CHAPTER

1 Introduction

CHAPTER OUTLINE

Microprocessors based on a single central processing unit (CPU), such as those in the Intel Pentium family and the AMD Opteron family, drove rapid performance increases and cost reductions in computer applications for more than two decades. These microprocessors brought giga floating-point operations per second (GFLOPS, or Giga (10^9) Floating-Point Operations per Second), to the desktop and tera floating-point operations per second (TFLOPS, or Tera (10^{12}) Floating-Point Operations per Second) to datacenters. This relentless drive for performance improvement has allowed application software to provide more functionality, have better user interfaces, and generate more useful results. The users, in turn, demand even more improvements once they become accustomed to these improvements, creating a positive (virtuous) cycle for the computer industry.

This drive, however, has slowed since 2003 due to energy consumption and heat dissipation issues that limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Since then, virtually all microprocessor vendors have switched to models where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power. This switch has exerted a tremendous impact on the software developer community [Sutter 2005].

Traditionally, the vast majority of software applications are written as sequential programs that are executed by processors whose design was envisioned by von Neumann in his seminal report in 1945 [vonNeumann 1945]. The execution of these

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00001-7

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 Such expectation is no longer valid from this day onward.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

Rather, the applications software that will continue to enjoy significant performance improvement with each new generation of microprocessors will be parallel programs, in which multiple threads of execution cooperate to complete the work faster. This new, dramatically escalated 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 typically ran on large scale, expensive computers. Only a few elite applications could justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers. Now that all new microprocessors are parallel computers, the number of applications that need to be developed as parallel programs has increased dramatically. There is now a great need for software developers to learn about parallel programming, which is the focus of this book.

1.1 HETEROGENEOUS PARALLEL COMPUTING

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessors [Hwu 2008]. The *multicore* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multicores began with two-core processors with the number of cores increasing with each semiconductor process generation. A current exemplar is a recent *Intel* multicore microprocessor with up to 12 processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full X86 instruction set, supporting hyper-threading with two hardware threads, designed to maximize the execution speed of sequential programs. For more discussion of CPUs, see https://en.wikipedia.org/wiki/Central_processing_unit.

In contrast, the *many-thread* trajectory focuses more on the execution throughput of parallel applications. The many-threads began with a large number of threads and once again, the number of threads increases with each generation. A current exemplar is the NVIDIA Tesla P100 graphics processing unit (GPU) with 10 s of 1000 s of threads, executing in a large number of simple, in order pipelines. Many-thread processors, especially the GPUs, have led the race of floating-point performance

since 2003. As of 2016, the ratio of peak floating-point calculation throughput between many-thread GPUs and multicore CPUs is about 10, and this ratio has been roughly constant for the past several years. These are not necessarily application speeds, but are merely the raw speed that the execution resources can potentially support in these chips. For more discussion of GPUs, see https://en.wikipedia.org/wiki/Graphics_processing_unit.

Such a large performance gap between parallel and sequential execution has amounted to a significant "electrical potential" build-up, and at some point, something will have to give. We have reached that point. To date, this large performance gap has already motivated many applications developers to move the computationally intensive parts of their software to GPU for execution. Not surprisingly, these computationally intensive parts are also the prime target of parallel programming—when there is more work to do, there is more opportunity to divide the work among cooperating parallel workers.

One might ask why there is such a large peak throughput gap between many-threaded 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 Fig. 1.1. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation throughput. As of 2016, the high-end general-purpose multicore microprocessors typically have eight or more large processor cores and many megabytes of on-chip cache memories designed to deliver strong sequential code performance.

Memory bandwidth is another important issue. The speed of many applications is limited by the rate at which data can be delivered from the memory system into the processors. Graphics chips have been operating at approximately 10x the memory bandwidth of contemporaneously available CPU chips. A GPU must be capable of moving extremely large amounts of data in and out of its main Dynamic Random

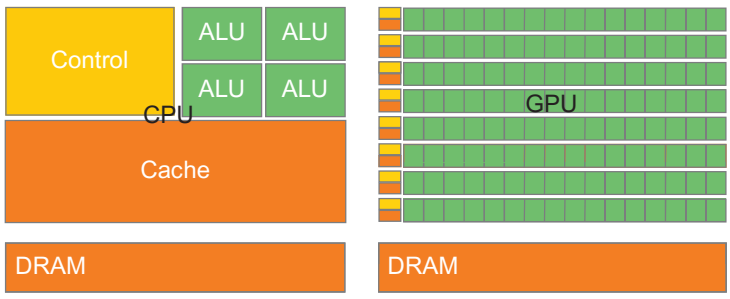


FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies.

Access Memory (DRAM) because of graphics frame buffer requirements. In contrast, general-purpose processors have to satisfy requirements from legacy operating systems, applications, and I/O devices that make memory bandwidth more difficult to increase. As a result, we expect that CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

The design philosophy of the GPUs has been shaped by the fast growing video game industry that exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced games. This demand motivates GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. An important observation is that reducing latency is much more expensive than increasing throughput in terms of power and chip area. Therefore, the prevailing solution is to optimize for the execution throughput of massive numbers of threads. The design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long-latency. The reduced area and power of the memory access hardware and arithmetic units allows the designers to have more of them on a chip and thus increase the total execution throughput.

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

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

It should be clear now that GPUs are designed as parallel, throughput-oriented computing engines and they will not perform well on some tasks on which CPUs are designed to perform well. For programs that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance than GPUs. When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance than CPUs. Therefore, one should expect that many applications use both CPUs and GPUs, executing the sequential parts on the

CPU and numerically intensive parts on the GPUs. This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPU–GPU execution of an application.[1] The demand for supporting joint CPU–GPU execution is further reflected in more recent programming models such as OpenCL (Appendix A), OpenACC (see chapter: Parallel programming with OpenACC), and C++AMP (Appendix D).

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

Another important decision factor is practical form factors and easy accessibility. Until 2006, parallel software applications 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. However, real-world clinical applications on MRI machines utilize 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 computer server boxes into clinical settings, while this is common in academic departmental settings. In fact, NIH refused to fund parallel programming projects for some time; they felt that the impact of parallel software would be limited because huge cluster-based machines would not work in the clinical setting. Today, many companies ship MRI products with GPUs, and NIH funds research using GPU computing.

Yet another important consideration in selecting a processor for executing numeric computing applications is the level of support for IEEE Floating-Point Standard. The standard enables predictable results across processors from different vendors. While the support for the IEEE Floating-Point Standard was not strong in early GPUs, this has also changed for new generations of GPUs since 2006. As we will discuss in Chapter 6, Numerical considerations, GPU support for the IEEE Floating-Point Standard has become comparable with that of the CPUs. As a result, one can expect

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

that more numerical applications will be ported to GPUs and yield comparable result values as the CPUs. Up to 2009, a major barrier was that the GPU floating-point arithmetic units were 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 only high-end CPU cores achieve. This makes the GPUs suitable for even more numerical applications. In addition, GPUs support Fused Multiply-Add, which reduces errors due to multiple rounding operations.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphics application programming interface (API) functions to access the processing units, meaning that OpenGL or Direct3D techniques were needed to program these chips. Stated more simply, a computation must be expressed as a function that paints a pixel in some way in order to execute on these early GPUs. This technique was called GPGPU, for General-Purpose Programming using a GPU. Even with a higher level programming environment, the underlying code still needs to fit into the APIs that are designed to paint pixels. These APIs limit the kinds of applications that one can actually write for early GPUs. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent research results.

But everything changed in 2007 with the release of CUDA [NVIDIA 2007]. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming, so this did not represent software changes 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. The general-purpose programming interface greatly expands the types of applications that one can easily develop for GPUs. Moreover, all the other software layers were redone as well, so that the programmers can use the familiar C/C++ programming tools. Some of our students tried to do their lab assignments using the old OpenGL-based programming interface, and their experience helped them to greatly appreciate the improvements that eliminated the need for using the graphics APIs for general-purpose computing applications.

1.2 ARCHITECTURE OF A MODERN GPU

Fig. 1.2 shows a high level view of the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Fig. 1.2, two SMs form a building block. However, the number of SMs in a building block can vary from one generation to another. Also, in Fig. 1.2, each SM has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with gigabytes of Graphics Double Data Rate (GDDR), Synchronous DRAM (SDRAM), referred to as Global Memory in Fig. 1.2. These

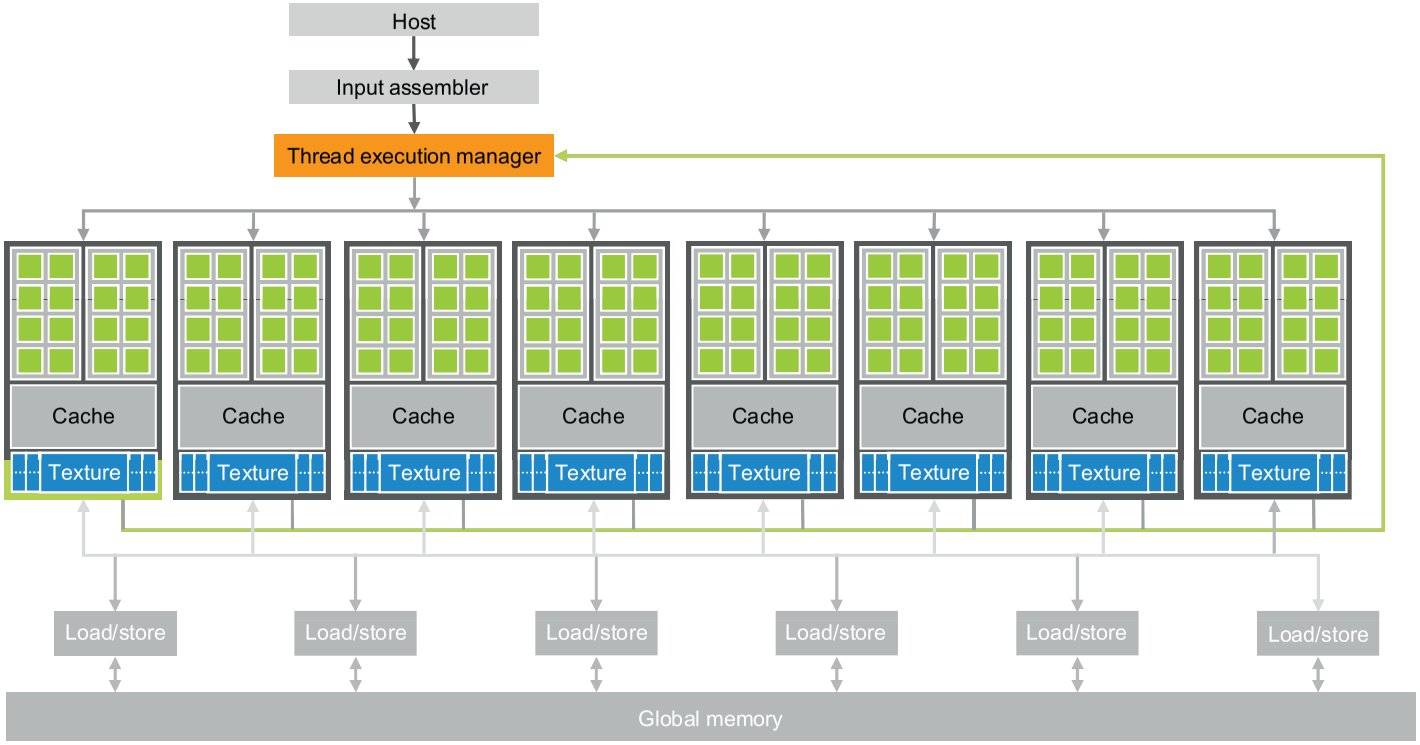


FIGURE 1.2

Architecture of a CUDA-capable GPU.

GDDR SDRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images and texture information for 3D rendering. For computing, they function as very high-bandwidth off-chip memory, though with somewhat longer latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency. More recent products, such as NVIDIA's Pascal architecture, may use High-Bandwidth Memory (HBM) or HBM2 architecture. For brevity, we will simply refer to all of these types of memory as DRAM for the rest of the book.

The G80 introduced the CUDA architecture and had a communication link to the CPU core logic over a PCI-Express Generation 2 (Gen2) interface. Over PCI-E Gen2, a CUDA application can transfer data from the system memory to the global memory at 4 GB/S, and at the same time upload data back to the system memory at 4 GB/S. Altogether, there is a combined total of 8 GB/S. More recent GPUs use PCI-E Gen3 or Gen4, which supports 8–16 GB/s in each direction. The Pascal family of GPUs also supports NVLINK, a CPU–GPU and GPU–GPU interconnect that allows transfers of up to 40 GB/s per channel. As the size of GPU memory grows, applications increasingly keep their data in the global memory and only occasionally use the PCI-E or NVLINK to communicate with the CPU system memory if there is need for using a library that is only available on the CPUs. The communication bandwidth is also expected to grow as the CPU bus bandwidth of the system memory grows in the future.

A good application typically runs 5000 to 12,000 threads simultaneously on this chip. For those who are used to multithreading in CPUs, note that Intel CPUs support 2 or 4 threads, depending on the machine model, per core. CPUs, however, are increasingly using Single Instruction Multiple Data (SIMD) instructions for high numerical performance. The level of parallelism supported by both GPU hardware and CPU hardware is increasing quickly. It is therefore very important to strive for high levels of parallelism when developing computing applications.

1.3 WHY MORE SPEED OR PARALLELISM?

As we stated in Section 1.1, the main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations. One might question if applications will continue to demand increased speed. Many applications that we have today seem to be running fast enough. As we will discuss in the case study chapters (see chapters: Application case study—non-Cartesian MRI, Application case study—molecular visualization and analysis, and Application case study—machine learning), when an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100 times (100x) speedup over sequential execution on a single CPU core. If the application contains what we call "data parallelism," it is often possible to achieve a 10x speedup with just a few hours of work. For anything beyond that, we invite you to keep reading!

Despite the myriad of computing applications in today's world, many exciting mass market applications of the future are what we previously consider "supercomputing applications," or super-applications. For example, the biology research community is moving more and more into the molecular-level. Microscopes, arguably the most important instrument in molecular biology, used to rely on optics or electronic instrumentation. But there are limitations to the molecular-level observations that we can make with these instruments. These limitations can be effectively addressed by incorporating a computational model to simulate the underlying molecular activities with boundary conditions set by traditional instrumentation. With simulation we can measure even more details and test more hypotheses than can ever be imagined with traditional instrumentation alone. These simulations will continue to benefit from the increasing computing speed in the foreseeable future in terms of the size of the biological system that can be modeled and the length of reaction time that can be simulated within a tolerable response time. These enhancements will have tremendous implications for science and medicine.

