Chapter 1
Introduction

Keywords: heterogeneous parallel computing, GPU Computing, throughput-oriented design, latency-oriented design, Amdahl's Law, scalability, data parallelism, memory bandwidth, parallel programming languages, parallel programming models, computational thinking

CHAPTER OUTLINE

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

This drive, however, has slowed since 2003 due to energy consumption and heat dissipation issues that limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Since then, virtually all microprocessor vendors have switched to models

where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power. This switch has exerted a tremendous impact on the software developer community [Sutter2005].

Traditionally, the vast majority of software applications are written as sequential programs that are executed by processors whose design was envisioned by von Neumann in his seminal report in 1945 [vonNeumann1945]. The execution of these programs can be understood by a human sequentially stepping through the code. Historically, most software developers have relied on the advances in hardware to increase the speed of their sequential applications under the hood; the same software simply runs faster as each new 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 Such expectation is no longer valid from this day onward.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 [Sutter2005]. The practice of parallel programming is by no means new. The high-performance computing community has been developing parallel programs for decades. These programs 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 [Hwu2008]. The *multi-core* trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multi-cores began with two-core processors with the number of cores increasing

with each semiconductor process generation. A current exemplar is a recent *Intel*™ multi-core 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 10s of 1000s of threads, executing in a large number of simple, in-order pipelines. Many-threads processors, especially the GPUs, have led the race of floating-point performance since 2003. As of 2016, the ratio of peak floating-point calculation throughput between many-thread GPUs and multi-core 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 multi-core CPUs. The answer lies in the differences in the fundamental design philosophies between the two types of processors, as illustrated in Figure 1.1. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation throughput. As of 2016, the high-end general-purpose multi-core microprocessors typically have eight or more large processor

cores and many mega bytes 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 DRAM (Dynamic Random Access Memory) because of graphics frame buffer requirements and the relaxed memory model (the way various system software, applications, and I/O devices expect how their memory accesses work). 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.

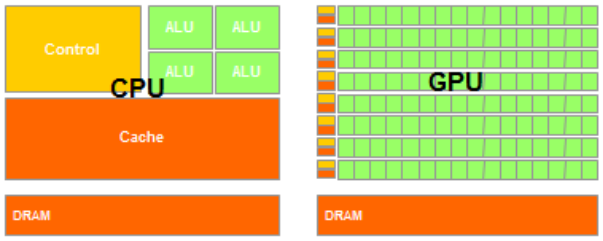


Figure 1.1. CPUs and GPUs have fundamentally different design philosophies.

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 (Chapter 19), and C++AMP (Appendix D).

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

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 more than 1 billion CUDA enabled GPUs in use to date. Such a large market presence has made these GPUs economically attractive targets for application developers.

Another important decision factor is practical form factors and easy accessibility. Until 2006, parallel software applications usually ran on data center servers or departmental clusters. But such execution environments tend to limit the use of these applications. For example, in an application such as medical imaging, it is fine to publish a paper based on a 64-node cluster machine. But actual clinical applications on MRI machines have been based on some combination of a PC and special hardware accelerators. The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks of compute server boxes into clinical settings, while this is common in academic departmental settings. In fact, 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 makes it possible to have 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, GPU support for the IEEE Floating-Point Standard has become comparable with that of the CPUs. As a result, one can expect that more numerical applications will be ported to GPUs and yield comparable result values as the CPUs. Up to 2009, a major remaining issue was that the GPUs floating-point arithmetic units were primarily single precision. Applications that

truly require double precision floating-point were not suitable for GPU execution. However, this has changed with the recent GPUs whose double precision execution speed approaches about half of 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 (FPA), 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 API (Application Programming Interface) functions to access the processing units, meaning that OpenGL or Direct3D techniques were needed to program these chips. Stated more simply, a computation must be expressed as a function that paints a pixel in some way in order to execute on these early GPUs. This technique was called GPGPU, for General Purpose Programming using a Graphics Processing Unit. Even with a higher level programming environment, the underlying code still needs to fit into the APIs that are designed to paint pixels. These APIs limit the kinds of applications that one can actually write for early GPUs. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent research results.

