reduce the number of accesses to the DRAM. One must, however, further optimize the code to get around limitations such as limited on-chip memory capacity. An important goal of this book is to help you to fully understand these optimizations and become skilled in them.

p0180 Keep in mind that the level of speedup achieved over CPU execution can also reflect the suitability of the CPU to the application. In some applications, CPUs perform very well, making it more difficult 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 in such a way 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.

p0185 Figure 1.4 illustrates the key parts of a typical application. Much of the code of a real application tends to be sequential. These portions are considered to be 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 difficult 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.

p0190 Then come the meat portions of the peach. These portions are easy to parallelize, as are some early graphics applications. For example, most of today's medical imaging applications are still running on combinations of



f0025 **FIGURE 1.4**

Coverage of sequential and parallel application portions.

microprocessor clusters and special-purpose hardware. The cost and size benefit of the GPUs can drastically improve the quality of these applications. As illustrated in Figure 1.4, early GPGPUs cover only a small portion of the meat section, which is analogous to a small portion of the most exciting applications coming in the next 10 years. As we will see, the CUDA programming model is designed to cover a much larger section of the peach meat portions of exciting applications.

s0030 1.4 PARALLEL PROGRAMMING LANGUAGES AND MODELS

p0195 Many parallel programming languages and models have been proposed in the past several decades [Mattson 2004]. The ones that are the most widely used are the Message Passing Interface (MPI) for scalable cluster computing and OpenMP™ for shared-memory multiprocessor systems. MPI is a model where computing nodes in a cluster do not share memory [MPI 2009]; all data sharing and interaction must be done through explicit message passing. MPI has been successful in the high-performance scientific computing domain. Applications written in MPI have been known to run successfully on cluster computing systems with more than 100,000 nodes. The amount of effort required to port an application into MPI,

however, can be extremely high due to lack of shared memory across computing 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 currently provides very limited shared memory capability between the CPU and the GPU. Programmers need to manage the data transfer between the CPU and GPU in a manner similar to "one-sided" message passing, a capability whose absence in MPI has been historically considered as a major weakness of MPI.

p0200 OpenMP supports shared memory, so it offers the same advantage as CUDA in programming efforts; however, it has not been able to scale beyond a couple hundred computing nodes due to thread management overheads and cache coherence hardware requirements. CUDA achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements. As we will see, however, CUDA does not support as wide a range of applications as OpenMP due to these scalability tradeoffs. On the other hand, many super-applications fit well into the simple thread management model of CUDA and thus enjoy the scalability and performance.

p0205 Aspects of CUDA are similar to both MPI and OpenMP in that the programmer manages the parallel code constructs, although OpenMP compilers do more of the automation in managing parallel execution. Several ongoing research efforts aim at adding more automation of parallelism management and performance optimization to the CUDA tool chain. Developers who are experienced with MPI and OpenMP will find CUDA easy to learn. Especially, many of the performance optimization techniques are common among these models.

p0210 More recently, several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a standardized programming model called OpenCL™ [Khronos 2009]. Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors. OpenCL is a standardized programming model in that applications developed in OpenCL can run without modification on all processors that support the OpenCL language extensions and API.

p0215 The reader might ask why the book is not based on OpenCL. The main reason is that OpenCL was still in its infancy when this book was written. The level of programming constructs in OpenCL is still at a lower level than CUDA and much more tedious to use. Also, the speed achieved in an application expressed in OpenCL is still much lower than in CUDA on

the platforms that support both. Because programming massively parallel processors is motivated by speed, we expect that most who program massively parallel processors will continue to use CUDA for the foreseeable future. Finally, those who are familiar with both OpenCL and CUDA know that there is a remarkable similarity between the key features of OpenCL and CUDA; that is, a CUDA programmer should be able to learn OpenCL programming with minimal effort. We will give a more detailed analysis of these similarities later in the book.

s0035 1.5 OVERARCHING GOALS

p0220 Our primary goal is to teach you, the reader, 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 materials on how to do *high-performance* parallel programming, 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.

p0225 Note that hardware architecture features have constraints. High-performance parallel programming on most of the chips will require some knowledge of how the hardware actually works. It will probably take 10 more years before we can build tools and machines so most programmers can work without this knowledge. We will not be teaching computer architecture as a separate topic; instead, we will teach the essential computer architecture knowledge as part of our discussions on high-performance parallel programming techniques.

p0230 Our second goal is 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 the users. We will show that with the CUDA programming model that focuses on data parallelism, one can achieve both high performance and high reliability in their applications.

p0235 Our third goal is achieving scalability across future hardware generations by exploring approaches to parallel programming such that future machines, which will be more and more parallel, can run your code faster than today's

machines. We want to help you to master parallel programming so your programs can scale up to the level of performance of new generations of machines.

p0240 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 several of 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.

1.6 ORGANIZATION OF THE BOOK

s0040

p0245 Chapter 2 reviews the history of GPU computing. It begins with a brief summary of the evolution of graphics hardware toward greater programmability and then discusses the historical GPGPU movement. Many of the current features and limitations of CUDA GPUs have their roots in these historic developments. A good understanding of these historic developments will help the reader to 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.

p0250 Chapter 3 introduces CUDA programming. This chapter relies on the fact that students have had previous experience with C programming. It first introduces CUDA as a simple, small extension to C that supports heterogeneous CPU/GPU joint computing and the widely used single-program, multiple-data (SPMD) parallel programming model. It then covers the thought processes involved in: (1) identifying the part of application programs to be parallelized, (2) isolating the data to be used by the parallelized code by using an API function to allocate memory on the parallel computing device, (3) using an API function to transfer data to the parallel computing device, (4) developing a kernel function that will be executed by individual 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. Although the objective of Chapter 3 is to teach enough concepts of the CUDA programming model so students can write a simple parallel CUDA 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 matrix–matrix multiplication to make this chapter concrete.

p0255 Chapters 4 through 7 are designed to give the readers more in-depth understanding of the CUDA programming model. Chapter 4 covers the thread organization and execution model required to fully understand the execution behavior of threads and basic performance concepts. Chapter 5 is dedicated to the special memories that can be used to hold CUDA variables for improved program execution speed. Chapter 6 introduces the major factors that contribute to the performance of a CUDA kernel function. Chapter 7 introduces the floating-point representation and concepts such as precision and accuracy. Although these chapters are based on CUDA, they help the readers build a 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 a solid footing to 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.

p0260 Chapters 8 and 9 are case studies of two real applications, which take the readers through the thought processes of parallelizing and optimizing their applications for significant speedups. For each application, we begin 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 necessary to achieve high performance. These two chapters help the readers put all the materials from the previous chapters together and prepare for their own application development projects.

p0265 Chapter 10 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 they can be done in parallel. We begin by discussing the translational process of organizing abstract scientific concepts into computational tasks, an important first step in producing quality application software, serial or parallel. The chapter then addresses 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, allowing the readers to place their knowledge in a wider context. With this chapter, the 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 the readers will be able to learn to program in any of them with the foundation gained in this book.

p0270 Chapter 11 introduces the OpenCL programming model from a CUDA programmer's perspective. The reader 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 necessary 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.

p0275 Chapter 12 offers some concluding remarks and an outlook for the future of massively parallel programming. We 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 and Further Reading

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

Khronos Group. (2009). *The OpenCL Specification Version 1.0*. Beaverton, OR: Khronos Group. (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 Programming*. Upper Saddle River, NJ: Addison-Wesley.

Message Passing Interface Forum. (2009). *MPI: A Message-Passing Interface Standard, Version 2.2*. Knoxville: University of Tennessee. (http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf).

NVIDIA. (2007). *CUDA Programming Guide*. Santa Clara, CA: NVIDIA Corp.