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

User interfaces can also be improved by improved computing speeds. Modern smart phone users enjoy a more natural interface with high-resolution touch screens that rival that of large-screen televisions. Undoubtedly future versions of these devices will incorporate sensors and displays with three-dimensional perspectives, applications that combine virtual and physical space information for enhanced usability, and voice and computer vision-based interfaces, requiring even more computing speed.

Similar developments are underway in consumer electronic gaming. In the past, driving a car in a game was in fact simply a prearranged set of scenes. If the player's car collided with obstacles, the behavior of the car did not change to reflect the damage. Only the game score changes—and the score determines the winner. The car would drive the same—despite the fact that the wheels should be bent or damaged. With increased computing speed, the races can actually proceed according to simulation instead of approximate scores and scripted sequences. We can expect to see more of these realistic effects in the future: collisions will damage your wheels and the player's driving experience will be much more realistic. Realistic modeling and simulation of physics effects are known to demand very large amounts of computing power.

All the new applications that we mentioned involve simulating a physical, concurrent world in different ways and at different levels, with tremendous amounts of data being processed. In fact, the problem of handling massive amounts of data is

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

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

1.4 SPEEDING UP REAL APPLICATIONS

What kind of speedup can we expect from parallelizing an application? It depends on the portion of the application that can be parallelized. If the percentage of time spent in the part that can be parallelized is 30%, a 100X speedup of the parallel portion will reduce the execution time by no more than 29.7%. The speedup for the entire application will be only about 1.4X. In fact, even infinite amount of speedup in the parallel portion can only slash 30% off execution time, achieving no more than 1.43X speedup. The fact that the level of speedup one can achieve through parallel execution can be severely limited by the parallelizable portion of the application is referred to as Amdahl's Law. On the other hand, if 99% of the execution time is in the parallel portion, a 100X speedup of the parallel portion will reduce the application execution to 1.99% of the original time. This gives the entire application a 50X speedup. Therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speed up its execution.

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

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

Fig. 1.3 illustrates the main parts of a typical application. Much of a real application's code tends to be sequential. These sequential parts are illustrated as the "pit" area of the peach: trying to apply parallel computing techniques to these portions is like biting into the peach pit—not a good feeling! These portions are very hard to parallelize. CPUs are pretty good with 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.

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

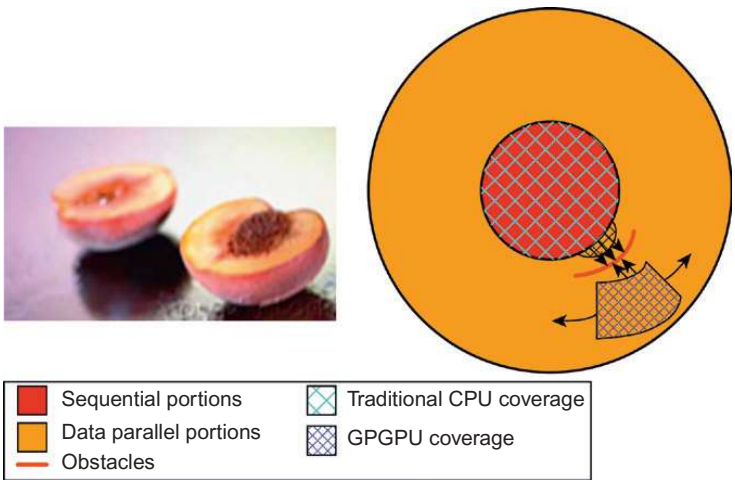


FIGURE 1.3

Coverage of sequential and parallel application portions.

1.5 CHALLENGES IN PARALLEL PROGRAMMING

What makes parallel programming hard? 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 then why bother to write a parallel program if you do not care about performance?

This book addresses several challenges in achieving high-performance in parallel programming. First and foremost, it can be challenging to design parallel algorithms with the same level of algorithmic (computational) complexity as sequential algorithms. Some parallel algorithms can add large overheads over their sequential counter parts so much that they can even end up running slower for larger input data sets.

Second, the execution speed of many applications is limited by memory access speed. We refer to these applications as memory-bound, as opposed to compute bound, which are limited by the number of instructions performed per byte of data. Achieving high-performance parallel execution in memory-bound applications often requires novel methods for improving memory access speed.

Third, the execution speed of parallel programs is often more sensitive to the input data characteristics than their sequential counter parts. Many real world applications need to deal with inputs with widely varying characteristics, such as erratic or unpredictable data rates, and very high data rates. The performance of parallel programs can sometimes vary dramatically with these characteristics.

Fourth, many real world problems are most naturally described with mathematical recurrences. Parallelizing these problems often requires nonintuitive ways of thinking about the problem and may require redundant work during execution.

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

1.6 PARALLEL PROGRAMMING LANGUAGES AND MODELS

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 message passing interface (MPI) [MPI 2009] for scalable cluster computing, and OpenMP [Open 2005] for shared memory multiprocessor systems. Both have become standardized programming interfaces supported by major computer vendors. An OpenMP implementation consists of a compiler and a runtime. A programmer specifies directives (commands) and pragmas (hints) about a loop to the OpenMP compiler. With these directives and pragmas, OpenMP compilers generate parallel code. The runtime system supports the execution of the parallel code by managing parallel threads and resources. OpenMP was originally designed for CPU execution. More recently, a variation called OpenACC (see chapter: Parallel programming with OpenACC)

has been proposed and supported by multiple computer vendors for programming heterogeneous computing systems.

The major advantage of OpenACC is that it provides compiler automation and runtime support for abstracting away many parallel programming details from programmers. Such automation and abstraction can help make the application code more portable across systems produced by different vendors, as well as different generations of systems from the same vendor. We can refer to this property as "performance portability." This is why we teach OpenACC programming in Chapter 19, Parallel programming with OpenACC. However, effective programming in OpenACC still requires the programmers to understand all the detailed parallel programming concepts involved. Because CUDA gives programmers explicit control of these parallel programming details, it is an excellent learning vehicle even for someone who would like to use OpenMP and OpenACC as their primary programming interface. Furthermore, from our experience, OpenACC compilers are still evolving and improving. Many programmers will likely need to use CUDA style interfaces for parts where OpenACC compilers fall short.

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 high-performance computing (HPC). Applications written in MPI have run successfully on cluster computing systems with more than 100,000 nodes. Today, many HPC clusters employ heterogeneous CPU/GPU nodes. While CUDA is an effective interface with each node, most application developers need to use MPI to program at the cluster level. It is therefore important that a parallel programmer in HPC understands how to do joint MPI/CUDA programming, which is presented in Chapter 18, Programming a Heterogeneous Computing Cluster.

The amount of effort needed to port an application into MPI, however, can be quite high due to lack of shared memory across computing nodes. The programmer needs to perform domain decomposition to partition the input and output data into cluster nodes. Based on the domain decomposition, the programmer also needs to call message sending and receiving functions to manage the data exchange between nodes. CUDA, on the other hand, provides shared memory for parallel execution in the GPU to address this difficulty. As for CPU and GPU communication, CUDA previously provided very limited shared memory capability between the CPU and the GPU. The programmers needed to manage the data transfer between CPU and GPU in a manner similar to the "one-sided" message passing. New runtime support for global address space and automated data transfer in heterogeneous computing systems, such as GMAC [GCN 2010], are now available. With such support, a CUDA programmer can declare variables and data structures as shared between CPU and GPU. The runtime hardware and software transparently maintains coherence by automatically performing optimized data transfer operations on behalf of the programmer as needed. Such support significantly reduces the programming complexity involved in overlapping data transfer with computation and I/O activities. As will be discussed later in Chapter 20, More on CUDA and GPU Computing, the Pascal architecture supports both a unified global address space and memory.

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

Those who are familiar with both OpenCL and CUDA know that there is a remarkable similarity between the key concepts and features of OpenCL and those of CUDA. That is, a CUDA programmer can learn OpenCL programming with minimal effort. More importantly, virtually all techniques learned using CUDA can be easily applied to OpenCL programming. Therefore, we introduce OpenCL in Appendix A and explain how one can apply the key concepts in this book to OpenCL programming.

1.7 OVERARCHING GOALS

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. Therefore, we are going to dedicate many pages to techniques for developing *high-performance* parallel programs. And, we believe that it will become easy once you develop the right insight and go about it the right way. In particular, we will focus on *computational thinking* [Wing 2006] techniques that will enable you to think about problems in ways that are amenable to high-performance parallel computing.

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

Our second goal is to teach parallel programming for correct functionality and reliability, which constitutes a subtle issue in parallel computing. Those who have worked on parallel systems in the past know that achieving initial performance is not enough. The challenge is to achieve it in such a way that you can debug the code and

support users. The CUDA programming model encourages the use of simple forms of barrier synchronization, memory consistency, and atomicity for managing parallelism. In addition, it provides an array of powerful tools that allow one to debug not only the functional aspects but also the performance bottlenecks. We will show that by focusing on data parallelism, one can achieve high performance without sacrificing the reliability of their applications.

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

Still, much technical knowledge will be required to achieve these goals, so we will cover quite a few principles and patterns [Mattson 2004] of parallel programming in this book. We will not be teaching these principles and patterns in a vacuum. We will teach them in the context of parallelizing useful applications. We cannot cover all of them, however, we have selected what we found to be 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.

1.8 ORGANIZATION OF THE BOOK

Chapter 2, Data parallel computing, introduces data parallelism and CUDA C programming. This chapter expects the reader to have had previous experience with C programming. It first introduces CUDA C as a simple, small extension to C that supports heterogeneous CPU/GPU joint computing and the widely used single program multiple data (SPMD) parallel programming model. It then covers the thought process involved in (1) identifying the part of application programs to be parallelized, (2) isolating the data to be used by the parallelized code, using an API function to allocate memory on the parallel computing device, (3) using an API function to transfer data to the parallel computing device, (4) developing a kernel function that will be executed by threads in the parallelized part, (5) launching a kernel function for execution by parallel threads, and (6) eventually transferring the data back to the host processor with an API function call.

While the objective of Chapter 2, Data parallel computing, is to teach enough concepts of the CUDA C programming model so that the students can write a simple parallel CUDA C program, it actually covers several basic skills needed to develop a parallel application based on any parallel programming model. We use a running example of vector addition to illustrate these concepts. In the later part of the book, we also compare CUDA with other parallel programming models including OpenMP, OpenACC, and OpenCL.

Chapter 3, Scalable parallel execution, presents more details of the parallel execution model of CUDA. It gives enough insight into the creation, organization, resource binding, data binding, and scheduling of threads to enable the reader to implement sophisticated computation using CUDA C and reason about the performance behavior of their CUDA code.

Chapter 4, Memory and data locality, is dedicated to the special memories that can be used to hold CUDA variables for managing data delivery and improving program execution speed. We introduce the CUDA language features that allocate and use these memories. Appropriate use of these memories can drastically improve the data access throughput and help to alleviate the traffic congestion in the memory system.

Chapter 5, Performance considerations, presents several important performance considerations in current CUDA hardware. In particular, it gives more details in desirable patterns of thread execution, memory data accesses, and resource allocation. These details form the conceptual basis for programmers to reason about the consequence of their decisions on organizing their computation and data.

Chapter 6, Numerical considerations, introduces the concepts of IEEE-754 floating-point number format, precision, and accuracy. It shows why different parallel execution arrangements can result in different output values. It also teaches the concept of numerical stability and practical techniques for maintaining numerical stability in parallel algorithms.

Chapters 7, Parallel patterns: convolution, Chapter 8, Parallel patterns: prefix sum, Chapter 9, Parallel patterns—parallel histogram computation, Chapter 10, Parallel patterns: sparse matrix computation, Chapter 11, Parallel patterns: merge sort, Chapter 12, Parallel patterns: graph search, present six important parallel computation patterns that give the readers more insight into parallel programming techniques and parallel execution mechanisms. Chapter 7, Parallel patterns: convolution, presents convolution and stencil, frequently used parallel computing patterns that require careful management of data access locality. We also use this pattern to introduce constant memory and caching in modern GPUs. Chapter 8, Parallel patterns: prefix sum, presents reduction tree and prefix sum, or scan, an important parallel computing pattern that converts sequential computation into parallel computation. We also use this pattern to introduce the concept of work-efficiency in parallel algorithms. Chapter 9, Parallel patterns—parallel histogram computation, covers histogram, a pattern widely used in pattern recognition in large data sets. We also cover merge operation, a widely used pattern in divide-and-concur work partitioning strategies. Chapter 10, Parallel patterns: sparse matrix computation, presents sparse matrix computation, a pattern used for processing very large data sets. This chapter introduces the reader to the concepts of rearranging data for more efficient parallel access: data compression, padding, sorting, transposition, and regularization. Chapter 11, Parallel patterns: merge sort, introduces merge sort, and dynamic input data identification and organization. Chapter 12, Parallel patterns: graph search, introduces graph algorithms and how graph search can be efficiently implemented in GPU programming.

While these chapters are based on CUDA, they help the readers build-up the foundation for parallel programming in general. We believe that humans understand best when they learn from concrete examples. That is, we must first learn the concepts in the context of a particular programming model, which provides us with solid footing to allow applying 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.

Chapter 13, CUDA dynamic parallelism, covers dynamic parallelism. This is the ability of the GPU to dynamically create work for itself based on the data or program structure, rather than waiting for the CPU to launch kernels exclusively.

Chapters 14, Application case study—non-Cartesian MRI, Chapter 15, Application case study—molecular visualization and analysis, Chapter 16, Application case study—machine learning, are case studies of three real applications, which take the readers through the thought process of parallelizing and optimizing their applications for significant speedups. For each application, we start by identifying alternative ways of formulating the basic structure of the parallel execution and follow up with reasoning about the advantages and disadvantages of each alternative. We then go through the steps of code transformation needed to achieve high-performance. These three chapters help the readers put all the materials from the previous chapters together and prepare for their own application development projects. Chapter 14, Application case study—non-Cartesian MRI, covers non-Cartesian MRI reconstruction, and how the irregular data affects the program. Chapter 15, Application case study—molecular visualization and analysis, covers molecular visualization and analysis. Chapter 16, Application case study—machine learning, covers Deep Learning, which is becoming an extremely important area for GPU computing. We provide an introduction, and leave more in-depth discussion to other sources.

Chapter 17, Parallel programming and computational thinking, introduces computational thinking. It does so by covering the concept of organizing the computation tasks of a program so that they can be done in parallel. We start by discussing the translational process of organizing abstract scientific concepts into computational tasks, which is an important first step in producing quality application software, serial or parallel. It then discusses parallel algorithm structures and their effects on application performance, which is grounded in the performance tuning experience with CUDA. 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 they gain in this book. We also present a high level case study to show the opportunities that can be seen through creative computational thinking.