But everything changed in 2007 with the release of CUDA [NVIDIA2007]. NVIDIA 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

Figure 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 Figure 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 Figure 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 GDDR (Graphics Double Data Rate) SDRAM (Synchronous DRAM), referred to as Global Memory in Figure 1.2. These 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 product such as NVIDIA's Pascal architecture, may use HBM (High-Bandwidth Memory) or HBM2 architecture. For brevity, we will simply refer to all of these types of memory as DRAM for the rest of the book.

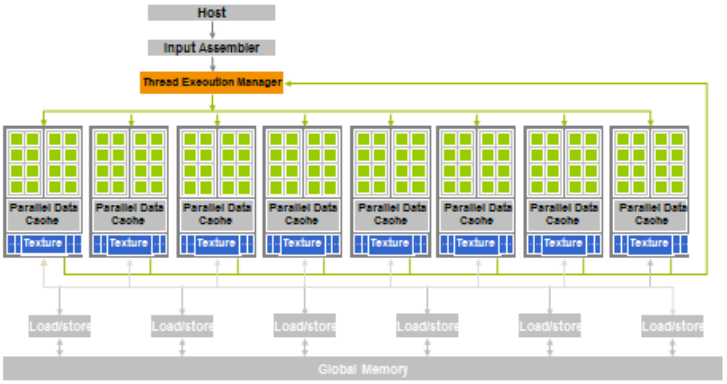


Figure 1.2. Architecture of a CUDA-capable GPU

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 40GB/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 5,000 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 ask why applications will continue to demand increased speed. Many applications that we have today seem to be running quite fast enough. As we will discuss in the case study chapters (Chapters 14, 15, and 16), 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 includes what we call "data parallelism," it's often a 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 to science and medicine.

For applications such as video and audio coding and manipulation, consider our satisfaction with digital high-definition (HD) TV vs. older NTSC TV. Once we experience the level of details in an HDTV, it is very hard to go back to older technology. But consider all the processing that's needed for that HD TV. It is a

very parallel process, as are 3D imaging and visualization. In the future, new functionalities such as view synthesis and high-resolution display of low resolution videos will demand more computing power in the TV. At the consumer level, we will begin to have an increasing number of video and image processing applications that improve the focus, lighting, and other key aspects of the pictures and videos.

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

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

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

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

How many times 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 speed up 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 speedup 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.

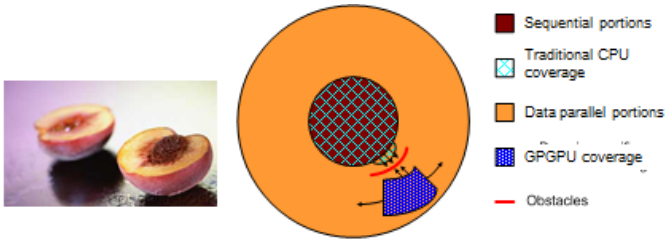


Figure 1.3. Coverage of sequential and parallel application portions

Figure 1.3 illustrates the main parts of a typical application. Much of a real application's code tends to be sequential. These sequential parts are illustrated as the "pit" area of the peach: trying to apply parallel computing techniques to these portions is like biting into the peach pit - not a good feeling! These portions are very hard to parallelize. CPUs tend to do a very good job on these portions. The good news is that these portions, although they can take up a large portion of the code, tend to account for only a small portion of the execution time of super-applications.

Then come what we call the "peach meat" portions. These portions are easy to parallelize, as are some early graphics applications. Parallel programming in heterogeneous computing systems can drastically improve the ~~quality~~ speed of these applications. As illustrated in Figure 1.3 early GPGPUs cover only a small portion of the meat section, which is analogous to a small portion of the most exciting applications. As we will see, the CUDA programming 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, 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.

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 do so much more work than their sequential counter parts that they ended up running slower for large input data sets.

Second, the execution speed of many applications are 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 require non-intuitive way 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 [Mattson2005]. The ones that are the most widely used are Message Passing Interface (MPI) [MPI2009] for scalable cluster computing and OpenMP [Open2005] for shared memory multiprocessor systems. Both have become standardized programming interfaces supported by major computer vendors. An OpenMP implementation consists of a compiler and a runtime. A programmer specifies directives (commands) and pragmas (hints) about a loop to the OpenMP compiler. With these directives and pragmas, OpenMP compilers generate parallel code. The runtime system supports the execution of the parallel code by managing parallel threads and resources. OpenMP was originally designed for CPU execution. More recently, a variation called OpenACC (Chapter 19) 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. However, effective programming in OpenACC still requires the programmers to understand all the detailed parallel programming concepts involved. Because CUDA gives programmers explicit control of these parallel programming details, it is an excellent learning vehicle even for someone who would like to use OpenMP and OpenACC as their primary programming interface. Furthermore, from our experience, OpenACC compilers are still evolving and improving. Many programmers will likely need to use ~~CDUA~~ CUDA style interfaces for parts where OpenACC compilers fall short.