OpenMP Architecture Review Board. (2005). *OpenMP Application Program Interface*. (http://www.openmp.org/mp-documents/spec25.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*. Contract No. W-670-ORD-4926, U.S. Army Ordnance Department and University of Pennsylvania (reproduced in H. H. Goldstine, Ed., *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ, 1972).

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

CHAPTER

c0010 History of GPU Computing 2

CHAPTER CONTENTS

s0010 INTRODUCTION

p0060 To CUDA™ and OpenCL™ programmers, graphics processing units (GPUs) are massively parallel numeric computing processors programmed in C with extensions. One need not understand graphics algorithms or terminology in order to be able to program these processors. However, understanding the graphics heritage of these processors illuminates the strengths and weaknesses of these processors 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 central processing units (CPUs), and bandwidth-centric memory interface design. Insights into the historical developments will also likely give the reader the context needed to project the future evolution of GPUs as computing devices.

s0015 2.1 EVOLUTION OF GRAPHICS PIPELINES

p0065 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-

Programming Massively Parallel Processors: A Hands-on Approach. DOI: 10.1016/B978-0-12-381472-2.00002-7

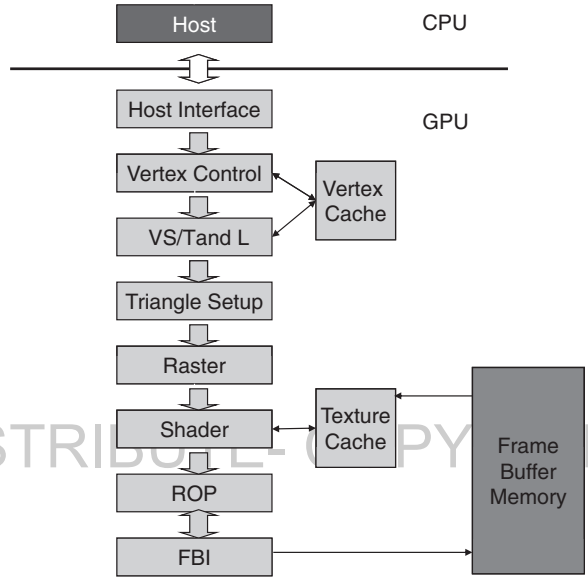
leading graphics subsystems decreased in price from $50,000 to $200. During the same period, the performance increased from 50 million pixels per second to 1 billion pixels per second and from 100,000 vertices per second to 10 million vertices per second. Although these advancements have much to do with the relentlessly shrinking feature sizes of semiconductor devices, they also have resulted from innovations in graphics algorithms and hardware design that have shaped the native hardware capabilities of modern GPUs.

p0070 The remarkable advancement of graphics hardware performance has been driven by the market demand for high-quality, real-time graphics in computer applications. In an electronic gaming application, for example, one needs to render ever more complex scenes at an 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 being a simple pipeline for drawing wire-frame diagrams to a highly parallel design consisting of several deep parallel pipelines capable of rendering the complex interactive imagery of 3D scenes. Concurrently, many of the hardware functionalities involved became far more sophisticated and user programmable.

s0020 2.1.1 The Era of Fixed-Function Graphics Pipelines

p0075 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 (i.e., a collection of library functions) that allows applications (such as games) to use software or hardware services and functionality. An API, for example, 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.

p0080 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 direct memory access (DMA) hardware to efficiently transfer



f0010 **FIGURE 2.1**

A fixed-function NVIDIA GeForce graphics pipeline.

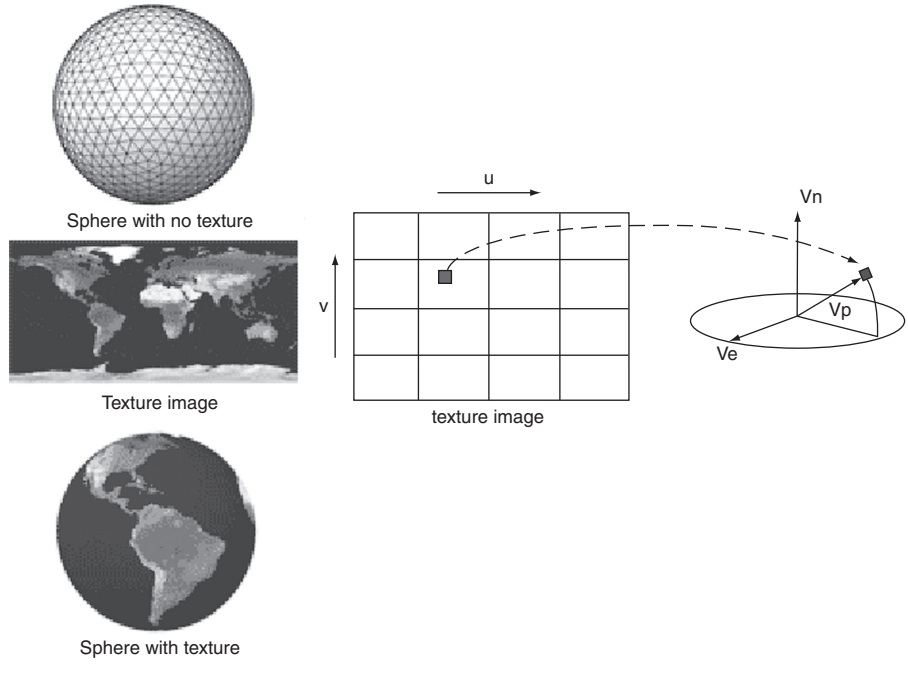
bulk data to and from the host system memory to the graphics pipeline. The host interface also communicates back the status and result data of executing the commands.

p0085 Before we describe the other stages of the pipeline, we should clarify that the term *vertex* usually refers to the corner of a polygon. The GeForce graphics pipeline is designed to render triangles, so the term *vertex* is typically used in this case 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 then converts the triangle data into a form that the hardware understands and places the prepared data into the vertex cache.

p0090 The vertex shading, transform, and lighting (VS/T&L) stage in Figure 2.1 transforms vertices and assigns per-vertex values (e.g., colors, normals, texture coordinates, tangents). The shading is done by the pixel shader hardware. The vertex shader can assign a color to each vertex, but color 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 (such as 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, including the color, position, and texture position that will be shaded (painted) on the pixel. Au1

p0095 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 must perform only a small number of coordinate transform



f0015 **FIGURE 2.2**

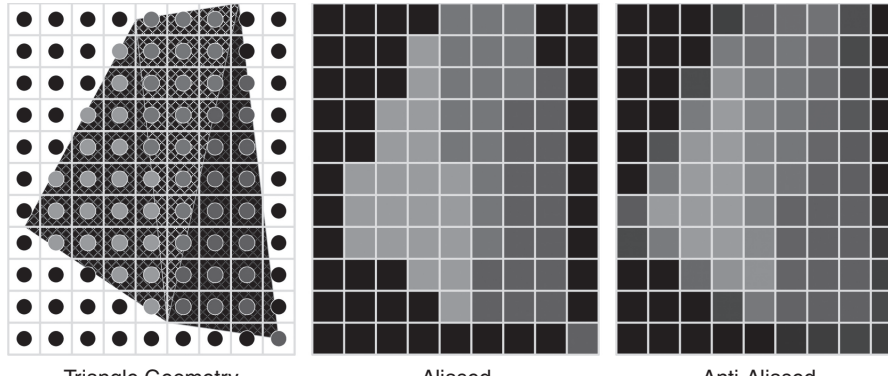
Texture mapping example: painting a world map texture image onto a globe object.