Chapter 18, Programming a heterogeneous computing cluster, covers CUDA programming on heterogeneous clusters where each compute node consists of both CPU and GPU. We discuss the use of MPI alongside CUDA to integrate both inter-node computing and intra-node computing, and the resulting communication issues and practices.

Chapter 19, Parallel programming with OpenACC, covers Parallel Programming with OpenACC. OpenACC is a directive-based high level programming approach

which allows the programmer to identify and specify areas of code that can be subsequently parallelized by the compiler and/or other tools. OpenACC is an easy way for a parallel programmer to get started.

Chapter 20, More on CUDA and GPU computing and Chapter 21, Conclusion and outlook, offer concluding remarks and an outlook for the future of massively parallel programming. We first revisit our goals and summarize how the chapters fit together to help achieve the goals. We then present a brief survey of the major trends in the architecture of massively parallel processors and how these trends will likely impact parallel programming in the future. We conclude with a prediction that these fast advances in massively parallel computing will make it one of the most exciting areas in the coming decade.

REFERENCES

Gelado, I., Cabezas, J., Navarro, N., Stone, J.E., Patel, S.J., Hwu, W.W. (2010). An asynchronous distributed shared memory model for heterogeneous parallel systems. *International conference on architectural support for programming languages and operating systems*.

Hwu, W. W., Keutzer, K., & Mattson, T. (2008). The concurrency challenge. *IEEE Design and Test of Computers*, *25*, 312–320.

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns of parallel programming*. Boston, MA: Addison-Wesley Professional.

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

NVIDIA Corporation. CUDA Programming Guide. February 2007.

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

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

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

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

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

CHAPTER

Data parallel computing 2

David Luebke

CHAPTER OUTLINE

Many code examples will be used to illustrate the key concepts in writing scalable parallel programs. For this we need a simple language that supports massive parallelism and heterogeneous computing, and we have chosen CUDA C for our code examples and exercises. CUDA C extends the popular C programming language with minimal new syntax and interfaces to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs. As the name implies, CUDA C is built on NVIDIA's CUDA platform. CUDA is currently the most mature framework for massively parallel computing. It is broadly used in the high performance computing industry, with sophisticated tools such as compilers, debuggers, and profilers available on the most common operating systems.

An important point: while our examples will mostly use CUDA C for its simplicity and ubiquity, the CUDA platform supports many languages and application programming interfaces (APIs) including C++, Python, Fortran, OpenCL, OpenACC, OpenMP, and more. CUDA is really an architecture that supports a set

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00002-9

of concepts for organizing and expressing massively parallel computation. It is those concepts that we teach. For the benefit of developers working in other languages (C++, FORTRAN, Python, OpenCL, etc.) we provide appendices that show how the concepts can be applied to these languages.

2.1 DATA PARALLELISM

When modern software applications run slowly, the problem is usually having too much data to be processed. Consumer applications manipulate images or videos, with millions to trillions of pixels. Scientific applications model fluid dynamics using billions of grid cells. Molecular dynamics applications must simulate interactions between thousands to millions of atoms. Airline scheduling deals with thousands of flights, crews, and airport gates. Importantly, most of these pixels, particles, cells, interactions, flights, and so on can be dealt with largely independently. Converting a color pixel to a greyscale requires only the data of that pixel. Blurring an image averages each pixel's color with the colors of nearby pixels, requiring only the data of that small neighborhood of pixels. Even a seemingly global operation, such as finding the average brightness of all pixels in an image, can be broken down into many smaller computations that can be executed independently. Such independent evaluation is the basis of *data parallelism*: (re)organize the computation around the data, such that we can execute the resulting independent computations in parallel to complete the overall job faster, often much faster.

TASK PARALLELISM VS. DATA PARALLELISM

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

In large applications, there are usually a larger number of independent tasks and therefore larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks include vibrational forces, rotational forces, neighbor identification for nonbonding forces, nonbonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce streams.

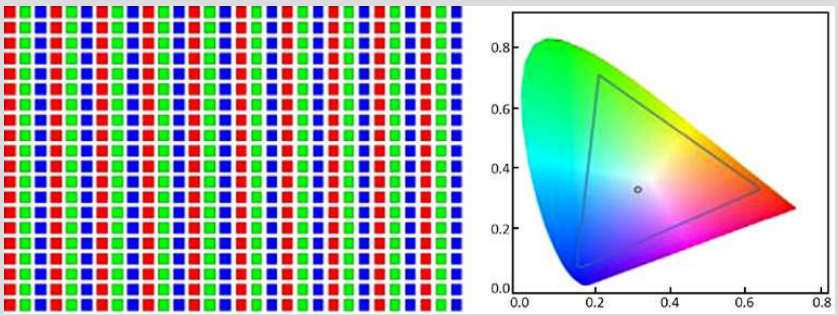
FIGURE 2.1

Conversion of a color image to a greyscale image.

We will use image processing as a source of running examples in the next chapters. Let us illustrate the concept of data parallelism with the color-to-greyscale conversion example mentioned above. Fig. 2.1 shows a color image (left side) consisting of many pixels, each containing a red, green, and blue fractional value (*r*, *g*, *b*) varying from 0 (black) to 1 (full intensity).

RGB COLOR IMAGE REPRESENTATION

In an RGB representation, each pixel in an image is stored as a tuple of (*r*, *g*, *b*) values. The format of an image's row is (*r g b*) (*r g b*) … (*r g b*), as illustrated in the following conceptual picture. Each tuple specifies a mixture of red (*R*), green (*G*) and blue (*B*). That is, for each pixel, the *r*, *g*, and *b* values represent the intensity (0 being dark and 1 being full intensity) of the red, green, and blue light sources when the pixel is rendered.



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

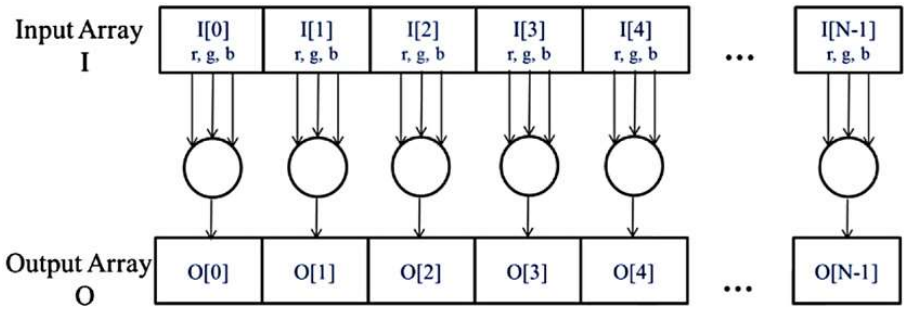


FIGURE 2.2

The pixels can be calculated independently of each other during color to greyscale conversion.

To convert the color image (left side of Fig. 2.1) to greyscale (right side) we compute the luminance value L for each pixel by applying the following weighted sum formula:

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

If we consider the input to be an image organized as an array I of RGB values and the output to be a corresponding array O of luminance values, we get the simple computation structure shown in Fig. 2.2. For example, $O[0]$ is generated by calculating the weighted sum of the RGB values in $I[0]$ according to the formula above; $O[1]$ by calculating the weighted sum of the RGB values in $I[1]$, $O[2]$ by calculating the weighted sum of the RGB values in $I[2]$, and so on. None of these per-pixel computations depends on each other; all of them can be performed independently. Clearly the color-to-greyscale conversion exhibits a rich amount of data parallelism. Of course, data parallelism in complete applications can be more complex and much of this book is devoted to teaching the "parallel thinking" necessary to find and exploit data parallelism.

2.2 CUDA C PROGRAM STRUCTURE

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

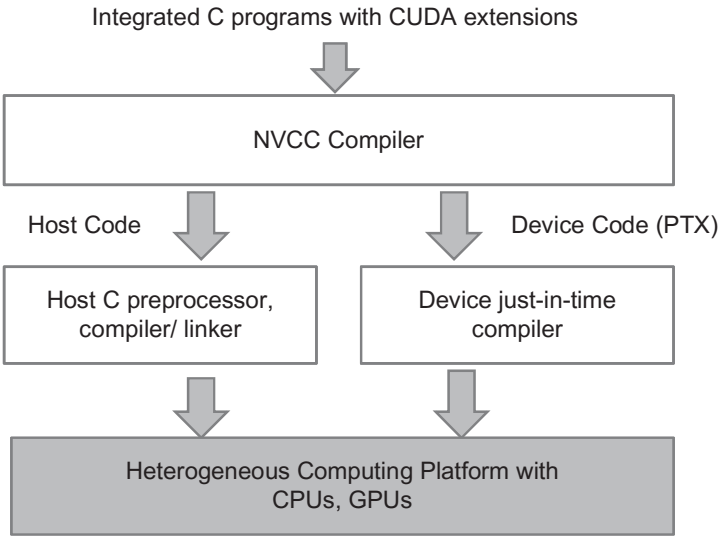


FIGURE 2.3

Overview of the compilation process of a CUDA C Program.

Once device functions and data declarations are added to a source file, it is no longer acceptable to a traditional C compiler. The code needs to be compiled by a compiler that recognizes and understands these additional declarations. We will be using a CUDA C compiler called NVCC (NVIDIA C Compiler). As shown at the top of Fig. 2.3, the NVCC compiler processes a CUDA C program, using the CUDA keywords to separate the host code and device code. The host code is straight ANSI C code, which is further compiled with the host's standard C/C++ compilers and is run as a traditional CPU process. The device code is marked with CUDA keywords for data parallel functions, called *kernels*, and their associated helper functions and data structures. The device code is further compiled by a run-time component of NVCC and executed on a GPU device. In situations where there is no hardware device available or a kernel can be appropriately executed on a CPU, one can also choose to execute the kernel on a CPU using tools like MCUDA [SSH 2008].

The execution of a CUDA program is illustrated in Fig. 2.4. The execution starts with host code (CPU serial code). When a kernel function (parallel device code) is called, or launched, it is executed by a large number of threads on a device. All the threads that are generated by a kernel launch are collectively called a *grid*. These threads are the primary vehicle of parallel execution in a CUDA platform. Fig. 2.4 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, the execution continues on the host until another kernel is launched. Note that Fig. 2.4 shows a simplified model where the CPU execution and the GPU execution do not overlap. Many heterogeneous computing applications actually manage overlapped CPU and GPU execution to take advantage of both CPUs and GPUs.

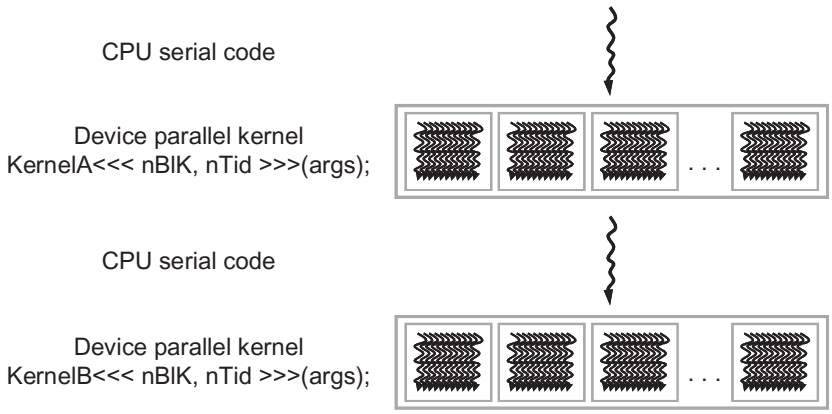


FIGURE 2.4

Execution of a CUDA program.

Launching a kernel typically generates a large number of threads to exploit data parallelism. In the color-to-greyscale conversion example, each thread could be used to compute one pixel of the output array *O*. In this case, the number of threads that will be generated by the kernel is equal to the number of pixels in the image. For large images, a large number of threads will be generated. In practice, each thread may process multiple pixels for efficiency. CUDA programmers can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support. This is in contrast with traditional CPU threads that typically take thousands of clock cycles to generate and schedule.

THREADS

A thread is a simplified view of how a processor executes a sequential program in modern computers. A thread consists of the code of the program, the particular point in the code that is being executed, and the values of its variables and data structures. The execution of a thread is sequential as far as a user is concerned. One can use a source-level debugger to monitor the progress of a thread by executing one statement at a time, looking at the statement that will be executed next and checking the values of the variables and data structures as the execution progresses.

Threads have been used in programming for many years. If a programmer wants to start parallel execution in an application, he/she creates and manages multiple threads using thread libraries or special languages. In CUDA, the execution of each thread is sequential as well. A CUDA program initiates parallel execution by launching kernel functions, which causes the underlying run-time mechanisms to create many threads that process different parts of the data in parallel.

2.3 A VECTOR ADDITION KERNEL

We now use vector addition to illustrate the CUDA C program structure. Vector addition is arguably the simplest possible data parallel computation, the parallel equivalent of "Hello World" from sequential programming. Before we show the kernel code for vector addition, it is helpful to first review how a conventional vector addition (host code) function works. Fig. 2.5 shows a simple traditional C program that consists of a main function and a vector addition function. In all our examples, whenever there is a need to distinguish between host and device data, we will prefix the names of variables that are processed by the host with "*h_*" and those of variables that are processed by a device "*d_*" to remind ourselves the intended usage of these variables. Since we only have host code in Fig. 2.5, we see only "*h_*" variables.

Assume that the vectors to be added are stored in arrays *A* and *B* that are allocated and initialized in the main program. The output vector is in array *C*, which is also allocated in the main program. For brevity, we do not show the details of how `A`, `B`, and `C` are allocated or initialized in the main function. The pointers (see sidebar below) to these arrays are passed to the vecAdd function, along with the variable *N* that contains the length of the vectors. Note that the formal parameters of the vecAdd function are prefixed with "*h_*" to emphasize that these are processed by the host. This naming convention will be helpful when we introduce device code in the next few steps.

The vecAdd function in Fig. 2.5 uses a for-loop to iterate through the vector elements. In the *i*th iteration, output element *h_*C[*i*] receives the sum of *h_*A[*i*] and *h_*B[*i*]. The vector length parameter *n* is used to control the loop so that the number of iterations matches the length of the vectors. The formal parameters *h_*A, *h_*B and *h_*C are passed by reference so the function reads the elements of *h_*A, *h_*B and writes the elements of *h_*C through the argument pointers `A`, `B`, and `C`. When the

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

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

FIGURE 2.5

A simple traditional vector addition C code example.