MPI is a model where computing nodes in a cluster do not share memory [MPI2009]. All data sharing and interaction must be done through explicit message passing. MPI has been successful in high-performance computing (HPC). Applications written in MPI have run successfully on cluster computing systems with more than 100,000 nodes. Today, many HPC clusters employ heterogeneous CPU/GPU nodes. While CUDA is an effective interface with each node, most application developers need to use MPI to program at the cluster level. 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 [GCN2010], 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 maintain coherence and automatically perform optimized data transfer operations on behalf of the programmer on a need basis. Such support significantly reduces the programming complexity involved in overlapping data transfer with computation and I/O

activities. As will be discussed later in Chapter 20, 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 Compute Language (OpenCL) [Khronos2009]. Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors. In comparison to CUDA, OpenCL relies more on APIs and less on language extensions. 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 explained 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're going to dedicate many pages to techniques for developing *high-performance* parallel programs. And, we believe that it will become easy once you develop the right insight and go about it the right way. In particular, we will focus on **computational thinking** [Wing2006] techniques that will enable you to think about problems in ways that are amenable to high-performance parallel computing.

Note that hardware architecture features have constraints. High-performance parallel programming on most processors will require some knowledge of how the hardware works. It will probably take 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 our discussions on high-performance parallel programming techniques.

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

Much technical knowledge will be required to achieve these goals, so we will cover quite a few principles and patterns [Mattson2004] of parallel programming in this book. We will not be teaching these principles and patterns on their own. We will teach them in the context of parallelizing useful applications. We cannot guarantee that we will cover all of them, however, so we have selected the most useful and well-proven techniques to cover in detail. To complement your knowledge and expertise, we include a list of recommended literature. We are now ready to give you a quick overview of the rest of the book.

1.8. Organization of the Book

Chapter 2 introduces data parallelism and the CUDA C programming. This chapter relies on the fact that students have had previous experience with C programming. It first introduces CUDA C as a simple, small extension to C that supports heterogeneous CPU/GPU joint computing and the widely used 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

(Application Programming Interface) function to allocate memory on the parallel computing device, (3) using an API function to transfer data to the parallel computing device, (4) developing a kernel function that will be executed by threads in the parallelized part, (5) launching a kernel function for execution by parallel threads, and (6) eventually transferring the data back to the host processor with an API function call.

While the objective of Chapter 2 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 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 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 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 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 through 12 present six important parallel computation patterns that give the readers more insight into parallel programming techniques and parallel execution mechanisms. Chapter 7 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 presents reduction tree and prefix sum, or scan, an important parallel computing pattern that coverts sequential computation into parallel computation. We also use this pattern to introduce the concept of work-efficiency in parallel algorithms. Chapter 9 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 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 introduces merge sort, and dynamic input data identification and organization. Chapter 12 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 we learn from concrete examples. That is, we must first learn the concepts in the context of a particular programming model, which provides us with solid footing when we generalize our knowledge to other programming models. As we do so, we can draw on our concrete experience from the CUDA 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 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 through 16 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 four chapters help the readers put all the materials from the previous chapters together and prepare for their own application development projects. Chapter 14 covers non-Cartesian MRI reconstruction, and how the irregular data affects the program. Chapter 15 covers Molecular Visualization and Analysis. Chapter 16 covers Deep Learning, which is becoming an extremely

important area for GPU computing. We merely provide an introduction, and leave more in-depth discussion to other sources.

Chapter 17 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 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.

Formatted: Left, Don't adjust space between Latin and Asian text, Don't adjust space between Asian text and numbers

Formatted: Font: (Default) Times New Roman

Formatted: Font: (Default) Segoe UI, 10 pt

Chapter 19 covers Parallel Programming with OpenACC. OpenACC is a directive-based high level programming approach with 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 was for a parallel programmer to get started.