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.

p0100 The raster operation (ROP) 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 view point.

p0105 Figure 2.3 illustrates anti-aliasing, one the of ROP stage operations. Notice the three adjacent triangles with a black background. In the 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 partially overlap the pixel. The contribution of each object to the color of the pixel is the amount of the pixel that the object overlaps.

p0110 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



f0020 **FIGURE 2.3**

Example of anti-aliasing operations.

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.

p0115 For two decades, each generation of hardware and its corresponding generation of API brought incremental improvements to the various stages of the graphics pipeline. Although each generation introduced additional hardware resources and configurability to the pipeline stages, 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.

s0025 2.1.2 Evolution of Programmable Real-Time Graphics

p0120 In 2001, the NVIDIA GeForce 3 took the first step toward achieving 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's 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. The 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.

p0125 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

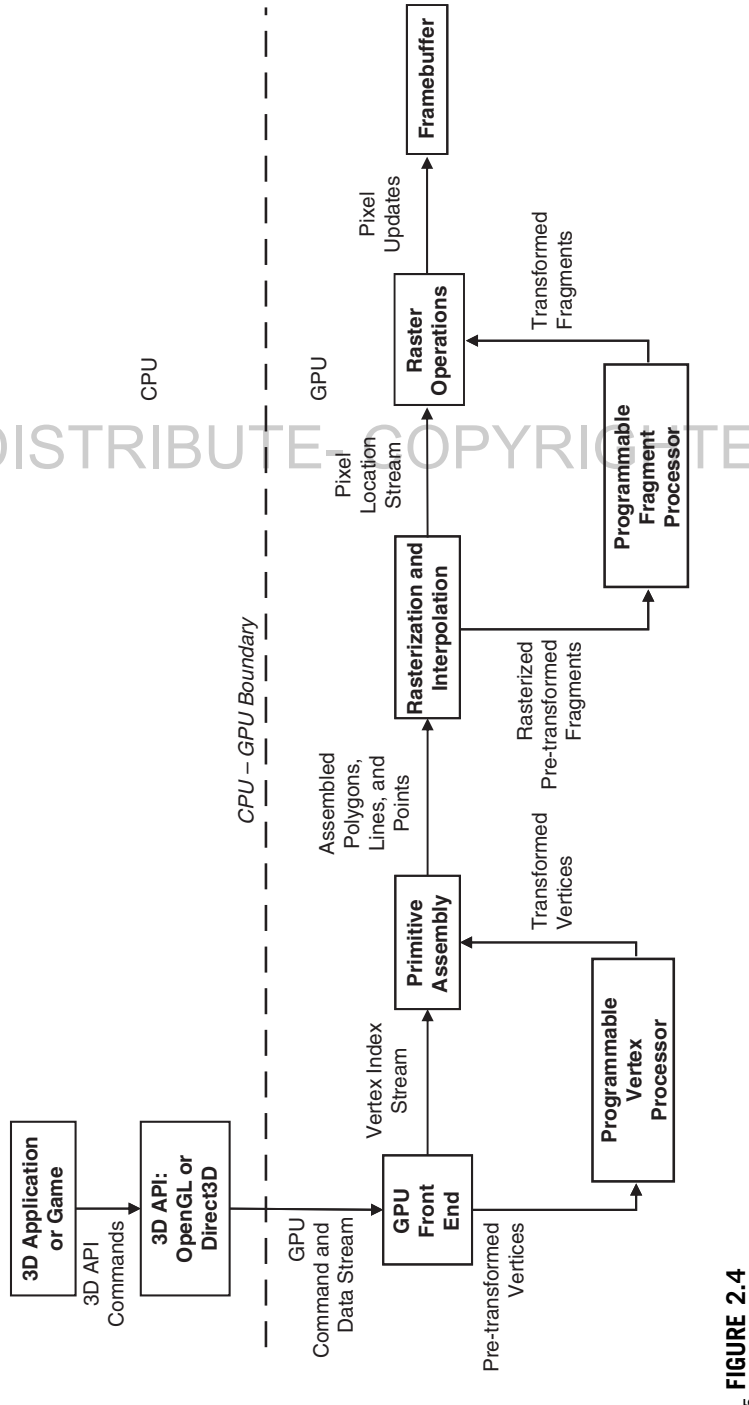
assumption for GPUs and CPUs. A single frame, rendered in 1/60th of a second, might have a million triangles and 6 million pixels. The opportunity to use hardware parallelism to exploit this data independence is tremendous.

p0130 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 vertex shader (VS/T&L) stage of the graphics pipeline.

p0135 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.

p0140 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 vertex shader stage, 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 which 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.

p0145 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



f0025 **FIGURE 2.4**
Example of a separate vertex processor and fragment processor in a programmable graphics pipeline.

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 datapath and fixed-function logic. GPU memory interfaces emphasize bandwidth over latency (as latency can be readily hidden by massively parallel execution); indeed, bandwidth is typically many times higher than that for a CPU, exceeding 100 GB/s in more recent designs.

s0030 ### 2.1.3 Unified Graphics and Computing Processors

p0150 Introduced in 2006, the 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. Because 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.

p0155 The GeForce 8800 hardware corresponds to the DirectX 10 API generation. By the DirectX 10 generation, the functionality of vertex and pixel shaders had been 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 in order to deliver the desired operation throughput as area efficiently as possible. High-clock-speed design requires substantially greater engineering effort, thus favoring the design of 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.