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

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

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

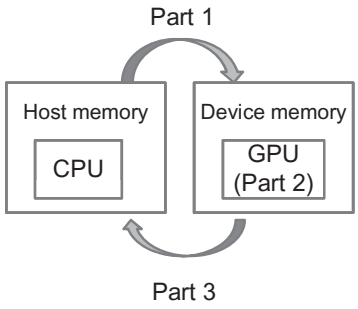


FIGURE 2.6

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

vecAdd function returns, the subsequent statements in the main function can access the new contents of `C`.

A straightforward way to execute vector addition in parallel is to modify the vecAdd function and move its calculations to a device. The structure of such a modified vecAdd function is shown in Fig. 2.6. At the beginning of the file, we need to add a `C` preprocessor directive to include the cuda.h header file. This file defines the CUDA API functions and built-in variables (see sidebar below) that we will be introducing soon. Part 1 of the function allocates space in the device (GPU) memory to hold copies of the `A`, `B`, and `C` vectors and copies the vectors from the host memory to the device memory. Part 2 launches parallel execution of the actual vector addition kernel on the device. Part 3 copies the sum vector `C` from the device memory back to the host memory and frees the vectors in device memory.

POINTERS IN THE C LANGUAGE

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

```
float V;
```

a pointer variable *P* can be declared with:

```
float *P;
```

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

An array in a C program can be accessed through a pointer that points to its 0th element. For example, the statement `P=&(A[0])` makes `P` point to the 0th element of array `A`. `P[i]` becomes a synonym for `A[i]`. In fact, the array name `A` is in itself a pointer to its 0th element.

In Fig. 2.5, passing an array name `A` as the first argument to function call to `vecAdd` makes the function's first parameter `h_A` point to the 0th element of `A`. We say that `A` is passed by reference to `vecAdd`. As a result, `h_A[i]` in the function body can be used to access `A[i]`.

See Patt&Patel [Patt] for an easy-to-follow explanation of the detailed usage of pointers in C.

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

2.4 DEVICE GLOBAL MEMORY AND DATA TRANSFER

In current CUDA systems, devices are often hardware cards that come with their own dynamic random access memory (DRAM). For example, the NVIDIA GTX1080 comes with up to 8 GB[1] of DRAM, called global memory. We will use the terms global memory and device memory interchangeably. In order to execute a kernel on a device, the programmer needs to allocate global memory on the device and transfer pertinent data from the host memory to the allocated device memory. This corresponds to Part 1 of Fig. 2.6. 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 Fig. 2.6. The CUDA run-time system provides API functions to perform these activities on behalf of the programmer. From this point on, we will simply say that

[1] There is a trend to integrate the address space of CPUs and GPUs into a unified memory space (Chapter 20). There are new programming frameworks such as GMAC that take advantage of the unified memory space and eliminate data copying cost.

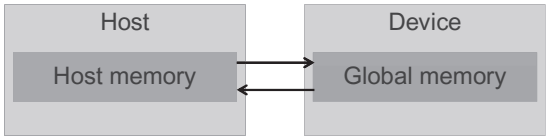


FIGURE 2.7

Host memory and device global memory.

a piece of data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device memory. The same holds for the opposite direction.

Fig. 2.7 shows a high level picture of the CUDA host memory and device memory model for programmers to reason about the allocation of device memory and movement of data between host and device. The device global memory can be accessed by the host to transfer data to and from the device, as illustrated by the bi-directional arrows between these memories and the host in Fig. 2.7. There are more device memory types than shown in Fig. 2.7. Constant memory can be accessed in a read-only manner by device functions, which will be described in Chapter 7, Parallel patterns: convolution. We will also discuss the use of registers and shared memory in Chapter 4, Memory and data locality. Interested readers can also see the CUDA programming guide for the functionality of texture memory. For now, we will focus on the use of global memory.

BUILT-IN VARIABLES

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

In Fig. 2.6, Part 1 and Part 3 of the vecAdd function need to use the CUDA API functions to allocate device memory for A, B, and C, transfer A and B from host memory to device memory, transfer C from device memory to host memory at the end of the vector addition, and free the device memory for A, B, and C. We will explain the memory allocation and free functions first.

Fig. 2.8 shows two API functions for allocating and freeing device global memory. The cudaMalloc function can be called from the host code to allocate a piece of device global memory for an object. The reader should notice the striking similarity between cudaMalloc and the standard C run-time library malloc function. This is intentional; CUDA is C with minimal extensions. CUDA uses the standard C

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

cudaFree()
- Frees object from device global memory
  - **Pointer** to freed object

FIGURE 2.8

CUDA API functions for managing device global memory.

run-time library malloc function to manage the host memory and adds cudaMalloc as an extension to the C run-time library. By keeping the interface as close to the original C run-time libraries as possible, CUDA minimizes the time that a C programmer spends to relearn the use of these extensions.

The first parameter to the cudaMalloc function is the **address** of a pointer variable that will be set to point to the allocated object. The address of the pointer variable should be cast to (void **) because the function expects a generic pointer; the memory allocation function is a generic function that is not restricted to any particular type of objects.[2] This parameter allows the cudaMalloc function to write the address of the allocated memory into the pointer variable.[3] The host code to launch kernels passes this pointer value to the kernels that need to access the allocated memory object. The second parameter to the cudaMalloc function gives the size of the data to be allocated, in number of bytes. The usage of this second parameter is consistent with the size parameter to the C malloc function.

We now use a simple code example to illustrate the use of cudaMalloc. This is a continuation of the example in Fig. 2.6. For clarity, we will start a pointer variable with letter "*d*_" to indicate that it points to an object in the device memory. The program passes the *address* of pointer *d*_A (i.e., &*d*_A) as the first parameter after casting it to a void pointer. That is, *d*_A will point to the device memory region allocated for the A vector. The size of the allocated region will be n times the size of a single-precision floating number, which is 4 bytes in most computers today. After the computation, cudaFree is called with pointer *d*_A as input to free the storage space for the A vector from the device global memory. Note that cudaFree does not need to

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

[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.

change the content of pointer variable *d*_A; it only needs to use the value of *d*_A to enter the allocated memory back into the available pool. Thus only the value, not the address of *d*_A, is passed as the argument.

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

The addresses in *d*_A, *d*_B, and *d*_C are addresses in the device memory. These addresses should not be dereferenced in the host code for computation. They should be mostly used in calling API functions and kernel functions. Dereferencing a device memory point in host code can cause exceptions or other types of run-time errors during execution.

The reader should complete Part 1 of the vecAdd example in Fig. 2.6 with similar declarations of *d*_B and *d*_C pointer variables as well as their corresponding cudaMalloc calls. Furthermore, Part 3 in Fig. 2.6 can be completed with the cudaFree calls for *d*_B and *d*_C.

Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling one of the CUDA API functions. Fig. 2.9 shows such an API function, cudaMemcpy. The cudaMemcpy function takes four parameters. The first parameter is a pointer to the destination location for the data object to be copied. The second parameter points to the source location. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the types of memory involved in the copy: from host memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory. For example, the memory copy function can be used to copy data from one location of the device memory to another location of the device memory.[4]

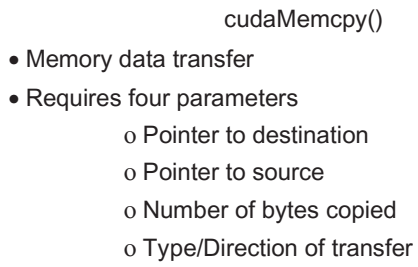


FIGURE 2.9

CUDA API function for data transfer between host and device.

[4] Please note cudaMemcpy currently cannot be used to copy between different GPU's in multi-GPU systems.

The vecAdd function calls the cudaMemcpy function to copy *h*_A and *h*_B vectors from host to device before adding them and to copy the *h*_C vector from the device to host after the addition is done. Assume that the values of *h*_A, *h*_B, *d*_A, *d*_B and size have already been set as we discussed before, the three cudaMemcpy calls are shown below. The two symbolic constants, cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost, are recognized, predefined constants of the CUDA programming environment. Note that the same function can be used to transfer data in both directions by properly ordering the source and destination pointers and using the appropriate constant for the transfer type.

```
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
```

To summarize, the main program in Fig. 2.5 calls vecAdd, which is also executed on the host. The vecAdd function, outlined in Fig. 2.6, allocates device memory, requests data transfers, and launches the kernel that performs the actual vector addition. We often refer to this type of host code as a *stub function* for launching a kernel. After the kernel finishes execution, vecAdd also copies result data from device to the host. We show a more complete version of the vecAdd function in Fig. 2.10.

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

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

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

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

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

FIGURE 2.10

A more complete version of `vecAdd()`.

ERROR CHECKING AND HANDLING IN CUDA

In general, it is very important for a program to check and handle errors. CUDA API functions return flags that indicate whether an error has occurred when they served the request. Most errors are due to inappropriate argument values used in the call.

For brevity, we will not show error checking code in our examples. For example, Fig. 2.10 shows a call to cudaMalloc:

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

In practice, we should surround the call with code that test for error condition and print out error messages so that the user can be aware of the fact that an error has occurred. A simple version of such checking code is as follows:

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

This way, if the system is out of device memory, the user will be informed about the situation. This can save many hours of debugging time.

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

Compared to Fig. 2.6, the vecAdd function in Fig. 2.10 is complete for Part 1 and Part 3. Part 1 allocates device memory for *d*_A, *d*_B, and *d*_C and transfer *h*_A to *d*_A and *h*_B to *d*_B. This is done by calling the cudaMalloc and cudaMemcpy functions. The readers are encouraged to write their own function calls with the appropriate parameter values and compare their code with that shown in Fig. 2.10. Part 2 invokes the kernel and will be described in the following subsection. Part 3 copies the sum data from device memory to host memory so that their values will be available in the main function. This is accomplished with a call to the cudaMemcpy function. It then frees the memory for *d*_A, *d*_B, and *d*_C from the device memory, which is done by calls to the cudaFree function.

2.5 KERNEL FUNCTIONS AND THREADING

We are now ready to discuss more about the CUDA kernel functions and the effect of launching these kernel functions. In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Since all these threads execute the same code, CUDA programming is an instance of the well-known Single-Program

Multiple-Data (SPMD) [Ata 1998] parallel programming style, a popular programming style for massively parallel computing systems.[5]

When a program's host code launches a kernel, the CUDA run-time system generates a grid of threads that are organized into a two-level hierarchy. Each grid is organized as an array of thread blocks, which will be referred to as blocks for brevity. All blocks of a grid are of the same size; each block can contain up to 1024 threads.[6] Fig. 2.11 shows an example where each block consists of 256 threads. Each thread is represented by a curly arrow stemming from a box that is labeled with a number. The total number of threads in each thread block is specified by the host code when a kernel is launched. The same kernel can be launched with different numbers of threads at different parts of the host code. For a given grid, the number of threads in a block is available in a built-in blockDim variable.

The blockDim variable is of struct type with three unsigned integer fields: *x*, *y*, and *z*, which help a programmer to organize the threads into a one-, two-, or three-dimensional array. For a one-dimensional organization, only the *x* field will be used. For a two-dimensional organization, *x* and *y* fields will be used. For a three-dimensional structure, all three fields will be used. The choice of dimensionality for organizing threads usually reflects the dimensionality of the data. This makes sense since the threads are created to process data in parallel. It is only natural that the organization of the threads reflects the organization of the data. In Fig. 2.11, each thread block is organized as a one-dimensional array of threads because the data are one-dimensional vectors. The value of the blockDim.x variable specifies the total number of threads in each block, which is 256 in Fig. 2.11. In general, the number of

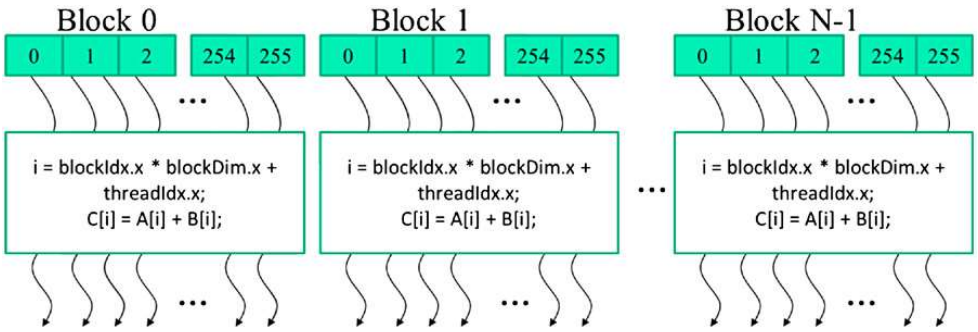


FIGURE 2.11

All threads in a grid execute the same kernel code.

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

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

threads in each dimension of thread blocks should be multiples of 32 due to hardware efficiency reasons. We will revisit this later.

CUDA kernels have access to two more built-in variables (threadIdx, blockIdx) that allow threads to distinguish among themselves and to determine the area of data each thread is to work on. Variable threadIdx gives each thread a unique coordinate within a block. For example, in Fig. 2.11, since we are using a one-dimensional thread organization, only threadIdx.x will be used. The threadIdx.x value for each thread is shown in the small shaded box of each thread in Fig. 2.11. The first thread in each block has value 0 in its threadIdx.x variable, the second thread has value 1, the third thread has value 2, etc.

The blockIdx variable gives all threads in a block a common block coordinate. In Fig. 2.11, all threads in the first block have value 0 in their blockIdx.x variables, those in the second thread block value 1, and so on. Using an analogy with the telephone system, one can think of threadIdx.x as local phone number and blockIdx.x as area code. The two together gives each telephone line a unique phone number in the whole country. Similarly, each thread can combine its threadIdx and blockIdx values to create a unique global index for itself within the entire grid.

In Fig. 2.11, a unique global index i is calculated as i = blockIdx.x*blockDim.x + threadIdx.x. Recall that blockDim is 256 in our example. The i values of threads in block 0 range from 0 to 255. The i values of threads in block 1 range from 256 to 511. The i values of threads in block 2 range from 512 to 767. That is, the i values of the threads in these three blocks form a continuous coverage of the values from 0 to 767. Since each thread uses i to access A, B, and C, these threads cover the first 768 iterations of the original loop. Note that we do not use the "*h_*" and "*d_*" convention in kernels since there is no potential confusion. We will not have any access to the host memory in our examples. By launching the kernel with a larger number of blocks, one can process larger vectors. By launching a kernel with n or more threads, one can process vectors of length n.

Fig. 2.12 shows a kernel function for vector addition. The syntax is ANSI C with some notable extensions. First, there is a CUDA C specific keyword "__global__" in front of the declaration of the vecAddKernel function. This keyword indicates that the function is a kernel and that it can be called from a host function to generate a grid of threads on a device.

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

FIGURE 2.12

A vector addition kernel function.

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

FIGURE 2.13

CUDA C keywords for function declaration.