Chapters 20 and 21 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

[Hwu2008] W. W. Hwu, K. Keutzer, T. Mattson, “The Concurrency Challenge,” IEEE Design and Test of Computers, July/August 2008, pp. 312-320.

[Khronos2009] The Khronos Group, “The OpenCL Specification version 1.0,” http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[Mattson2004] T. G. Mattson, B.A. Sanders, and B.L. Massingill, "Patterns of Parallel Programming," Addison-Wesley Professional, 2004.

[MPI2009] 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.

[NVIDIA2007] NVIDIA Corporation, "CUDA Programming Guide," February 2007.

[GCN2010] I. Gelado, J. Cabezas, N. Navarro, J.E. Stone, S. J. Patel, W.W. Hwu, "An Asynchronous Distributed Shared Memory Model for Heterogeneous Parallel Systems," International Conference on Architectural Support for Programming Languages and Operating Systems," March 2010.

Technical Report, IMPACT Group, University of Illinois, Urbana-Champaign.

[Open2005] OpenMP Architecture Review Board, "OpenMP application program interface," May 2005.

[Sutter2005] Sutter, Herb, Larus, and James, Software and the Concurrency Revolution, in *ACM Queue*, vol. 3, no. 7, pp. 54-62, September 2005.

[vonNeumann1945] John von Neumann, "First Draft of a Report on the EDVAC," in Goldstine, Herman H. (1972). *The Computer: from Pascal to von Neumann*. Princeton, New Jersey: Princeton University Press. ISBN 0-691-02367-0.

[Wing2006] Jeannette Wing, "Computational Thinking," Communications of the ACM, March 2006, Vol. 49, No. 3.

Chapter 2

Data Parallel Computing

With special contribution from David Luebke

Keywords: data parallelism, scalable parallel program, thread, kernel, API, RGB, greyscale, kernel launch, execution configuration n parameters, data transfer, error handling, stub function, SPMD

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

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 data, 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

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 non-bonding forces, non-bonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large 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.

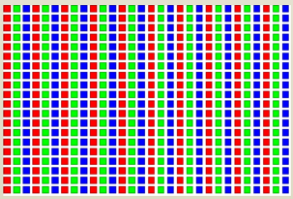
independent computations in parallel to complete the overall job faster, often much faster.

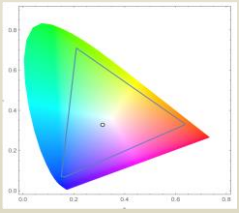
We will use image processing as a source of running examples in the next chapters. Let us illustrate the concept of data parallelism with the color-to-greyscale conversion example mentioned above. Figure 2.1 shows a color image (left side) consisting of many pixels, each containing a red, green, and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

Figure 2.1 Conversion of a color image to a grey–scale image

RGB Color Image Representation

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





The actual allowable mixtures of these three colors vary across industry-specified color spaces. Here, the valid combinations of the three colors in the AdbobeRGB™ color space are shown as the interior of the triangle. The vertical coordinate (y value) and horizontal coordinate (x value) of each mixture show the fraction of the pixel intensity that should be G and R. The remaining fraction (1-y–x) of the pixel intensity 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).

To convert the color image (left side of Figure 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 Figure 2.2. For example, *O*[0] is generated by calculating the weighted sum the RGB values in *I*[0] according to the formula above; *O*[1] by calculating the weighted sum of the RGB values in *I*[1], *O*[2] by calculating the weighted sum of the RGB values in *I*[2], and so on. None of these per-pixel computations depends on each other; all of them can be performed independently. Clearly 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.

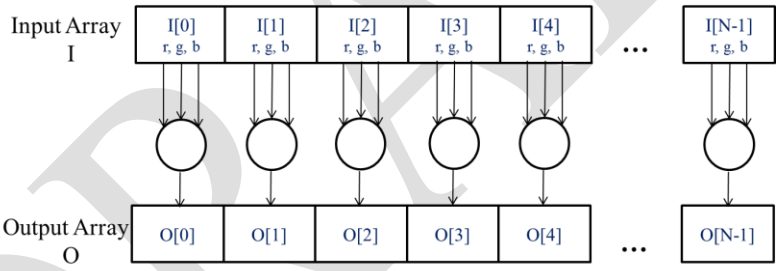


Figure 2.2 The pixels can be calculated independently of each other during color to greyscale conversion.