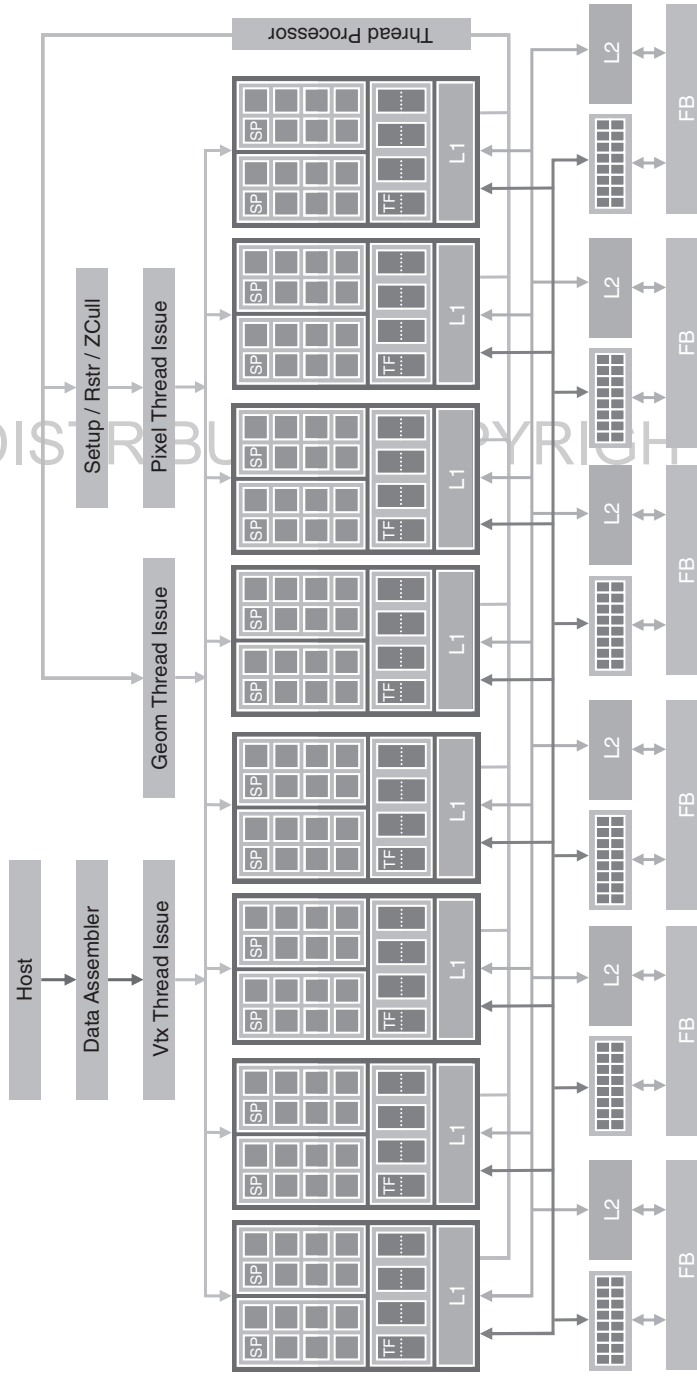


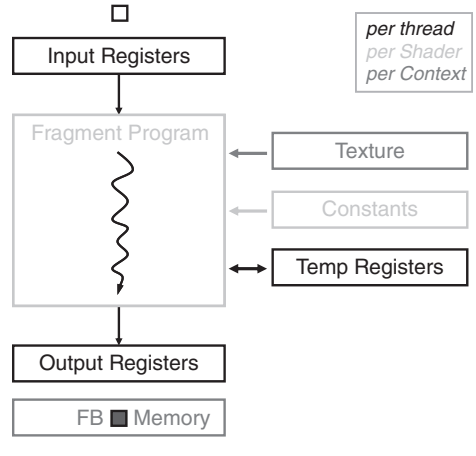
FIGURE 2.5

f0030 Unified programmable processor array of the GeForce 8800 GTX graphics pipeline.

s0035 ### 2.1.4 GPGPU: An Intermediate Step

p0160 While the GPU hardware designs evolved toward more unified processors, they 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 began 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 native graphics operations so the computation could be launched through OpenGL or DirectX API calls. To run many simultaneous instances of a compute function, for example, the computation 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 was what was desired). The output had to be cast as a set of pixels generated from the raster operations.

p0165 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 whose memory locations have been predetermined; thus, the graphics processor array is designed with very restricted memory reading and writing capability. Figure 2.6 illustrates the limited



f0035 **FIGURE 2.6**

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

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 two-dimensional frame buffer.

p0170 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 under-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.

s0040 2.2 GPU COMPUTING

p0175 While developing the Tesla™ GPU architecture, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU like a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

p0180 For the DirectX 10 generation of 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 GPU architecture took another step. The shader processors became fully programmable processors with large 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 non-graphics application programmers, the Tesla GPU architecture 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 the GeForce 8800 GTX, which was followed later by G92 and GT200.

s0045 ### 2.2.1 Scalable GPUs

p0185 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 began to appear, there was room in the market for a range of offerings. In 1998, 3dfx introduced multiboard scaling with their original Scan Line Interleave (SLI) on their Voodoo2, which held the performance crown for its time. 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 in 2001, NVIDIA continued the multi-GPU SLI concept; for example, the GeForce 6800 provides multi-GPU scalability transparently to both the programmer and the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

p0190 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- to eightfold 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 multithreaded 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.

2.2.2 Recent Developments

p0195 Academic and industrial work on applications using CUDA has produced hundreds of examples of successful CUDA programs. Many of these programs run the application tens or hundreds of times faster than multicore CPUs are capable of running them. With the introduction of tools such as MCUDA [Stratton2008], the parallel threads of a CUDA program can also run efficiently on a multicore CPU, although at a lower speed than on GPUs due to lower levels 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 arrival of double-precision floating point in GPUs enabled an even broader range of applications to benefit from GPU acceleration.

p0200 For an exhaustive list and examples of current developments in applications that are accelerated by GPUs, visit CUDA Zone at http://www.nvidia.com/CUDA. For resources in developing research applications, see CUDA Research at http://www.cudaresearch.org.

2.3 FUTURE TRENDS

p0205 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 enjoy 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 architecture to increase the actual utilization of the calculating units. Because scalable parallel computing on GPUs is still a young field, novel applications are rapidly being created. By studying them, GPU designers will discover and implement new machine optimizations. Chapter 10 provides more details of such future trends.

References and Further Reading

Akeley, K., & Jermoluk, T. (1988). High-Performance Polygon Rendering. *Computer Graphics (SIGGRAPH 88)*, *22*(4), 239–246.

Akeley, K. (1993). Reality Engine Graphics. *Computer Graphics (SIGGRAPH 93)*, *27*, 109–116.

Blelloch, G. B. (1990). Prefix Sums and Their Applications. In J. H. Reif (Ed.), *Synthesis of Parallel Algorithms*. San Francisco, CA: Morgan Kaufmann.

Blythe, D. (2006). The Direct3D 10 System. *ACM Transactions on Graphics*, *25* (3), 724–734.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahlian, K., Houston, M., et al. (2004). Brooks for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, *23*(3), 777–786 (http://doi.acm.org/10.1145/1186562.1015800).

Elder, G. (2002). Radeon 9700. In *Proceedings of the ACM Eurographics/SIGGRAPH Workshop on Graphics Hardware 2002* (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.), *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Reading, MA: Addison-Wesley (http://developer.nvidia.com/object/gpu_gems_home.html).

Foley, J., van Dam, A., Feiner, S., & Hughes, J. *Interactive Computer Graphics: Principles and Practice, C Edition* (2nd ed.). Reading, MA: Addison-Wesley.

Hillis, W. D., & Steele, G. L. (1986). Data Parallel Algorithms. *Communications of the ACM*, *29*(12), 1170–1183 (http://doi.acm.org/10.1145/7902.7903).

IEEE 754R Working Group. (2006). *Standard for Floating-Point Arithmetic P754 (Draft)*. Piscataway, NJ: Institute of Electrical and Electronics Engineers (http://www.validlab.com/754R/drafts/archive/2006-10-04.pdf).

Industrial Light and Magic. (2003). *OpenEXR*. San Mateo, CA: Industrial Light and Magic (www.openexr.com).

Intel. (2007). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Order No. 248966-016. Santa Clara, CA: Intel Corp. (http://www3.intel.com/design/processor/manuals/248966.pdf).

Kessenich, J., Baldwin, D., & Rost, R. (2006). *The OpenGL® Shading Language, Language Version 1.20*. Madison, AL: 3Dlabs, Inc. (http://www.opengl.org/documentation/specs/).

Kirk, D., & Voorhies, D. (1990). The Rendering Architecture of the DN10000VS. *Computer Graphics (SIGGRAPH 1990)*, *24*(4), 299–307.

Lindholm, E., Kilgard, M. J., & Moreton, H. (2001). A User-Programmable Vertex Engine. In *Proceedings of the 28th Annual ACM Conference on Computer Graphics and Interactive Techniques* (pp. 149–158). Reading, MA: ACM Press/Addison-Wesley.

Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, *28*(2), 39–55.

Microsoft. (2003). *Microsoft DirectX 9 Programmable Graphics Pipeline*. Redmond, WA: Microsoft Press.

Microsoft. (2009). *Microsoft DirectX Specification*. Redmond, WA: Microsoft Press (http://msdn.microsoft.com/directx/).

Montrym, J., Baum, D., Dignam, D., & Migdal, C. (1997). InfiniteReality: A Real-Time Graphics System. In G. O. Owen, T. Whitted & B. Mones-Hattal (Eds.), *Proceedings of the 24th Annual ACM Conference on Computer Graphics and Interactive Techniques* (pp. 293–301). Reading, MA: ACM Press/Addison-Wesley.

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), 114–117.

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. (2007a). *NVIDIA CUDA—Compute Unified Device Architecture, Programming Guide, Version 1.1* (http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf).

NVIDIA. (2007b). *NVIDIA Compute—PTX: Parallel Thread Execution, ISA Version 1.1* (http://www.nvidia.com/object/io_1195170102263.html).