In general, CUDA C extends the C language with three qualifier keywords that can be used in function declarations. The meaning of these keywords is summarized in Fig. 2.13 The "__global__" keyword indicates that the function being declared is a CUDA C kernel function. Note that there are two underscore characters on each side of the word "global." Such kernel function is to be executed on the device and can only be called from the host code except in CUDA systems that support *dynamic parallelism*, as we will explain in Chapter 13, CUDA dynamic parallelism. The "__device__" keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can only be called from a kernel function or another device function.[7]

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

Note that one can use both "__host__" and "__device__" in a function declaration. This combination tells the compilation system to generate two versions of object files for the same function. One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function. This supports a common use case when the same function source code can be recompiled to generate a device version. Many user library functions will likely fall into this category.

The second notable extension to ANSI C, in Fig. 2.12, are the built-in variables "threadIdx.x" "blockIdx.x" and "blockDim.x". Recall that all threads execute the same kernel code. There needs to be a way for them to distinguish among themselves and direct each thread towards a particular part of the data. These built-in variables

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

are the means for threads to access hardware registers that provide the identifying coordinates to threads. Different threads will see different values in their threadIdx.x, blockIdx.x and blockDim.x variables. For simplicity, we will refer to a thread as $\text{thread}_{\text{blockIdx.x, threadIdx.x}}$. Note that the ".x" implies that there should be ".y" and ".z". We will come back to this point soon.

There is an automatic (local) variable *i* in Fig. 2.12. In a CUDA kernel function, automatic variables are private to each thread. That is, a version of i will be generated for every thread. If the kernel is launched with 10,000 threads, there will be 10,000 versions of *i*, one for each thread. The value assigned by a thread to its *i* variable is not visible to other threads. We will discuss these automatic variables in more details in Chapter 4, Memory and data locality.

A quick comparison between Figs. 2.5 and 2.12 reveals an important insight for CUDA kernels and CUDA kernel launch. The kernel function in Fig. 2.12 does not have a loop that corresponds to the one in Fig. 2.5. The readers should ask where the loop went. The answer is that the loop is now replaced with the grid of threads. The entire grid forms the equivalent of the loop. Each thread in the grid corresponds to one iteration of the original loop. This type of data parallelism is sometimes also referred to as *loop parallelism*, where iterations of the original sequential code are executed by threads in parallel.

Note that there is an if ($i < n$) statement in addVecKernel in Fig. 2.12. This is because not all vector lengths can be expressed as multiples of the block size. For example, let's assume that the vector length is 100. The smallest efficient thread block dimension is 32. Assume that we picked 32 as block size. One would need to launch four thread blocks to process all the 100 vector elements. However, the four thread blocks would have 128 threads. We need to disable the last 28 threads in thread block 3 from doing work not expected by the original program. Since all threads are to execute the same code, all will test their *i* values against *n*, which is 100. With the if ($i < n$) statement, the first 100 threads will perform the addition whereas the last 28 will not. This allows the kernel to process vectors of arbitrary lengths.

When the host code launches a kernel, it sets the grid and thread block dimensions via *execution configuration parameters*. This is illustrated in Fig. 2.14. The configuration parameters are given between the " <<<" and ">>>" before the traditional C function arguments. The first configuration parameter gives the number of thread blocks in the grid. The second specifies the number of threads in each thread block. In this example, there are 256 threads in each block. In order to ensure that we

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

FIGURE 2.14

A vector addition kernel launch statement.

have enough threads to cover all the vector elements, we apply the C ceiling function to n/256.0. Using floating-point value 256.0 ensures that we generate a floating value for the division so that the ceiling function can round it up correctly. For example, if we have 1000 threads, we would launch ceil(1000/256.0) = 4 thread blocks. As a result, the statement will launch 4*256 = 1024 threads. With the if ($i < n$) statement in the kernel as shown in Fig. 2.12, the first 1000 threads will perform addition on the 1000 vector elements. The remaining 24 will not.