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 co-existence 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.

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 Figure 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 [SSH2008].

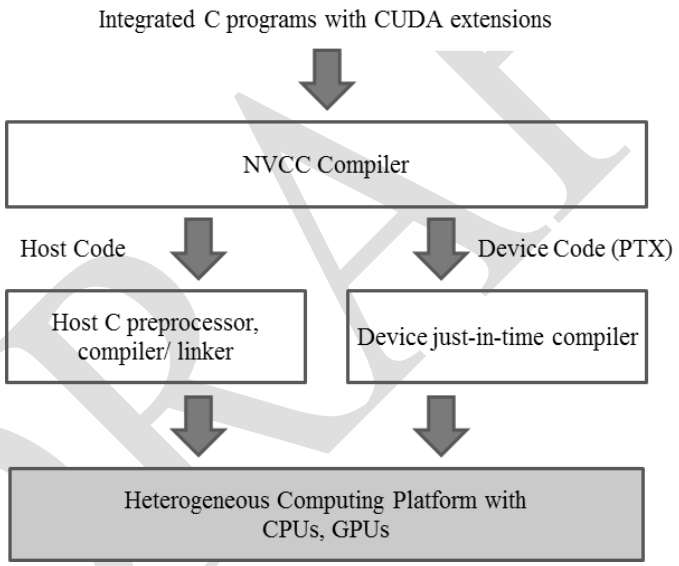


Figure 2.3 Overview of the compilation process of a CUDA C Program

The execution of a CUDA program is illustrated in Figure 24. 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 thread are the primary vehicle of parallel execution in a CUDA platform. Figure 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 Figure 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.

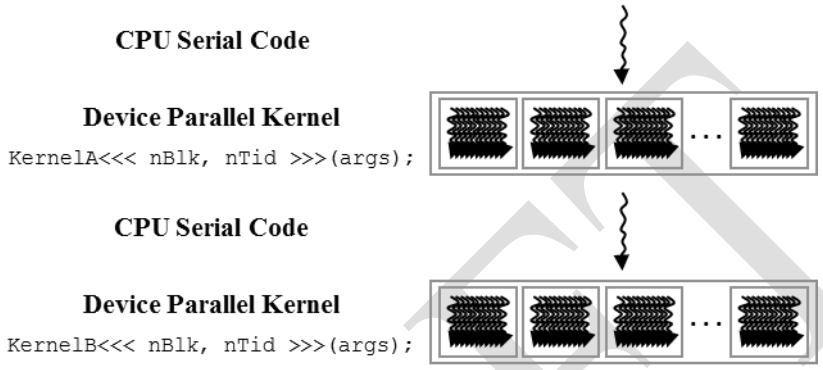


Figure 2.4 Execution of a CUDA program

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 runtime 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. Figure 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 Figure 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 vectorAdd function are pre-fixed 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.