NVIDIA. (2009). *CUDA Zone* (http://www.nvidia.com/CUDA).

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., & Siu, M. Y. (2005). A High-Performance Area-Efficient Multifunction Interpolator. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic* (pp. 272–279). Cape Cod, MA.

Patterson, D. A., & Hennessy, J. L. (2004). *Computer Organization and Design: The Hardware/Software Interface* (3rd ed.). San Francisco, CA: 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., & Garland, M. (2008). Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of the IEEE International Parallel and Distributed Symposium*. Au2

Segal, M., & Akeley, K. (2006). *The OpenGL® Graphics System: A Specification, Version 2.1*. Mountain View, CA: Silicon Graphics (http://www.opengl.org/documentation/specs/).

Sengupta, S., Harris, M., Zhang, Y., & Owens, J. D. (2007). Scan Primitives for GPU Computing. In T. Aila & M. Segal (Eds.), *Graphics Hardware* (pp. 97–106). San Diego, CA: ACM Press.

Stratton, J. A., Stone, S. S., & Hwu, W. W. (2008). MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Canada: Edmonton.

Volkov, V., & Demmel, J. (2008). *LU, QR and Cholesky Factorizations Using Vector Capabilities of GPUs*. Technical Report No. UCB/EECS-2008-49. Berkeley: EECS Department, University of California (http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html).

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., & Demmel, J. (2008). Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Parallel Computing—Special Issue on Revolutionary Technologies for Acceleration of Emerging Petascale Applications*.

CHAPTER

c0015 Introduction to CUDA 3

CHAPTER CONTENTS

s0010 INTRODUCTION

p0065 To a CUDA™ programmer, the computing system consists of a *host*, which is a traditional central processing unit (CPU), such as an Intel® architecture microprocessor in personal computers today, and one or more *devices*, which are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Because 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.

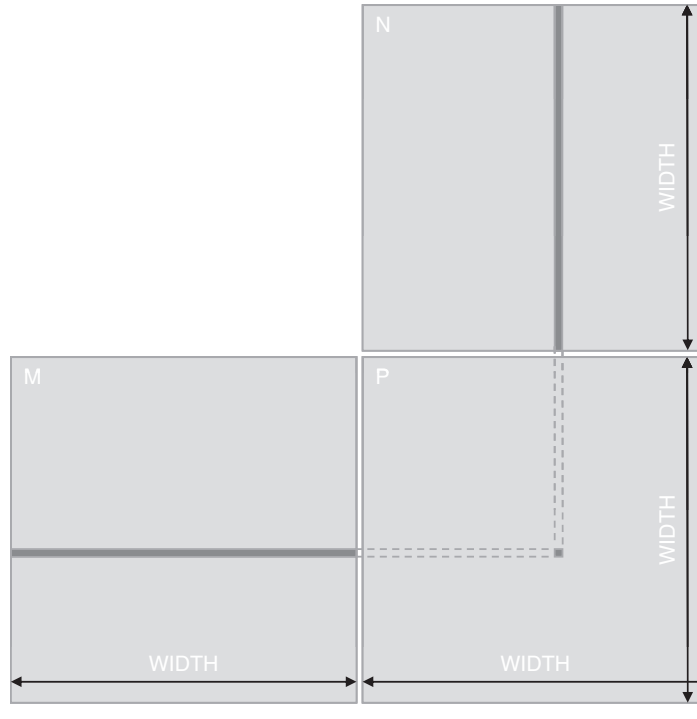
s0015 3.1 DATA PARALLELISM

p0070 Many software applications that process a large amount of data and thus incur long execution times on today's computers are designed to model real-world, physical phenomena. Images and video frames are snapshots

Programming Massively Parallel Processors: A Hands-on Approach. DOI: 10.1016/B978-0-12-381472-2.00003-9

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. Such independent evaluation is the basis of data parallelism in these applications.

p0075 As we mentioned earlier, data parallelism refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner. We illustrate the concept of data parallelism with a matrix–matrix multiplication (matrix multiplication, for brevity) example in Figure 3.1. In this example, each element of the product matrix **P** is generated by performing a dot product between a row of input matrix **M** and a column of input matrix **N**. In Figure 3.1, the highlighted element of matrix **P** is generated by taking the dot product of the highlighted row of matrix **M** and the highlighted column of matrix **N**. Note that the dot product operations for computing different matrix **P** elements can be



f0010 **FIGURE 3.1**

Data parallelism in matrix multiplication.

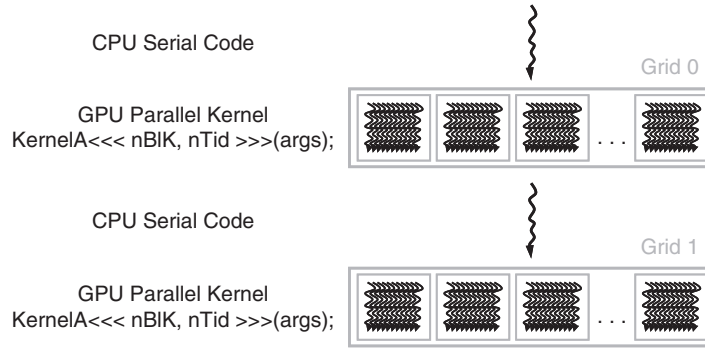
simultaneously performed. That is, none of these dot products will affect the results of each other. For large matrices, the number of dot products can be very large; for example, a 1000 × 1000 matrix multiplication has 1,000,000 independent dot products, each involving 1000 multiply and 1000 accumulate arithmetic operations. Therefore, matrix multiplication of large dimensions can have very large amount of data parallelism. By executing many dot products in parallel, a CUDA device can significantly accelerate the execution of the matrix multiplication over a traditional host CPU. The data parallelism in real applications is not always as simple as that in our matrix multiplication example. In a later chapter, we will discuss these more sophisticated forms of data parallelism.

3.2 CUDA PROGRAM STRUCTURE

s0020

p0080 A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. A CUDA program is a unified source code encompassing both host and device code. The NVIDIA® C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. The device code is typically further compiled by the nvcc and executed on a GPU device. In situations where no device is available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU using the MCUDA tool [Stratton 2008].

p0085 The kernel functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism. In the matrix multiplication example, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix **P**. In this example, the number of threads used by the kernel is a function of the matrix dimension. For a 1000 × 1000 matrix multiplication, the kernel that uses one thread to compute one **P** element would generate 1,000,000 threads when it is invoked. It is worth noting that CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically require thousands of clock cycles to generate and schedule.



f0015 **FIGURE 3.2**
Execution of a CUDA program.

p0090 The execution of a typical CUDA program is illustrated in Figure 3.2. The execution starts with host (CPU) execution. When a kernel function is invoked, or *launched*, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a *grid*. Figure 3.2 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 invoked.

s0025 3.3 A MATRIX–MATRIX MULTIPLICATION EXAMPLE

p0095 At this point, it is worthwhile to introduce a code example that concretely illustrates the CUDA program structure. Figure 3.3 shows a simple main function skeleton for the matrix multiplication example. For simplicity, we assume that the matrices are square in shape, and the dimension of each matrix is specified by the parameter `Width`.