2.6 KERNEL LAUNCH

Fig. 2.15 shows the final host code in the vecAdd function. This source code completes the skeleton in Fig. 2.6. Figs. 2.12 and 2.15 jointly illustrate a simple CUDA program that consists of both host code and a device kernel. The code is hardwired to use thread blocks of 256 threads each. The number of thread blocks used, however, depends on the length of the vectors (n). If n is 750, three thread blocks will be used. If n is 4000, 16 thread blocks will be used. If n is 2,000,000, 7813 blocks will be used. Note that all the thread blocks operate on different parts of the vectors. They can be executed in any arbitrary order. Programmers must not make any assumptions regarding execution order. A small GPU with a small amount of execution resources may execute only one or two of these thread blocks in parallel. A larger GPU may execute 64 or 128 blocks in parallel. This gives CUDA kernels scalability in execution speed with hardware, that is, same code runs at lower speed on small GPUs and

```
void vecAdd(float* A, float* B, float* C, int n)
{

  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

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

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

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

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

FIGURE 2.15

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

higher speed on larger GPUs. We will revisit this point later in Chapter 3, Scalable parallel execution.

It is important to point out again that the vector addition example is used for its simplicity. In practice, the overhead of allocating device memory, input data transfer from host to device, output data transfer from device to host, and de-allocating device memory will likely make the resulting code slower than the original sequential code in Fig. 2.5. This is because the amount of calculation done by the kernel is small relative to the amount of data processed. Only one addition is performed for two floating-point input operands and one floating-point output operand. Real applications typically have kernels where much more work is needed relative to the amount of data processed, which makes the additional overhead worthwhile. They also tend to keep the data in the device memory across multiple kernel invocations so that the overhead can be amortized. We will present several examples of such applications.

2.7 SUMMARY

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

FUNCTION DECLARATIONS

CUDA C extends the C function declaration syntax to support heterogeneous parallel computing. The extensions are summarized in Fig. 2.13. Using one of "__global__", "__device__", or "__host__", a CUDA C programmer can instruct the compiler to generate a kernel function, a device function, or a host function. All function declarations without any of these keywords default to host functions. If both "__host__" and "_device__" are used in a function declaration, the compiler generates two versions of the function, one for the device and one for the host. If a function declaration does not have any CUDA C extension keyword, the function defaults into a host function.

KERNEL LAUNCH

CUDA C 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 2016] for more details of the kernel launch extensions as well as other types of execution configuration parameters.

BUILT-IN (PREDEFINED) VARIABLES

CUDA kernels can access a set of built-in, predefined read-only variables that allow each thread to distinguish among themselves and to determine the area of data each thread is to work on. We discussed the threadIdx, blockDim, and blockIdx variables in this chapter. In Chapter 3, Scalable parallel execution, we will discuss more details of using these variables.

RUN-TIME API

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

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

2.8 EXERCISES

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

A. `i=threadIdx.x + threadIdx.y;`
B. `i=blockIdx.x + threadIdx.x;`
C. `i=blockIdx.x*blockDim.x + threadIdx.x;`
D. `i=blockIdx.x * threadIdx.x;`

2. Assume that we want to use each thread to calculate two (adjacent) elements of a vector addition. What would be the expression for mapping the thread/block indices to i, the data index of the first element to be processed by a thread?

A. `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
B. `i=blockIdx.x*threadIdx.x*2;`
C. `i=(blockIdx.x*blockDim.x + threadIdx.x)*2;`
D. `i=blockIdx.x*blockDim.x*2 + threadIdx.x;`

3. We want to use each thread to calculate two elements of a vector addition. Each thread block processes 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each processing one element. Assume that variable *i* should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

A. `i=blockIdx.x*blockDim.x + threadIdx.x +2;`
B. `i=blockIdx.x*threadIdx.x*2;`
C. `i=(blockIdx.x*blockDim.x + threadIdx.x)*2;`
D. `i=blockIdx.x*blockDim.x*2 + threadIdx.x;`

4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

A. 8000
B. 8196
C. 8192
D. 8200

5. If we want to allocate an array of *v* integer elements in CUDA device global memory, what would be an appropriate expression for the second argument of the cudaMalloc call?

A. `n`
B. `v`
C. `n * sizeof(int)`
D. `v * sizeof(int)`

6. If we want to allocate an array of *n* floating-point elements and have a floating-point pointer variable *d*_A to point to the allocated memory, what would be an appropriate expression for the first argument of the cudaMalloc() call?

A. `n`
B. `(void *) d_A`
C. `*d_A`
D. `(void **) &d_A`

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

A. `cudaMemcpy(3000, h_A, d_A, cudaMemcpyHostToDevice);`
B. `cudaMemcpy(h_A, d_A, 3000, cudaMemcpyDeviceTHost);`
C. `cudaMemcpy(d_A, h_A, 3000, cudaMemcpyHostToDevice);`
D. `cudaMemcpy(3000, d_A, h_A, cudaMemcpyHostToDevice);`

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

A. `int err;`

B. `cudaError err;`

C. `cudaError_t err;`

D. `cudaSuccess_t err;`

9. A new summer intern was frustrated with CUDA. He has been complaining that CUDA is very tedious: he had to declare many functions that he plans to execute on both the host and the device twice, once as a host function and once as a device function. What is your response?

REFERENCES

Atallah, M. J. (Ed.). (1998). *Algorithms and theory of computation handbook*. Boca Raton: CRC Press.

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, *C-21*, 948-–960.

NVIDIA Corporation. (2016). NVIDIA CUDA C Programming Guide, version 7.

Patt, Y. N., & Patel, S. J. (2003). *Introduction to computing systems: from bits and gates to C and beyond*. McGraw Hill Publisher.

Stratton, J. A., Stone, S. S., & Hwu, W. W. (2008). MCUDA: an Efficient Implementation of CUDA Kernels for Multi-Core CPUs. *The 21st international workshop on languages and compilers for parallel computing*, July 30–31, Canada, 2008. Also available as Lecture Notes in Computer Science.

This page intentionally left blank

CHAPTER

Scalable parallel execution 3

Mark Ebersole

CHAPTER OUTLINE

In Chapter 2, Data parallel computing, we learned to write a simple CUDA C program that launches a kernel and a grid of threads to operate on elements in one-dimensional arrays. The kernel specifies the C statements executed by each thread. As we unleash such a massive execution activity, we need to control these activities to achieve desired results, efficiency, and speed. In this chapter, we will study important concepts involved in the control of parallel execution. We will start by learning how thread index and block index can facilitate processing multidimensional arrays. Subsequently, we will explore the concept of flexible resource assignment and the concept of occupancy. We will then advance into thread scheduling, latency tolerance, and synchronization. A CUDA programmer who masters these concepts is well-equipped to write and understand high-performance parallel applications.

3.1 CUDA THREAD ORGANIZATION

All CUDA threads in a grid execute the same kernel function; they rely on coordinates to distinguish themselves from one another and identify the appropriate portion of data to process. These threads are organized into a two-level hierarchy: a grid consists of one or more blocks, and each block consists of one or more threads. All

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00003-0

threads in a block share the same block index, which is the value of the `blockIdx` variable in a kernel. Each thread has a thread index, which can be accessed as the value of the `threadIdx` variable in a kernel. When a thread executes a kernel function, references to the `blockIdx` and `threadIdx` variables return the coordinates of the thread. The execution configuration parameters in a kernel launch statement specify the dimensions of the grid and the dimensions of each block. These dimensions are the values of the variables `gridDim` and `blockDim` in kernel functions.

HIERARCHICAL ORGANIZATIONS

Similar to CUDA threads, many real-world systems are organized hierarchically. The United States telephone system is a good example. At the top level, the telephone system consists of "areas," each of which corresponds to a geographical area. All telephone lines within the same area have the same 3-digit "area code". A telephone area can be larger than a city; e.g., many counties and cities in Central Illinois are within the same telephone area and share the same area code 217. Within an area, each phone line has a seven-digit local phone number, which allows each area to have a maximum of about ten million numbers.

Each phone line can be considered as a CUDA thread, the area code as the value of `blockIdx`, and the seven-digit local number as the value of `threadIdx`. This hierarchical organization allows the system to accommodate a considerably large number of phone lines while preserving "locality" for calling the same area. When dialing a phone line in the same area, a caller only needs to dial the local number. As long as we make most of our calls within the local area, we seldom need to dial the area code. If we occasionally need to call a phone line in another area, we dial "1" and the area code, followed by the local number. (This is the reason why no local number in any area should start with "1.") The hierarchical organization of CUDA threads also offers a form of locality, which will be examined here.

In general, a grid is a three-dimensional array of blocks[1], and each block is a three-dimensional array of threads. When launching a kernel, the program needs to specify the size of the grid and blocks in each dimension. The programmer can use fewer than three dimensions *by setting the size of the unused dimensions to 1*. The exact organization of a grid is determined by the execution configuration parameters (within <<< >>>) of the kernel launch statement. The first execution configuration parameter specifies the dimensions of the grid in the number of blocks. The second specifies the dimensions of each block in the number of threads. Each such parameter is of the `dim3` type, which is a C `struct` with three unsigned integer fields: *x*, *y*, and *z*. These three fields specify the sizes of the three dimensions.

[1] Devices with compute capability less than 2.0 support grids with up to two-dimensional arrays of blocks.

To illustrate, the following host code can be used to launch the `vecAddkernel()` kernel function and generate a 1D grid that consists of 32 blocks, each of which consists of 128 threads. The total number of threads in the grid is 128*32 = 4096.

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

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

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

The grid and block dimensions can also be calculated from other variables. The kernel launch in Fig. 2.15 can be written as follows:

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

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

For convenience, CUDA C provides a special shortcut for launching a kernel with one-dimensional grids and blocks. Instead of `dim3` variables, arithmetic expressions can be used to specify the configuration of 1D grids and blocks. In this case, the CUDA C compiler simply takes the arithmetic expression as the *x* dimensions and assumes that the *y* and *z* dimensions are 1. Thus, the kernel launch statement is as shown in Fig. 2.15:

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

Readers familiar with the use of structures in C would realize that this "shorthand" convention for 1D configurations takes advantage of the fact that the *x* field is the first field of the `dim3` structures `gridDim(x, y, z)` and `blockDim(x, y, z)`. This shortcut allows the compiler to conveniently initialize the *x* fields of `gridDim` and `blockDim` with the values provided in the execution configuration parameters.

Within the kernel function, the *x* field of the variables `gridDim` and `blockDim` are pre-initialized according to the values of the execution configuration parameters.

If n is equal to 4000, references to gridDim.x and blockDim.x in the vectAddkernel kernel will obtain 16 and 256, respectively. Unlike the dim3 variables in the host code, the names of these variables within the kernel functions are part of the CUDA C specification and cannot be changed—i.e., gridDim and blockDim in a kernel always reflect the dimensions of the grid and the blocks.

In CUDA C, the allowed values of gridDim.x, gridDim.y and gridDim.z range from 1 to 65,536. All threads in a block share the same blockIdx.x, blockIdx.y, and blockIdx.z values. Among blocks, the blockIdx.x value ranges from 0 to gridDim.x-1, the blockIdx.y value from 0 to gridDim.y-1, and the blockIdx.z value from 0 to gridDim.z-1.

Regarding the configuration of blocks, each block is organized into a three-dimensional array of threads. Two-dimensional blocks can be created by setting blockDim.z to 1. One-dimensional blocks can be created by setting both blockDim.y and blockDim.z to 1, as was the case in the vectorAddkernel example. As previously mentioned, all blocks in a grid have the same dimensions and sizes. The number of threads in each dimension of a block is specified by the second execution configuration parameter at the kernel launch. Within the kernel, this configuration parameter can be accessed as the *x*, *y*, and *z* fields of blockDim.

The total size of a block is limited to 1024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1024. For instance, blockDim(512, 1, 1), blockDim(8, 16, 4), and blockDim(32, 16, 2) are allowable blockDim values, but blockDim(32, 32, 2) is not allowable because the total number of threads would exceed 1024.[2]

The grid can have higher dimensionality than its blocks and vice versa. For instance, Fig. 3.1 shows a small toy grid example of gridDim(2, 2, 1) with blockDim(4, 2, 2). The grid can be generated with the following host code:

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

The grid consists of four blocks organized into a 2 × 2 array. Each block in Fig. 3.1 is labeled with (blockIdx.y, blockIdx.x), e.g., Block(1,0) has blockIdx.y=1 and blockIdx.x=0. The labels are ordered such that the highest dimension comes first. *Note that this block labeling notation is the reversed ordering of that used in the C statements for setting configuration parameters where the lowest dimension comes first*. This reversed ordering for labeling blocks works more effectively when we illustrate the mapping of thread coordinates into data indexes in accessing multidimensional data.

Each threadIdx also consists of three fields: the *x* coordinate threadId.x, the *y* coordinate threadIdx.y, and the *z* coordinate threadIdx.z. Fig. 3.1 illustrates the organization of threads within a block. In this example, each block is organized into 4 × 2 × 2 arrays of threads. All blocks within a grid have the same dimensions; thus, we

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

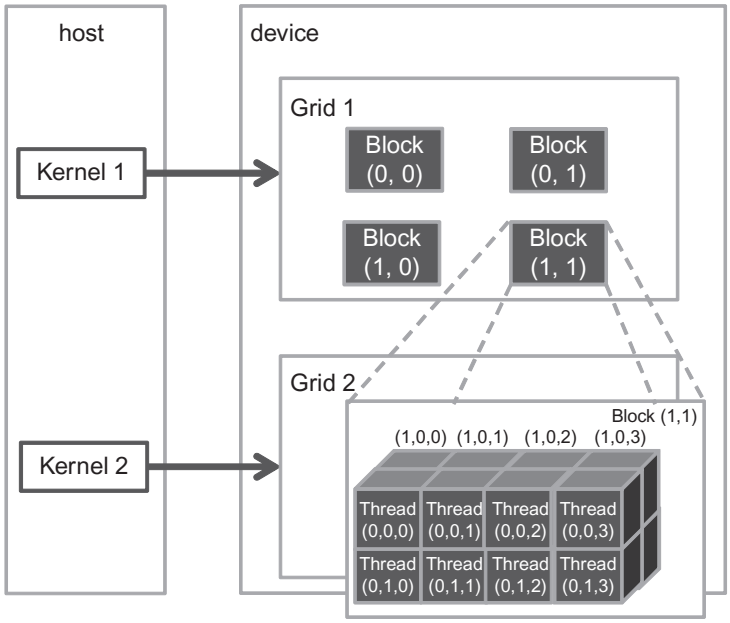


FIGURE 3.1

A multidimensional example of CUDA grid organization.

only need to show one of them. Fig. 3.1 expands Block(1,1) to show its 16 threads. For instance, Thread(1,0,2) has `threadIdx.z=1`, `threadIdx.y=0`, and `threadIdx.x=2`. This example shows 4 blocks of 16 threads each, with a total of 64 threads in the grid. We use these small numbers to keep the illustration simple. Typical CUDA grids contain thousands to millions of threads.

3.2 MAPPING THREADS TO MULTIDIMENSIONAL DATA

The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data. Pictures are 2D array of pixels. Using a 2D grid that consists of 2D blocks is often convenient for processing the pixels in a picture. Fig. 3.2 shows such an arrangement for processing a 76 × 62 picture `P` (76 pixels in the horizontal or x direction and 62 pixels in the vertical or y direction). Assume that we decided to use a 16 × 16 block, with 16 threads in the x direction and 16 threads in the y direction. We will need 5 blocks in the x direction and 4 blocks in the y direction, resulting in $5 \times 4 = 20$ blocks, as shown in Fig. 3.2. The heavy lines mark the block boundaries. The shaded area depicts the threads that cover pixels. It is easy to verify that one can identify the `Pin` element processed by thread(0,0) of block(1,0) with the formula:

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

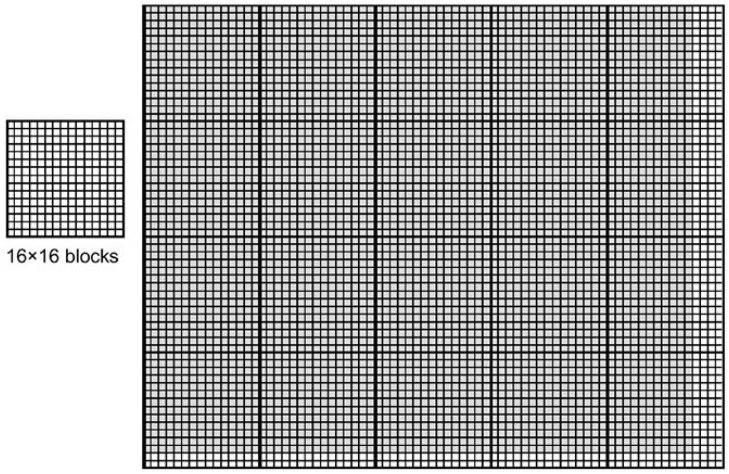


FIGURE 3.2

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

Note that we have 4 extra threads in the *x* direction and 2 extra threads in the *y* direction—i.e., we will generate 80 × 64 threads to process 76 × 62 pixels. This case is similar to the situation in which a 1000-element vector is processed by the 1D kernel `vecAddKernel` in Fig. 2.11 by using four 256-thread blocks. Recall that an if statement is needed to prevent the extra 24 threads from taking effect. Analogously, we should expect that the picture processing kernel function will have if statements to test whether the thread indexes `threadIdx.x` and `threadIdx.y` fall within the valid range of pixels.

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

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

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

MEMORY SPACE

Memory space is a simplified view of how a processor accesses its memory in modern computers. It is usually associated with each running application. The data to be processed by an application and instructions executed for the application are stored in locations in its memory space. Typically, each location can accommodate a byte and has an address. Variables that require multiple bytes—4 bytes for float and 8 bytes for double—are stored in consecutive byte locations. The processor generates the starting address (address of the starting byte location) and the number of bytes needed when accessing a data value from the memory space.

The locations in a memory space are similar to phones in a telephone system where everyone has a unique phone number. Most modern computers have at least 4G byte-sized locations, where each G is 1,073,741,824 (2^{30}). All locations are labeled with an address ranging from 0 to the largest number. Every location has only one address; thus, we say that the memory space has a "flat" organization. As a result, all multidimensional arrays are ultimately "flattened" into equivalent one-dimensional arrays. Whereas a C programmer can use a multidimensional syntax to access an element of a multidimensional array, the compiler translates these accesses into a base pointer that points to the initial element of the array, along with an offset calculated from these multidimensional indexes.

Before we show the kernel code, we need to first understand how C statements access elements of dynamically allocated multidimensional arrays. Ideally, we would like to access `d_Pin` as a two-dimensional array where an element at row *j* and column *i* can be accessed as `d_Pin[j][i]`. However, the ANSI C standard on which the development of CUDA C was based requires that the number of columns in `d_Pin` be known at compile time for `d_Pin` to be accessed as a 2D array. Unfortunately, this information is not known at compiler time for dynamically allocated arrays. In fact, part of the reason dynamically allocated arrays are used is to allow the sizes and dimensions of these arrays to vary according to data size at run time. Thus, the information on the number of columns in a dynamically allocated two-dimensional array is unknown at compile time by design. Consequently, programmers need to explicitly linearize or "flatten" a dynamically allocated two-dimensional array into an equivalent one-dimensional array in the current CUDA C. The newer C99 standard allows multidimensional syntax for dynamically allocated arrays. Future CUDA C versions may support multidimensional syntax for dynamically allocated arrays.

In reality, all multidimensional arrays in C are linearized because of the use of a "flat" memory space in modern computers (see "Memory Space" sidebar). In statically allocated arrays, the compilers allow the programmers to use higher-dimensional indexing syntax such as `d_Pin[j][i]` to access their elements. Under the hood, the

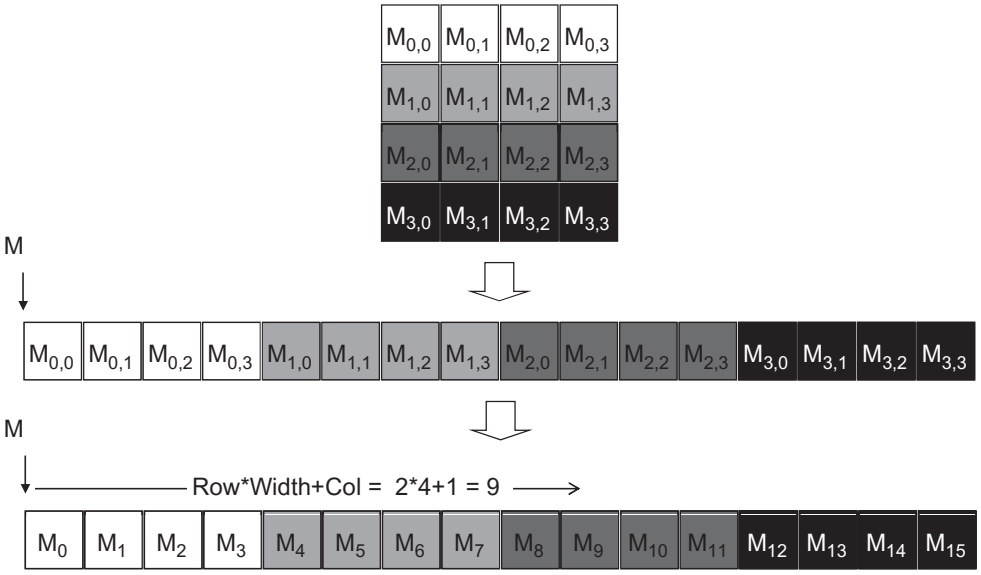


FIGURE 3.3

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

compiler linearizes them into an equivalent one-dimensional array and translates the multidimensional indexing syntax into a one-dimensional offset. In dynamically allocated arrays, the current CUDA C compiler leaves the work of such translation to the programmers because of the lack of dimensional information at compile time.

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

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

array is equivalent to the row-major layout of its transposed form. Readers whose primary previous programming experience were with FORTRAN should be aware that CUDA C uses the row-major layout rather than the column-major layout. In addition, numerous C libraries that are designed for FORTRAN programs use the column-major layout to match the FORTRAN compiler layout. Consequently, the manual pages for these libraries, such as Basic Linear Algebra Subprograms (BLAS) (see "Linear Algebra Functions" sidebar), usually instruct the users to transpose the input arrays if they call these libraries from C programs.

LINEAR ALGEBRA FUNCTIONS

Linear algebra operations are widely used in science and engineering applications. BLAS, a de facto standard for publishing libraries that perform basic algebraic operations, includes three levels of linear algebra functions. As the level increases, the number of operations performed by the function increases as well. Level-1 functions perform vector operations of the form $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. Our vector addition example is a special case of a level-1 function with α=1. Level-2 functions perform matrix–vector operations of the form $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$, where $\mathbf{A}$ is a matrix, $\mathbf{x}$ and $\mathbf{y}$ are vectors, and α, β are scalars. We will be examining a form of level-2 function in sparse linear algebra. Level-3 functions perform matrix–matrix operations of the form $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are matrices and α, β are scalars. Our matrix–matrix multiplication example is a special case of a level-3 function, where α=1 and β=0. These BLAS functions are used as basic building blocks of higher-level algebraic functions such as linear system solvers and eigenvalue analysis. As we will discuss later, the performance of different implementations of BLAS functions can vary by orders of magnitude in both sequential and parallel computers.

We are now ready to study the source code of `colorToGreyscaleConversion` shown in Fig. 3.4. The kernel code uses the formula

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

to convert each color pixel to its greyscale counterpart.

A total of `blockDim.x*gridDim.x` threads can be found in the horizontal direction. As in the `vecAddKernel` example, the expression

`Col=blockIdx.x*blockDim.x+threadIdx.x` generates every integer value from 0 to `blockDim.x*gridDim.x-1`. We know that `gridDim.x*blockDim.x` is greater than or equal to `width` (`m` value passed in from the host code). We have at least as many threads as the number of pixels in the horizontal direction. Similarly, we know that

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * Pout,  unsigned
            char * Pin, int width, int height) {,
int Col = threadIdx.x + blockIdx.x * blockDim.x;
int Row = threadIdx.y + blockIdx.y * blockDim.y;
if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

FIGURE 3.4

Source code of colorToGreyscaleConversion showing 2D thread mapping to data.

at least as many threads as the number of pixels in the vertical direction are present. Therefore, as long as we test and make sure only the threads with both `Row` and `Col` values are within range—i.e., (`Col<width`) && (`Row<height`)—we can cover every pixel in the picture.

Given that each row has `width` pixels, we can thus generate the one-dimensional index for the pixel at row `Row` and column `Col` as `Row*width+Col`. This one-dimensional index `greyOffset` is the pixel index for `Pout` as each pixel in the output greyscale image is one byte (unsigned char). By using our 76 × 62 image example, the linearized one-dimensional index of the `Pout` pixel is calculated by thread(0,0) of block(1,0) with the formula:

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

As for `Pin`, we multiply the gray pixel index by 3 because each pixel is stored as (*r*, *g*, *b*), with each equal to one byte. The resulting `rgbOffset` gives the starting location of the color pixel in the `Pin` array. We read the *r*, *g*, and *b* values from the three consecutive byte locations of the `Pin` array, perform the calculation of the greyscale pixel value, and write that value into the `Pout` array by using `greyOffset`. With our 76 × 62 image example, the linearized one-dimensional index of the `Pin` pixel is calculated by thread(0,0) of block(1,0) with the following formula:

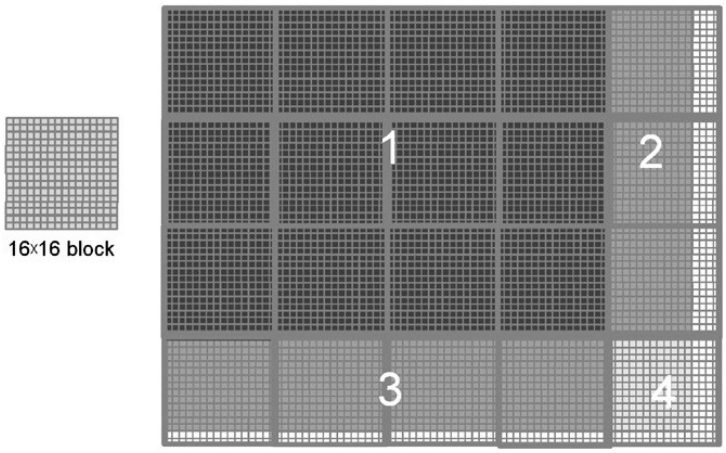


FIGURE 3.5

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

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

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

Fig. 3.5 illustrates the execution of `colorToGreyscaleConversion` when processing our 76 × 62 example. Assuming that we use 16 × 16 blocks, launching `colorToGreyscaleConvertion` generates 80 × 64 threads. The grid will have 20 blocks—5 in the horizontal direction and 4 in the vertical direction. The execution behavior of blocks will fall into one of four different cases, depicted as four shaded areas in Fig. 3.5.

The first area, marked as "1" in Fig. 3.5, consists of threads that belong to the 12 blocks covering the majority of pixels in the picture. Both the `Col` and `Row` values of these threads are within range; all these threads will pass the `if`-statement test and process pixels in the heavily shaded area of the picture—i.e., all 16 × 16 = 256 threads in each block will process pixels. The second area, marked as "2" in Fig. 3.5, contains the threads that belong to the three blocks in the medium-shaded area covering the upper right pixels of the picture. Although the `Row` values of these threads are always within range, some `Col` values exceed the `m` value (76). The reason is that the number of threads in the horizontal direction is always a multiple of the `blockDim.x` value chosen by the programmer (16 in this case). The smallest multiple of 16 needed to cover 76 pixels is 80. Thus, 12 threads in each row will have `Col` values that are within range and will process pixels. Meanwhile, 4 threads in each row will have `Col` values that are out of range and thus fail the `if`-statement condition. These threads will not process any pixels. Overall, 12 × 16 = 192 of the 16 × 16 = 256 threads in each of these blocks will process pixels.

The third area, marked "3" in Fig. 3.5, accounts for the 3 lower left blocks covering the medium-shaded area in the picture. Although the `Col` values of these threads are always within range, some `Row` values exceed the m value (62). The reason is that the number of threads in the vertical direction is always a multiple of the `blockDim.y` value chosen by the programmer (16 in this case). The smallest multiple of 16 to cover 62 is 64. Thus, 14 threads in each column will have `Row` values that are within range and will process pixels. Meanwhile, 2 threads in each column will fail the `if`-statement of area 2 and will not process any pixels. Of the 256 threads, $16 \times 14 = 224$ will process pixels. The fourth area, marked "4" in Fig. 3.5, contains threads that cover the lower right, lightly shaded area of the picture. In each of the top 14 rows, 4 threads will have `Col` values that are out of range, similar to Area 2. The entire bottom two rows of this block will have `Row` values that are out of range, similar to area "3". Thus, only $14 \times 12 = 168$ of the $16 \times 16 = 256$ threads will process pixels.

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