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

int main()
{
    // Memory allocation for arrays A, B, and C
    // I/O to read A and B, N elements each
    ...
    vecAdd(A, B, C, N);
}
```

Figure 2.5 A simple traditional vector addition C code example

The vecAdd function in Figure 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 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 Figure 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.

Pointers in the C Language

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

float V;

a pointer variable P can be declared with:

*float *P;*

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

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

In Figure 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 0^{th} 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.

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

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

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

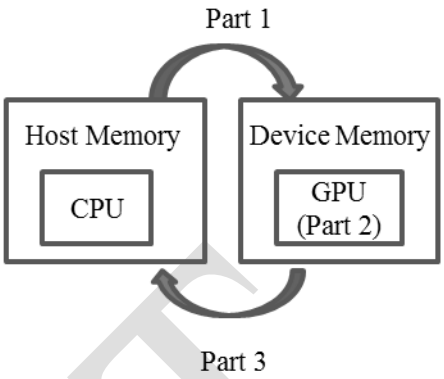


Figure 2.6 Outline of a revised vecAdd function that moves the work to a device

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 GTX480 comes with up to 4 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.

[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.

This corresponds to Part 1 of Figure 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 Figure 2.6. The CUDA runtime system provides Application Programming Interface (API) functions to perform these activities on behalf of the programmer. From this point on, we will simply say that a piece of data is transferred from host to device as shorthand for saying that the data is copied from the host memory to the device memory. The same holds for the opposite direction.

Figure 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 Figure 2.7. There are more device memory types than shown in Figure 2.7. Constant memory can be accessed in a read-only manner by device functions, which will be described in Chapter 7. We will also discuss the use of registers and shared memory in Chapter 4. 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 pre-initialized by the runtime system and are typically read-only in the program. The programmers should refrain from using these variables for any other purposes.

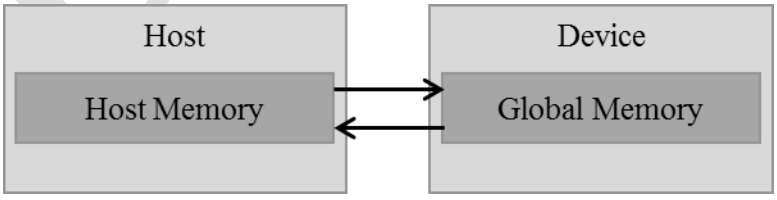


Figure 2.7 Host memory and device global memory

In Figure 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.

Figure 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 runtime library malloc function. This is intentional; CUDA is C with minimal extensions. CUDA uses the standard C runtime library malloc function to manage the host memory and adds cudaMalloc as an extension to the C runtime library. By keeping the interface as close to the original C runtime libraries as possible, CUDA minimizes the time that a C programmer spends to relearn the use of these extensions.

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

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

Figure 2.8 CUDA API functions for managing device global memory

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

[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.

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 Figure 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 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 error during runtime.

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

cudaMemcpy()

- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

Figure 2.9 CUDA API function for data transfer between host and device

Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device. This is accomplished by calling one of the CUDA API functions. Figure 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]

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 value 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 Figure 2.5 calls vecAdd, which is also executed on the host. The vecAdd function, outlined in Figure 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 Figure 2.10.

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

```
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, Figure 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 Figure 2.6, the vecAdd function in Figure 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 Figure 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) [Ata1998] parallel programming style, a popular programming style for massively parallel computing systems.[5]

When a program's host code launches a kernel, the CUDA runtime 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 1,024 threads. [6] Figure 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 of threads, the number of threads in a block is available in a built-in blockDim variable.

[5] Note that SPMD is not the same as SIMD (Single Instruction Multiple Data) [Flynn1972]. In an SPMD system, the parallel processing units execute the same program on multiple parts of the data. However, these processing units do not need to be executing the same instruction at the same time. In an SIMD system, all processing units are executing the same instruction at any instant.
[6] Each thread block can have up to 1,024 threads in CUDA 3.0 and beyond. Some earlier CUDA versions allow only up to 512 threads in a block.

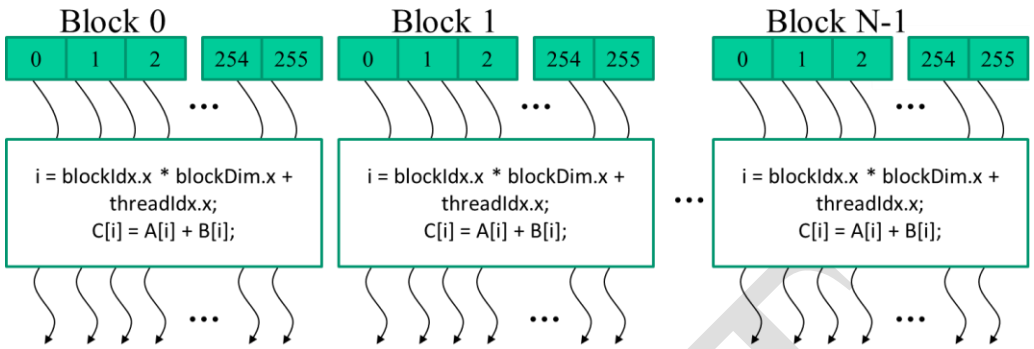


Figure 2.11 All threads in a grid execute the same kernel code

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 reflect the organization of the data. In Figure 2.11, each thread block is organized as a one-dimension array of threads because the data are one-dimensional vectors. The value of the blockDim.x variable specifies the total number of threads in each block, which is 256 in Figure 2.11. In general, the number of 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 Figure 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 Figure 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 Figure 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 Figure 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 ranges from 0 to 255. The i values of threads in block 1 ranges from 256 to 511. The i values of threads in block 2 ranges 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.