p0100 The main program first allocates the **M**, **N**, and **P** matrices in the host memory and then performs I/O to read in **M** and **N** in Part 1. These are ANSI C operations, so we are not showing the actual code for the sake of brevity. The detailed code of the main function and some user-defined ANSI C functions is shown in Appendix A. Similarly, after completing the matrix multiplication, Part 3 of the main function performs I/O to write the product matrix **P** and to free all the allocated matrices. The details of Part 3 are also shown in Appendix A. Part 2 is the main focus of our

```
int main(void) {
1.  // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

2.  // M * N on the device
    MatrixMultiplication(M, N, P, Width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

f0020 **FIGURE 3.3**

A simple main function for the matrix multiplication example.

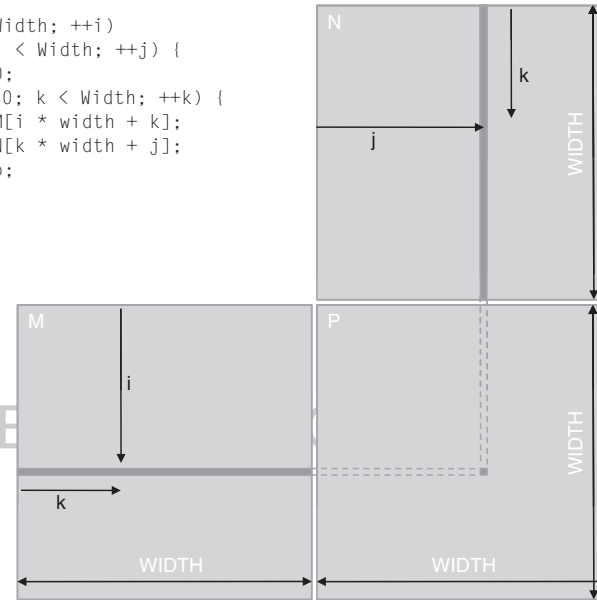
example. It calls a function, `MatrixMultiplication()`, to perform matrix multiplication on a device.

p0105 Before we explain how to use a CUDA device to execute the matrix multiplication function, it is helpful to first review how a conventional CPU-only matrix multiplication function works. A simple version of a CPU-only matrix multiplication function is shown in Figure 3.4. The `MatrixMultiplication()` function implements a straightforward algorithm that consists of three loop levels. The innermost loop iterates over variable k and steps through one row of matrix **M** and one column of matrix **N**. The loop calculates a dot product of the row of **M** and the column of **N** and generates one element of **P**. Immediately after the innermost loop, the **P** element generated is written into the output **P** matrix.

p0110 The index used for accessing the **M** matrix in the innermost loop is `i*Width+k`. This is because the **M** matrix elements are placed into the system memory that is ultimately accessed with a linear address. That is, every location in the system memory has an address that ranges from 0 to the largest memory location. For C programs, the placement of a 2-dimensional matrix into this linear addressed memory is done according to the row-major convention, as illustrated in Figure 3.5.[1] All elements of a row are placed into consecutive memory locations. The rows are then placed one after another. Figure 3.5 shows an example where a 4×4 matrix is

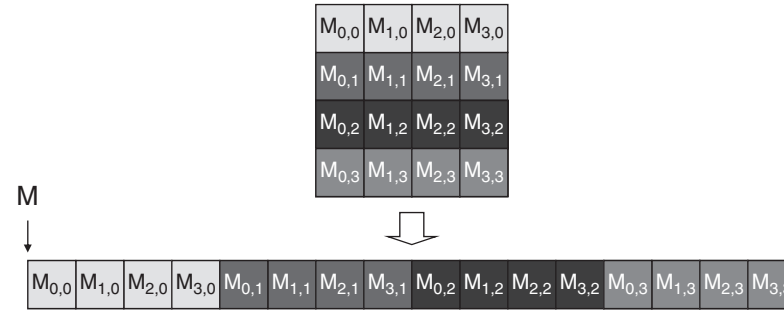
fn0010 [1]Note that FORTRAN adopts the column–major placement approach: All elements of a column are first placed into consecutive locations, and all columns are then placed in their numerical order.

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   for (int i = 0 ; i < Width; ++i)
       for (int j = 0 ; j < Width; ++j) {
           double sum = 0;
           for (int k  = 0; k < Width; ++k) {
             double a = M[i * width + k];
             double b = N[k * width + j];
             sum + =a * b;
          }
          P[i * Wid
      }
}
```



f0025 **FIGURE 3.4**

A simple matrix multiplication function with only host code.



f0030 **FIGURE 3.5**

Placement of two-dimensional array elements into the linear address system memory.

placed into 16 consecutive locations, with all elements of row 0 first followed by the four elements of row 1, etc. Therefore, the index for an **M** element in row *i* and column *k* is `i*Width+k`. The `i*Width` term skips over all elements of the rows before row *i*. The *k* term then selects the proper element within the section for row *i*.

p0115 The outer two (*i* and *j*) loops in Figure 3.4 jointly iterate over all rows of **M** and all columns of **N**; each joint iteration performs a row–column dot product to generate one **P** element. Each *i* value identifies a row. By systematically iterating all **M** rows and all **N** columns, the function generates all **P** elements. We now have a complete matrix multiplication function that executes solely on the CPU. Note that all of the code that we have shown so far is in standard C.

p0120 Assume that a programmer now wants to port the matrix multiplication function into CUDA. A straightforward way to do so is to modify the `MatrixMultiplication()` function to move the bulk of the calculation to a CUDA device. The structure of the revised function is shown in Figure 3.6. Part 1 of the function allocates device (GPU) memory to hold copies of the **M**, **N**, and **P** matrices and copies these matrices over to the device memory. Part 2 invokes a kernel that launches parallel execution of the actual matrix multiplication on the device. Part 3 copies the product matrix **P** from the device memory back to the host memory.

p0125 Note that the revised `MatrixMultiplication()` 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

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M, N, and P to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

f0035 **FIGURE 3.6**

Outline of a revised host code `MatrixMultiplication()` that moves the matrix multiplication to a device.

a way that the main program does not have to even be aware that the matrix multiplication is now actually done on a device. The details of the revised function, as well as the way to compose the kernel function, will serve as illustrations as we introduce the basic features of the CUDA programming model.

3.4 DEVICE MEMORIES AND DATA TRANSFER

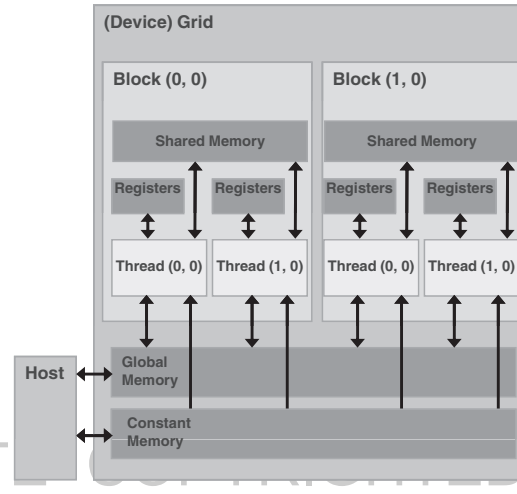
In CUDA, the host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own dynamic random access memory (DRAM). For example, the NVIDIA T10 processor comes with up to 4 GB (billion bytes, or gigabytes) of DRAM. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 of Figure 3.6. 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 3.6. 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 piece of data is transferred from the host memory to the device memory. The same holds for the opposite data transfer direction.

Figure 3.7 shows an overview of the CUDA device memory model for programmers to reason about the allocation, movement, and usage of the various memory types of a device. At the bottom of the figure, we see global memory and constant memory. These are the memories that the host code can transfer data to and from the device, as illustrated by the bidirectional arrows between these memories and the host. Constant memory allows read-only access by the device code and is described in Chapter 5. For now, we will focus on the use of global memory. Note that the host memory is not explicitly shown in Figure 3.7 but is assumed to be contained in the host.[2]

The CUDA memory model is supported by API functions that help CUDA programmers to manage data in these memories. Figure 3.8 shows the API functions for allocating and deallocating device global memory. The function `cudaMalloc()` can be called from the host code to allocate

[2]Note that we have omitted the texture memory from Figure 3.7 for simplicity. We will introduce texture memory later.