```
int Plane = blockIdx.z*blockDim.z + threadIdx.z
```

The linearized access to a three-dimensional array `P` will be of the form `P[Plane*m*n+Row*m+Col]`. A kernel processing the 3D `P` array needs to check whether all the three global indexes—`Plane`, `Row`, and `Col`—fall within the valid range of the array.

3.3 IMAGE BLUR: A MORE COMPLEX KERNEL

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

Image blurring smooths out the abrupt variation of pixel values while preserving the edges that are essential for recognizing the key features of the image. Fig. 3.6 illustrates the effect of image blurring. Simply stated, we make the image appear blurry. To the human eye, a blurred image tends to obscure the fine details and present

FIGURE 3.6

An original image and a blurred version.

the "big picture" impression or the major thematic objects in the picture. In computer image processing algorithms, a common use case of image blurring is to reduce the impact of noise and granular rendering effects in an image by correcting problematic pixel values with the clean surrounding pixel values. In computer vision, image blurring can be used to allow edge detection and object recognition algorithms to focus on thematic objects rather than being impeded by a massive quantity of fine-grained objects. In displays, image blurring is sometimes used to highlight a particular part of the image by blurring the rest of the image.

Mathematically, an image blurring function calculates the value of an output image pixel as a weighted sum of a patch of pixels encompassing the pixel in the input image. As we will learn in Chapter 7, Parallel pattern: convolution, the computation of such weighted sums belongs to the *convolution* pattern. We will be using a simplified approach in this chapter by taking a simple average value of the `N×N` patch of pixels surrounding, and including, our target pixel. To keep the algorithm simple, we will not place a weight on the value of any pixels based on its distance from the target pixel, which is common in a convolution blurring approach such as Gaussian blur.

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

Fig. 3.8 shows an image blur kernel. Similar to that in `colorToGreyscaleConversion`, we use each thread to calculate an output pixel. That is, the thread to output data mapping remains the same. Thus, at the beginning of the kernel, we see the familiar calculation of the `Col` and `Row` indexes. We also see the familiar `if`-statement that verifies whether both `Col` and `Row` are within the valid range according to the height and width of the image. Only the threads whose `Col` and `Row` indexes are within the value ranges will be allowed to participate in the execution.

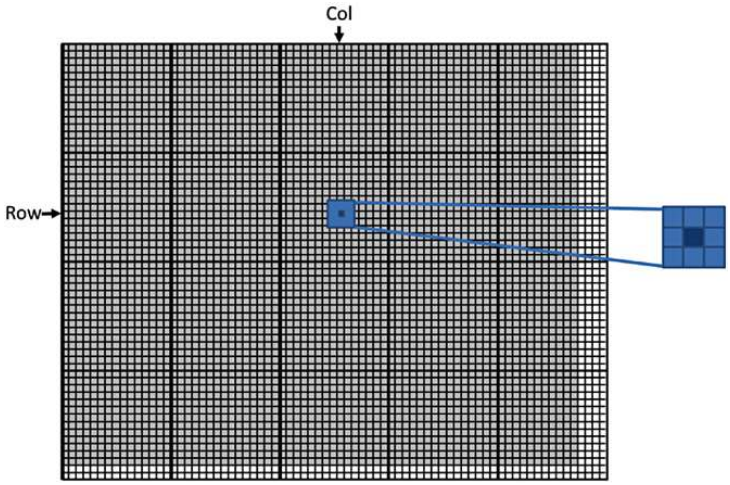


FIGURE 3.7

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

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
    {
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
1.    int pixVal = 0;
2.    int pixels = 0;

    // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.    for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol)
     {

5.        int curRow = Row + blurRow;
6.        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
7.        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {

8.          pixVal += in[curRow * w + curCol];
9.          pixels++; // Keep track of number of pixels in the avg
          }
        }
      }

    // Write our new pixel value out
10.   out[Row * w + Col] = (unsigned char)(pixVal / pixels);
  }
 }
```

FIGURE 3.8

An image blur kernel.

As shown in Fig. 3.7, the `Col` and `Row` values also generate the central pixel location of the patch used to calculate the output pixel for the thread. The nested `for`-loop Lines 3 and 4 of Fig. 3.8 iterate through all pixels in the patch. We assume that the program has a defined constant, `BLUR_SIZE`. The value of `BLUR_SIZE` is set such that 2*`BLUR_SIZE` gives the number of pixels on each side of the patch. For a 3 × 3 patch, `BLUR_SIZE` is set to 1, whereas for a 7 × 7 patch, `BLUR_SIZE` is set to 3. The outer loop iterates through the rows of the patch. For each row, the inner loop iterates through the columns of the patch.

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

Line 8 uses the linearized index of `curRow` and `curCol` to access the value of the input pixel visited in the current iteration. It accumulates the pixel value into a running sum variable `pixVal`. Line 9 records the addition of one more pixel value into the running sum by incrementing the `pixels` variable. After all pixels in the patch are processed, Line 10 calculates the average value of the pixels in the patch by dividing the `pixVal` value by the `pixels` value. It uses the linearized index of `Row` and `Col` to write the result into its output pixel.

Line 7 contains a conditional statement that guards the execution of Lines 9 and 10. For output pixels near the edge of the image, the patch may extend beyond the valid range of the picture. This is illustrated in Fig. 3.9 assuming 3 × 3 patches. In Case 1, the pixel at the upper left corner is being blurred. Five of the nine pixels in the intended patch do not exist in the input image. In this case, the `Row` and `Col` values of the output pixel are 0 and 0. During the execution of the nested loop, the `CurRow` and `CurCol` values for the nine iterations are (−1,−1), (−1,0), (−1,1), (0,−1), (0,0), (0,1), (1,−1), (1,0), and (1,1). Note that for the five pixels outside the image, at least one of the values is less than 0. The `curRow<0` and `curCol<0` conditions of the `if`-statement capture these values and skip the execution of Lines 8 and 9. As a result, only the values of the four valid pixels are accumulated into the running sum variable. The `pixels` value is also correctly incremented four times so that the average can be calculated properly at Line 10.

The readers should work through the other cases in Fig. 3.9 and analyze the execution behavior of the nested loop in the `blurKernel`. Note that most of the threads will find all pixels in their assigned 3 × 3 patch within the input image. They will accumulate all the nine pixels in the nested loop. However, for the pixels on the four corners, the responsible threads will accumulate only 4 pixels. For other pixels on the four edges, the responsible threads will accumulate 6 pixels in the nested loop.

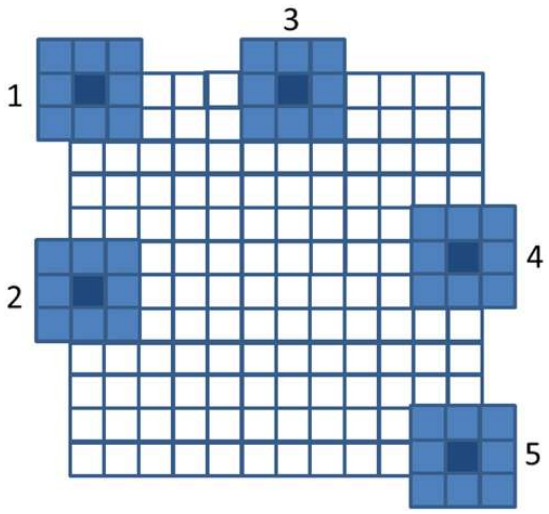


FIGURE 3.9

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

These variations necessitate keeping track of the actual number of pixels accumulated with variable `pixels`.

3.4 SYNCHRONIZATION AND TRANSPARENT SCALABILITY

We have discussed thus far how to launch a kernel for execution by a grid of threads and how to map threads to parts of the data structure. However, we have not yet presented any means to coordinate the execution of multiple threads. We will now study a basic coordination mechanism. CUDA allows threads in the same block to coordinate their activities by using a barrier synchronization function `__syncthreads()`. Note that "__" consists of two "_" characters. When a thread calls `__syncthreads()`, it will be held at the calling location until every thread in the block reaches the location. This process ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can proceed to the next phase.

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

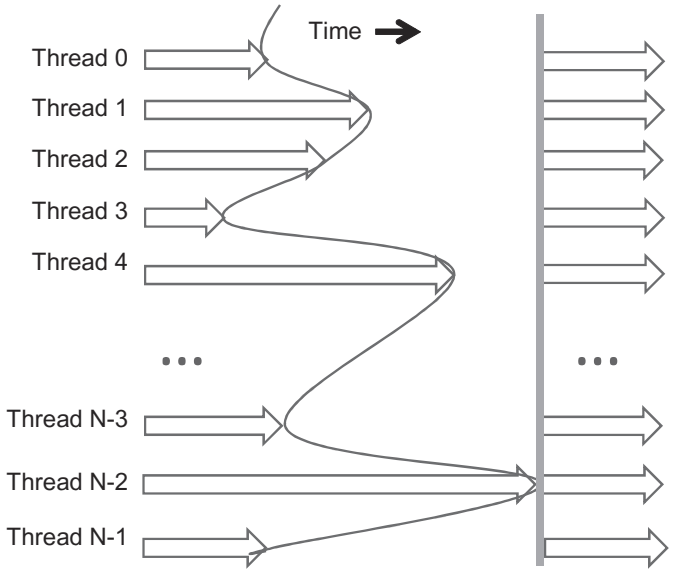


FIGURE 3.10

An example execution timing of barrier synchronization.

Fig. 3.10 illustrates the execution of barrier synchronization. There are N threads in the block. Time goes from left to right. Some of the threads reach the barrier synchronization statement early and some of them much later. The ones who reach the barrier early will wait for those who arrive late. When the latest one arrives at the barrier, everyone can continue their execution. With barrier synchronization, "No one is left behind."

In CUDA, a `__syncthreads()` statement, if present, must be executed by all threads in a block. When a `__syncthread()` statement is placed in an `if`-statement, either all or none of the threads in a block execute the path that includes the `__syncthreads()`. For an `if-then-else` statement, if each path has a `__syncthreads()` statement, either all threads in a block execute the `then`-path or all of them execute the `else`-path. The two `__syncthreads()` are different barrier synchronization points. If a thread in a block executes the `then`-path and another executes the `else`-path, they would be waiting at different barrier synchronization points. They would end up waiting for each other forever. It is the responsibility of the programmers to write their code so that these requirements are satisfied.