Figure 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 functions 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 = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<n) C[i] = A[i] + B[i];
}
```

Figure 2.12 A vector addition kernel function.

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 Figure 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. 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]

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

Figure 2.13 CUDA C keywords for function declaration

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 Figure 2.12, is 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 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

[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.

thread as thread$_{blockIdx.x,\ threadIdx.x}$. 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 Figure 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.

A quick comparison between Figure 2.5 and Figure 2.12 reveals an important insight for CUDA kernels and CUDA kernel launch. The kernel function in Figure 2.12 does not have a loop that corresponds to the one in Figure 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 is referred to as *loop parallelism*, where iterations of the original sequential code are executed by threads in paralle.

Note that there is an if (i<n) statement in addVecKernel in Figure 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.

```
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

When the host code launches a kernel, it sets the grid and thread block dimensions via *execution configuration parameters*. This is illustrated in Figure 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 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 Figure 2.12, the first 1000 threads will perform addition on the 1000 vector elements. The remaining 24 will not.

2.6. Kernel Launch

Figure 2.15 shows the final host code in the `vecAdd` function. This source code completes the skeleton in Figure 2.6. Figures 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 higher speed on larger GPUs. We will revisit this point later in Chapter 3.

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 Figure 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.

```
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.

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 Figure 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 [NVIDIA2016] 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, we will discuss more details of using these variables.

Runtime API

CUDA supports a set of Application Programming Interface (API) functions to provide services to CUDA C programs. The services that we discussed in this chapter are 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.

References

[Patt] Y. N. Patt and S. J. Patel, *Introduction to Computing Systems: from Bits and Gates to C and Beyond,* McGraw Hill Publisher, ISBN 0072467509, 2003.

[SSH2008] J. A. Stratton, S. S. Stone and W. W. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," The 21st International Workshop on Languages and Compilers for Parallel Computing, July 30-31, Canada, 2008. Also available as Lecture Notes in Computer Science 2008.

[Ata1998] M. J. Atallah, ed., *Algorithms and Theory of Computation Handbook*, CRC Press, 1998.

[NVIDIA2016] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide, version 7" March 2016.

[FLYNN1972] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". *IEEE Trans. Comput.* **C-21**: 948.

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 process 2*blockDim.x consecutive elements that form two sections. All threads in each block will first process a section first, each processing one element. They will then all move to the next section, each

processing one element. Assume that variable i should be the index for the first element to be processed by a thread. What would be the expression for mapping the thread/block indices to data index of the first element?

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

4. For a vector addition, assume that the vector length is 8000, each thread calculates one output element, and the thread block size is 1024 threads. The programmer configures the kernel 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?

Chapter 3

Scalable Parallel Execution

With special contribution from Mark Ebersole

Keywords: execution configuration parameters, order, multi-dimensional arrays, multi-dimensional grids, multi-dimensional blocks, row-major order, column-major, linearization, flattening, thread scheduling, transparent scalability, device property query, barrier synchronization, latency tolerance, zero-overhead thread scheduling

CHAPTER OUTLINE

In Chapter 2, we learned to write a simple CUDA C program that launches a kernel and a grid of threads to operate on elements one-dimensional arrays. The kernel specifies the C statements that are executed by each individual thread. As we unleash such massive execution activity, we need to be able to control these activities to achieve desired results, efficiency and speed. In this chapter, we will study several 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. We then 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 to understand high-performance parallel applications.