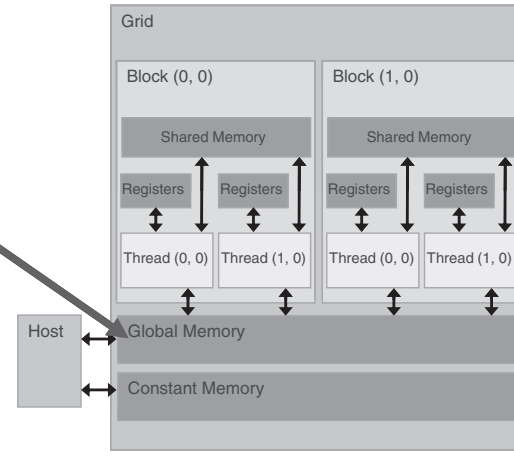
- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - Transfer data to/from per-grid global and constant memories



f0040 **FIGURE 3.7**

Overview of the CUDA device memory model.

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



f0045 **FIGURE 3.8**

CUDA API functions for device global memory management.

a piece of global memory for an object. The reader should be able to 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 needs to relearn the use of these extensions.

p0145 The first parameter of the cudaMalloc() function is the address of a pointer variable that must point to the allocated object after allocation. The address of the pointer variable should be cast to (void **) because the function expects a generic pointer value; the memory allocation function is a generic function that is not restricted to any particular type of objects. This address allows the cudaMalloc() function to write the address of the allocated object into the pointer variable.[3] The second parameter of the cudaMalloc() function gives the size of the object to be allocated, in terms of bytes. The usage of this second parameter is consistent with the size parameter of the C Malloc() function.

p0150 We now use a simple code example illustrate the use of cudaMalloc(). This is a continuation of the example in Figure 3.6. For clarity, we will end a pointer variable with the letter "d" to indicate that the variable is used to point to an object in the device memory space. The programmer passes the address of **Md** (i.e., &Md) as the first parameter after casting it to a void pointer; that is, **Md** is the pointer that points to the device global memory region allocated for the **M** matrix. The size of the allocated array will be Width*Width*4 (the size of a single-precision floating number). After the computation, cudaFree() is called with pointer **Md** as input to free the storage space for the **M** matrix from the device global memory:

```
u0015 float *Md
u0020 int size = Width * Width * sizeof(float);
u0025 cudaMalloc((void**)&Md, size);
u0030 ...
u0035 cudaFree(Md);
```

p0180 The reader should complete Part 1 of the MatrixMultiplication() example in Figure 3.6 with similar declarations of an **Nd** and a **Pd** pointer variable as

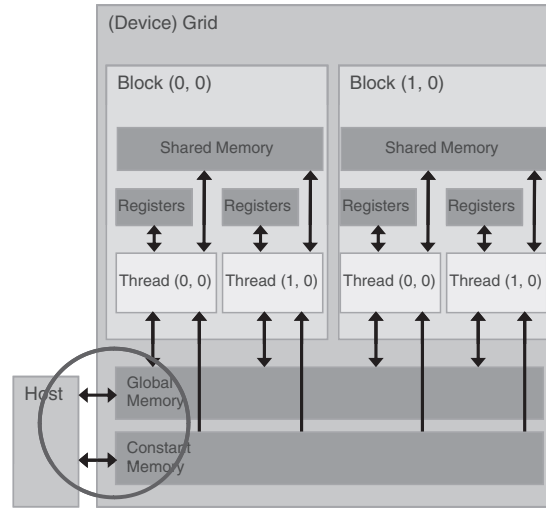
fn0020 [3]Note that cudaMalloc() has a different format from the C Malloc() function. The C Malloc() function returns a pointer to the allocated object. It takes only one parameter that specifies the size of the allocated object. The cudaMalloc() function writes to the pointer variable whose address is given as the first parameter. As a result, the cudaMalloc() function takes two parameters. The two-parameter format of cudaMalloc() allows it to use the return value to report any errors in the same way as other CUDA API functions.

well as their corresponding `cudaMalloc()` calls. Furthermore, Part 3 in Figure 3.6 can be completed with the `cudaFree()` calls for **Nd** and **Pd**.

p0185 Once a program has allocated device global memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling one of the CUDA API functions, `cudaMemcpy()`, for data transfer between memories. Figure 3.9 shows the API function for such a data transfer. The `cudaMemcpy()` function takes four parameters. The first parameter is a pointer to the source data object to be copied. The second parameter points to the destination location for the copy operation. 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 memory copy function can be used to copy data from one location of the device memory to another location of the device memory. Please note that `cudaMemcpy` cannot be used to copy between different GPUs in multi-GPU systems. Au1

p0190 For the matrix multiplication example, the host code calls the `cudaMemcpy()` function to copy the **M** and **N** matrices from the host memory to the device memory before the multiplication and then to copy the **P** matrix from the device memory to the host memory after the multiplication is done.

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Transfer is asynchronous



f0050 **FIGURE 3.9**

CUDA API functions for data transfer between memories.

Assume that **M**, **P**, **Md**, **Pd**, and size have already been set as we discussed before; the two function calls are shown below. Note that the two symbolic constants, `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`, are recognized, predefined constants of the CUDA programming environment. 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:

u0040
```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
```
u0045
```
cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```

p0205 To summarize, the main program in Figure 3.3 calls `MatrixMultiplication()`, which is also executed on the host. `MatrixMultiplication()`, as outlined in Figure 3.6, is responsible for allocating device memory, performing data transfers, and activating the kernel that performs the actual matrix multiplication. We often refer to this type of host code as the *stub function* for invoking a kernel. After the matrix multiplication, `MatrixMultiplication()` also copies result data from device to the host. We show a more fleshed out version of the `MatrixMultiplication()` function in Figure 3.10.

p0210 Compared to Figure 3.6, the revised `MatrixMultiplication()` function is complete in Part 1 and Part 3. Part 1 allocates device memory for **Md**,

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;

1. // Transfer M and N to device memory
  cudaMalloc(Md, size);
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMalloc(Nd, size);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

  // Allocate P on the device
  cudaMalloc(Pd, size);

2. // Kernel invocation code - to be shown later
  ...
3. // Transfer P from device to host
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
  // Free device matrices
  cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
  }
```

f0055 **FIGURE 3.10**

The revised `MatrixMultiplication()` function.

Nd, and **Pd**, the device counterparts of **M**, **N**, and **P**, and transfers **M** to **Md** and **N** to **Nd**. This is accomplished with calls to the `cudaMalloc()` and `cudaMemcpy()` functions. The readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Figure 3.10. Part 2 invokes the kernel and will be described in the following text. Part 3 reads the product data from device memory to host memory so the value will be available to `main()`. This is accomplished with a call to the `cudaMemcpy()` function. It then frees **Md**, **Nd**, and **Pd** from the device memory, which is accomplished with calls to the `cudaFree()` functions.

3.5 KERNEL FUNCTIONS AND THREADING

We are now ready to discuss more about the CUDA kernel functions and the effect of invoking these kernel functions. In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Because all of these threads execute the same code, CUDA programming is an instance of the well-known single-program, multiple-data (SPMD) parallel programming style [Atallah 1998], a popular programming style for massively parallel computing systems.[4]

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

In general, CUDA extends C function declarations with three qualifier keywords. The meanings of these keywords are summarized in Figure 3.12. The `__global__` keyword indicates that the function being declared is a CUDA kernel function. The function will be executed on the device and can only be called from the host to generate a grid of threads on a device. We will show the host code syntax for calling a kernel function later in Figure 3.14. The `__device__` keyword indicates that the function being declared is a CUDA device function. A device function executes on a

[4]Note that SPMD is not the same as single-instruction, multiple-data (SIMD). In an SPMD system, the parallel processing units execute the same program on multiple parts of the data; however, these processing units do not have 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.