The ability to synchronize also imposes execution constraints on threads within a block. These threads should execute in close temporal proximity with each other to avoid excessively long waiting times. In fact, one needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier. Otherwise, a thread that never arrives at the barrier synchronization point can cause everyone else to wait forever. CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit. A block can begin execution only when the runtime system has secured all resources needed for

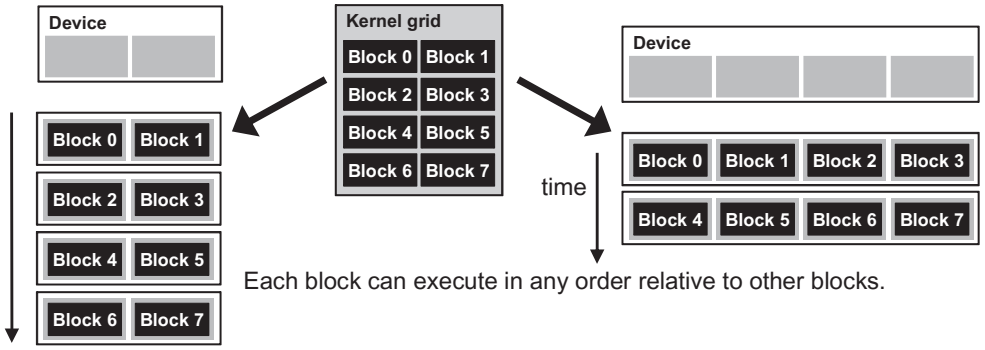


FIGURE 3.11

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

all threads in the block to complete execution. When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This condition ensures the temporal proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization.

This leads us to an important tradeoff in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other because none of them need to wait for each other. This flexibility enables scalable implementations as shown in Fig. 3.11, where time progresses from top to bottom. In a low-cost system with only a few execution resources, one can execute a small number of blocks simultaneously, portrayed as executing two blocks at a time on the left hand side of Fig. 3.11. In a high-end implementation with more execution resources, one can execute a large number of blocks simultaneously, shown as four blocks at a time on the right hand side of Fig. 3.11.

The ability to execute the same application code within a wide range of speeds allows the production of a wide range of implementations in accordance with the cost, power, and performance requirements of particular market segments. For instance, a mobile processor may execute an application slowly but at extremely low power consumption, and a desktop processor may execute the same application at a higher speed but at increased power consumption. Both execute exactly the same application program with no change to the code. The ability to execute the same application code on hardware with different numbers of execution resources is referred to as *transparent scalability*. This characteristic reduces the burden on application developers and improves the usability of applications.

3.5 RESOURCE ASSIGNMENT

Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads. As discussed in the previous section, these threads are assigned to

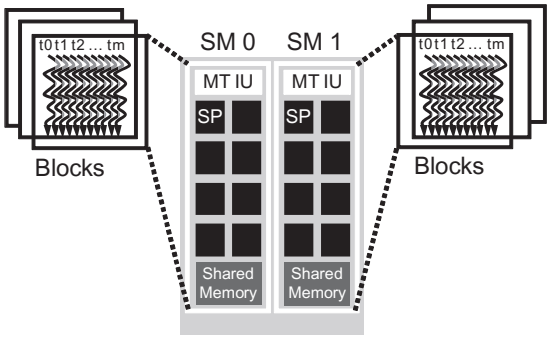


FIGURE 3.12

Thread block assignment to Streaming Multiprocessors (SMs).

execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into Streaming Multiprocessors (SMs). Fig. 3.12 illustrates that multiple thread blocks can be assigned to each SM. Each device sets a limit on the number of blocks that can be assigned to each SM. For instance, let us consider a CUDA device that may allow up to 8 blocks to be assigned to each SM. In situations where there is shortage of one or more types of resources needed for the simultaneous execution of 8 blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until their combined resource usage falls below the limit. With limited numbers of SMs and limited numbers of blocks that can be assigned to each SM, the number of blocks that can be actively executing in a CUDA device is limited as well. Most grids contain many more blocks than this number. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as previously assigned blocks complete execution.

Fig. 3.12 shows an example in which three thread blocks are assigned to each SM. One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled. It takes hardware resources (built-in registers) for SMs to maintain the thread and block indexes and track their execution status. Therefore, each generation of hardware sets a limit on the number of blocks and number of threads that can be assigned to an SM. For instance in the Fermi architecture, up to 8 blocks and 1536 threads can be assigned to each SM. This could be in the form of 6 blocks of 256 threads each, 3 blocks of 512 threads each, and so on. If the device only allows up to 8 blocks in an SM, it should be obvious that 12 blocks of 128 threads each is not a viable option. If a CUDA device has 30 SMs, and each SM can accommodate up to 1536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution.

3.6 QUERYING DEVICE PROPERTIES

Our discussions on assigning execution resources to blocks raise an important question. How do we find out the amount of resources available? When a CUDA

application executes on a system, how can it determine the number of SMs in a device and the number of blocks and threads that can be assigned to each SM? Other resources have yet to be discussed that can be relevant to the execution of a CUDA application. In general, many modern applications are designed to execute on a wide variety of hardware systems. The application often needs to *query* the available resources and capabilities of the underlying hardware in order to take advantage of the more capable systems while compensating for the less capable systems.

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

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

RESOURCE AND CAPABILITY QUERIES

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

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

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

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

While it may not be obvious, a modern PC system often has two or more CUDA devices. The reason is that many PC systems come with one or more "integrated" GPUs. These GPUs are the default graphics units and provide rudimentary capabilities and hardware resources to perform minimal graphics functionalities for

modern Windows-based user interfaces. Most CUDA applications will not perform very well on these integrated devices. This weakness would be a reason for the host code to iterate through all the available devices, query their resources and capabilities, and choose the ones with adequate resources to execute the application satisfactorily.

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

```
cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
  //decide if device has sufficient resources and capabilities
}
```

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

As the name suggests, the field `dev_prop.maxThreadsPerBlock` indicates the maximal number of threads allowed in a block in the queried device. Some devices allow up to 1024 threads in each block and other devices allow fewer. Future devices may even allow more than 1024 threads per block. Therefore, the available devices should be queried, and the ones that will allow a sufficient number of threads in each block should be determined.

The number of SMs in the device is given in `dev_prop.multiProcessorCount`. As we discussed earlier, some devices have only a small number of SMs (e.g., two) and some have a much larger number of SMs (e.g., 30). If the application requires a large number of SMs in order to achieve satisfactory performance, it should definitely check this property of the prospective device. Furthermore, the clock frequency of the device is in `dev_prop.clockRate`. The combination of the clock rate and the number of SMs provides a good indication of the hardware execution capacity of the device.

The host code can find the maximal number of threads allowed along each dimension of a block in fields `dev_prop.maxThreadsDim[0]`, `dev_prop.maxThreadsDim[1]`, and `dev_prop.maxThreadsDim[2]` (for the x, y, and z dimensions). Such information can be used for an automated tuning system to set the range of block dimensions when evaluating the best performing block dimensions for the

underlying hardware. Similarly, it can determine the maximal number of blocks allowed along each dimension of a grid in `dev_prop.maxGridSize[0]`, `dev_prop.maxGridSize[1]`, and `dev_prop.maxGridSize[2]` (for the *x*, *y*, and *z* dimensions). This information is typically used to determine whether a grid can have sufficient threads to handle the entire data set or whether some iteration is needed.

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

3.7 THREAD SCHEDULING AND LATENCY TOLERANCE

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

The warp is the unit of thread scheduling in SMs. Fig. 3.13 shows the division of blocks into warps in an implementation. Each warp consists of 32 threads of consecutive `threadIdx` values: thread 0 through 31 form the first warp, 32 through 63 the

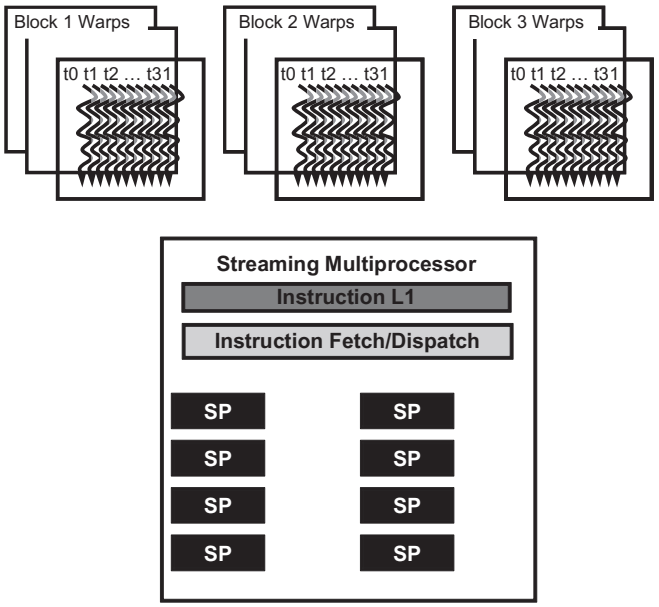


FIGURE 3.13

Blocks are partitioned into warps for thread scheduling.

second warp, and so on. In this example, three blocks—Block 1, Block 2, and Block 3—are assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.

We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM. In Fig. 3.13, if each block has 256 threads, we can determine that each block has 256/32 or 8 warps. With three blocks in each SM, we have $8 \times 3 = 24$ warps in each SM.

An SM is designed to execute all threads in a warp following the Single Instruction, Multiple Data (SIMD) model—i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. This situation is illustrated in Fig. 3.13 with a single instruction fetch/dispatch shared among execution units (SPs) in the SM. These threads will apply the same instruction to different portions of the data. Consequently, all threads in a warp will always have the same execution timing.

Fig. 3.13 also shows a number of hardware Streaming Processors (SPs) that actually execute instructions. In general, there are fewer SPs than the threads assigned to each SM; i.e., each SM has only enough hardware to execute instructions from a small subset of all threads assigned to the SM at any point in time. In early GPU designs, each SM can execute only one instruction for a single warp at any given instant. In recent designs, each SM can execute instructions for a small number of warps at any point in time. In either case, the hardware can execute instructions for a small subset of all warps in the SM. A legitimate question is why we need to have so many warps in an SM if it can only execute a small subset of them at any instant. The answer is that this is how CUDA processors efficiently execute long-latency operations, such as global memory accesses.

When an instruction to be executed by a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution. Instead, another resident warp that is no longer waiting for results will be selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution. This mechanism of filling the latency time of operations with work from other threads is often called “latency tolerance” or “latency hiding” (see “Latency Tolerance” sidebar).

Warp scheduling is also used for tolerating other types of operation latencies, such as pipelined floating-point arithmetic and branch instructions. Given a sufficient number of warps, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations. The selection of ready warps for execution avoids introducing idle or wasted time into the execution timeline, which is referred to as zero-overhead thread scheduling. With warp scheduling, the long waiting time of warp instructions is “hidden” by executing instructions from other warps. This ability to tolerate long-latency operations is the main reason GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as do CPUs. Thus, GPUs can dedicate more of its chip area to floating-point execution resources.

LATENCY TOLERANCE

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

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

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

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

We are now ready for a simple exercise.[3] Assume that a CUDA device allows up to 8 blocks and 1024 threads per SM, whichever becomes a limitation first. Furthermore, it allows up to 512 threads in each block. For image blur, should we use 8×8, 16×16, or 32×32 thread blocks? To answer the question, we can analyze the pros and cons of each choice. If we use 8×8 blocks, each block would have only 64 threads. We will need $1024/64 = 12$ blocks to fully occupy an SM. However, each SM can only allow up to 8 blocks; thus, we will end up with only $64 \times 8 = 512$ threads in each SM. This limited number implies that the SM execution resources will likely be underutilized because fewer warps will be available to schedule around long-latency operations.

The 16×16 blocks result in 256 threads per block, implying that each SM can take $1024/256 = 4$ blocks. This number is within the 8-block limitation and is a good configuration as it will allow us a full thread capacity in each SM and a maximal number of warps for scheduling around the long-latency operations. The 32×32 blocks would give 1024 threads in each block, which exceeds the 512 threads per block limitation of this device. Only 16×16 blocks allow a maximal number of threads assigned to each SM.

[3] Note that this is an over-simplified exercise. As we will explain in Chapter 4, Memory and data locality, the usage of other resources such as registers and shared memory must also be considered when determining the most appropriate block dimensions. This exercise highlights the interactions between the limit on number of blocks and the limit on the number of threads that can be assigned to each SM.

3.8 SUMMARY

The kernel execution configuration parameters define the dimensions of a grid and its blocks. Unique coordinates in `blockIdx` and `threadIdx` allow threads of a grid to identify themselves and their domains of data. It is the responsibility of the programmer to use these variables in kernel functions so that the threads can properly identify the portion of the data to process. This model of programming compels the programmer to organize threads and their data into hierarchical and multidimensional organizations.

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

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

Once a block is assigned to an SM, it is further partitioned into warps. All threads in a warp have identical execution timing. At any time, the SM executes instructions of only a small subset of its resident warps. This condition allows the other warps to wait for long-latency operations without slowing down the overall execution throughput of the massive number of execution units.

3.9 EXERCISES

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

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

B. Write a kernel that has each thread to produce one output matrix element. Fill in the execution configuration parameters for this design.
C. Write a kernel that has each thread to produce one output matrix row. Fill in the execution configuration parameters for the design.
D. Write a kernel that has each thread to produce one output matrix column. Fill in the execution configuration parameters for the design.
E. Analyze the pros and cons of each kernel design above.

2. A matrix–vector multiplication takes an input matrix B and a vector C and produces one output vector A. Each element of the output vector A is the dot product of one row of the input matrix B and C, i.e., $A[i] = \sum^{j} B[i][j] + C[j]$. For simplicity, we will only handle square matrices whose elements are single-precision floating-point numbers. Write a matrix–vector multiplication kernel and a host stub function that can be called with four parameters: pointer-to-the-output matrix, pointer-to-the-input matrix, pointer-to-the-input vector, and the number of elements in each dimension. Use one thread to calculate an output vector element.

3. If the SM of a CUDA device can take up to 1536 threads and up to 4 thread blocks. Which of the following block configuration would result in the largest number of threads in the SM?
A. 128 threads per block
B. 256 threads per block
C. 512 threads per block
D. 1024 threads per block

4. For a vector addition, assume that the vector length is 2000, each thread calculates one output element, and the thread block size is 512 threads. How many threads will be in the grid?
A. 2000
B. 2024
D. 2048
D. 2096

5. With reference to the previous question, how many warps do you expect to have divergence due to the boundary check on vector length?
A. 1
B. 2
C. 3
D. 6

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

7. With reference to the previous question, how many idle threads do you expect to have?

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

9. Indicate which of the following assignments per multiprocessor is possible. In the case where it is not possible, indicate the limiting factor(s).
   - **A.** 8 blocks with 128 threads each on a device with compute capability 1.0
   - **B.** 8 blocks with 128 threads each on a device with compute capability 1.2
   - **C.** 8 blocks with 128 threads each on a device with compute capability 3.0
   - **D.** 16 blocks with 64 threads each on a device with compute capability 1.0
   - **E.** 16 blocks with 64 threads each on a device with compute capability 1.2
   - **F.** 16 blocks with 64 threads each on a device with compute capability 3.0

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

11. A student mentioned that he was able to multiply two 1024×1024 matrices by using a tiled matrix multiplication code with 32×32 thread blocks. He is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

This page intentionally left blank