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```


f0060 **FIGURE 3.11**

The matrix multiplication kernel function.

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

f0065 **FIGURE 3.12**

CUDA extensions to C functional declaration.

CUDA device and can only be called from a kernel function or another device function. Device functions can have neither recursive function calls nor indirect function calls through pointers in them. 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, as 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 default into host functions spares the programmer the tedious work of changing all original function declarations.

p0230 Note that one can use both `__host__` and `__device__` in a function declaration. This combination triggers the compilation system to generate two versions of 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 when the same function source code can be simply recompiled to generate a device version. Many user library functions will likely fall into this category.

p0235 Other notable extensions of ANSI C, in this example, are the keywords `threadIdx.x` and `threadIdx.y`, which refer to the thread indices of a thread. Note that all threads execute the same kernel code. There needs to be a mechanism to allow them to distinguish themselves and direct themselves toward the particular parts of the data structure that they are designated to work on. These keywords identify predefined variables that allow a thread to access the hardware registers at runtime that provide the identifying coordinates to the thread. Different threads will see different values in their `threadIdx.x` and `threadIdx.y` variables. For simplicity, we will refer to a thread as $Thread_{threadIdx.x,\ threadIdx.y}$. Note that the coordinates reflect a multidimensional organization for the threads. We will come back to this point soon.

p0240 A quick comparison of Figure 3.4 and Figure 3.11 reveals an important insight for CUDA kernel functions and CUDA kernel invocation. The kernel function in Figure 3.11 has only one loop, which corresponds to the innermost loop in Figure 3.4. The readers should ask where the other two levels of outer loops go. The answer is that the outer two loop levels are now replaced with the grid of threads. The entire grid forms the equivalent of the two-level loop. Each thread in the grid corresponds to one of the iterations of the original two-level loop. The original loop variables *i* and *j* are now replaced with *threadIdx.x* and *threadIdx.y*. Instead of having the loop increment the values of *i* and *j* for use in each loop iteration, the CUDA threading hardware generates all of the *threadIdx.x* and *threadIdx.y* values for each thread.

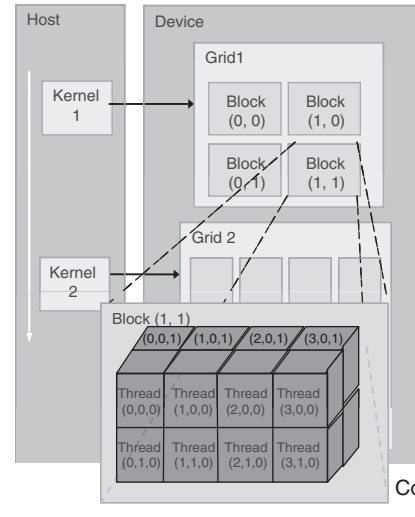
p0245 In Figure 3.11, each thread uses the *threadIdx.x* and *threadIdx.y* to identify the row of **Md** and the column of **Nd** to perform the dot product operation. It should be clear that they simply take over the role of variables *i* and *j* in Figure 3.8. Note that we assigned *threadIdx.x* to the automatic C variable *tx* and *threadIdx.y* to variable *ty* for brevity in Figure 3.8. Each thread

also uses its *threadIdx.x* and *threadIdx.y* values to select the **Pd** element that it is responsible for; for example, $Thread_{2,3}$ will perform a dot product between row 2 of **Md** and column 3 of **Nd** and write the result into element (2,3) of **Pd**. This way, the threads collectively generate all the elements of the **Pd** matrix.

p0250 When a kernel is invoked, or *launched*, it is executed as *grid* of parallel threads. In Figure 3.13, the launch of Kernel 1 creates Grid 1. Each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example, each element of a large array might be computed in a separate thread.

p0255 Threads in a grid are organized into a two-level hierarchy, as illustrated in Figure 3.13. For simplicity, a small number of threads are shown in Figure 3.13. In reality, a grid will typically consist of many more threads. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. In Figure 3.13, Grid 1is organized as a 2×2 array of 4 blocks. Each block has a unique two-dimensional coordinate given by the CUDA specific keywords `blockIdx.x` and `blockIdx.y`. All thread blocks must have the same number of threads organized in the same manner.

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

f0070 **FIGURE 3.13**

CUDA thread organization.

p0260 Each thread block is in turn organized as a three-dimensional array of threads with a total size of up to 512 threads. The coordinates of threads in a block are uniquely defined by three thread indices: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. Not all applications will use all three dimensions of a thread block. In Figure 3.12, each thread block is organized into a 4×2×2 three-dimensional array of threads. This gives Grid 1 a total of 4*16 = 64 threads. This is obviously a simplified example.

p0265 In the matrix multiplication example, a grid is invoked to compute the product matrix. The code in Figure 3.11 does not use any block index in accessing input and output data. Threads with the same `threadIdx` values from different blocks would end up accessing the same input and output data elements. As a result, the kernel can use only one thread block. The `threadIdx.x` and `threadIdx.y` values are used to organize the block into a two-dimensional array of threads. Because a thread block can have only up to 512 threads, and each thread calculates one element of the product matrix in Figure 3.11, the code can only calculate a product matrix of up to 512 elements. This is obviously not acceptable. As we explained before, the product matrix must have millions of elements in order to have a sufficient amount of data parallelism to benefit from execution on a device. We will address this issue in Chapter 4 using multiple blocks.

p0270 When the host code invokes a kernel, it sets the grid and thread block dimensions via *execution configuration* parameters. This is illustrated in Figure 3.14. Two `struct` variables of type `dim3` are declared. The first is for describing the configuration of blocks, which are defined as 16×16 groups of threads. The second variable, `dimGrid`, describes the configuration of the grid. In this example, we only have one (1×1) block in each grid. The final line of code invokes the kernel. The special syntax between the name of the kernel function and the traditional C parameters of the function is a CUDA extension to ANSI C. It provides the dimensions of grids in terms of number of blocks and the dimensions of blocks in terms of number of threads.

```
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

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

f0075 **FIGURE 3.14**

Example of host code that launches a kernel.

s0040 3.6 SUMMARY

p0275 This chapter serves as a quick overview of the CUDA programming model. CUDA extends the C language to support parallel computing. The extensions discussed in this chapter are summarized below.

s0045 3.6.1 Function Declarations

p0280 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. If both `__host__` Au2 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.

s0050 3.6.2 Kernel Launch

p0285 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. The reader should refer to the *CUDA Programming Guide* [NVIDIA 2007] for more details regarding the kernel launch extensions as well as other types of execution configuration parameters.

s0055 3.6.3 Predefined Variables

p0290 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` variable in this chapter. In Chapter 4, we will further discuss `blockIdx`, `gridDim`, and Au3 `blockDim` variables.

s0060 3.6.4 Runtime API

p0295 CUDA supports a set of application programming interface (API) functions to provide services to CUDA programs. The services that we discussed in this chapter are `cudaMalloc()` and `cudaMemcpy()` functions. These

functions allocate device memory and transfer data between the host and device on behalf of the calling program. The reader is referred to the *CUDA Programming Guide* [NVIDIA 2007] for other CUDA API functions.

p0300 Our goal for this chapter was to introduce the fundamental 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 rest of the book; however, our emphasis will be on key concepts rather than details. In general, we would like to encourage the reader to always consult the *CUDA Programming Guide* for more details on the concepts that we cover.

References and Further Reading

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

NVIDIA. (2007). *CUDA Programming Guide*. Santa Clara, CA: NVIDIA.

Stratton, J. A., Stone, S. S., & Hwu, W. W. (2008). MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. In *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Canada: Edmonton.