3.1. CUDA Thread Organization

All CUDA threads in a grid execute the same kernel function and they rely on coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy: a grid consists of one or more blocks and each block in turn consists of one or more threads. All threads in a block share the same block index, which is available as the value of the blockIdx variable in a kernel. Each thread also 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 available as the values of variables gridDim and blockDim in kernel functions.

Hierarchical Organizations:

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

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

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 number of blocks. The second specifies the dimensions of each block in number of threads. Each such parameter is of dim3 type, which is a C `struct` with three unsigned integer fields, x, y, and z. These three fields specify the sizes of the three dimensions.

For example, 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 the 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 dim3 type and the kernel launch uses the appropriate names. For example, the following statements accomplish the same as the statements above:

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

The grid and block dimensions can also be calculated from other variables. For example, the kernel launch in Figure 2.15 can be written as:

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

This allows the number of blocks to vary with the size of the vectors so that the grid will have enough threads to cover all vector elements. In this example, the programmer chose to fix the block size at 256. The value of variable n at kernel launch time will determine dimension of the grid. If n is equal to 1,000, the grid will consists of four blocks. If n is equal to 4,000, the grid will have 16 blocks. In

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

each case, there will be enough threads to cover all the vector elements. Once vecAddKernel is launched, the grid and block dimensions will remain the same until the entire grid finishes execution.

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

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

Readers who are familiar with the use of structures in C would realize that this "short-hand" 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 allows the compiler to conveniently initialize the x fields of gridDim and blockDim with the values provided in the execution configuration parameters when the shortcut is used.

Within the kernel function, the x field of variables gridDim and blockDim are pre-initialized according to the values of the execution configuration parameters. For example, if n is equal to 4,000, references to gridDim.x and blockDim.x in the vectAddkernel kernel will result in 16 and 256 respectively. Note that unlike the dim3 variables in the host code, the names of these variables within the kernel functions are part of the CUDA C specification and cannot be changed. That is, the gridDim and blockDim variables 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, the blockIdx.z value from 0 to gridDim.z-1.

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

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

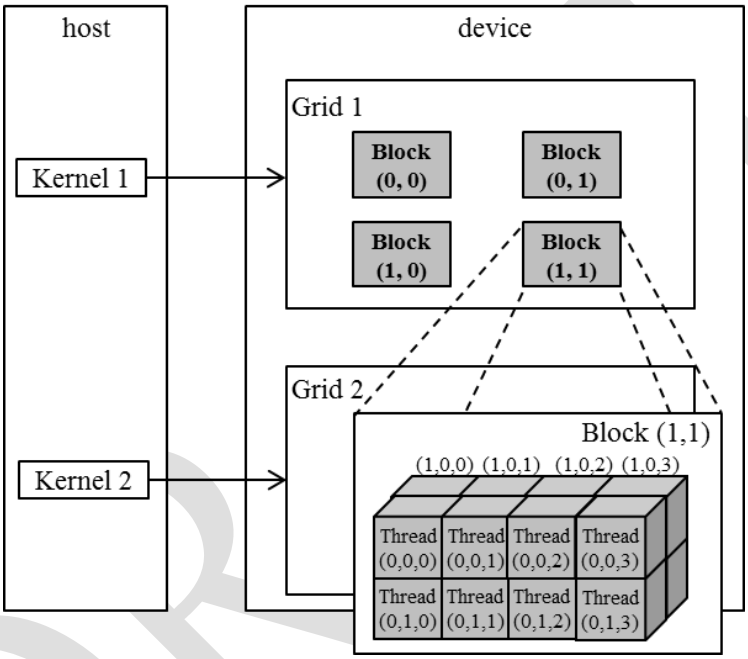


Figure 3.1 *A multi-dimensional example of CUDA grid organization*

Note that the grid can have higher dimensionality than its blocks and vice versa. For example, Figure 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>>>(…);
```

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

The grid consists of four blocks organized into a 2×2 array. Each block in Figure 3.1 is labeled with (blockIdx.y, blockIdx.x). For example, Block(1,0) has blockIdx.y=1 and blockIdx.x=0. Note that the ordering of the labels is such that highest dimension comes first. **Note that this block labeling notation is in 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 better when we illustrate the mapping of thread coordinates into data indexes in accessing multi-dimensional data.

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

3.2. Mapping Threads to Multi-Dimensional Data

The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data. For example, 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. Figure 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, which results in 5×4=20 blocks as shown in Figure 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:

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

Note that we have 4 extra threads in the x direction and 2 extra threads in the y direction. That is, we will generate 80×64 threads to process 76×62 pixels. This is similar to the situation where a 1,000-element vector is processed by the 1D kernel vecAddKernel in Figure 2.11 using four 256-thread blocks. Recall that an if-statement is needed to prevent the extra 24 threads from taking effect. Analogously,

we should expect that the picture processing kernel function will have if-statements to test whether the thread indices threadIdx.x and threadIdx.y fall within the valid range of pixels.

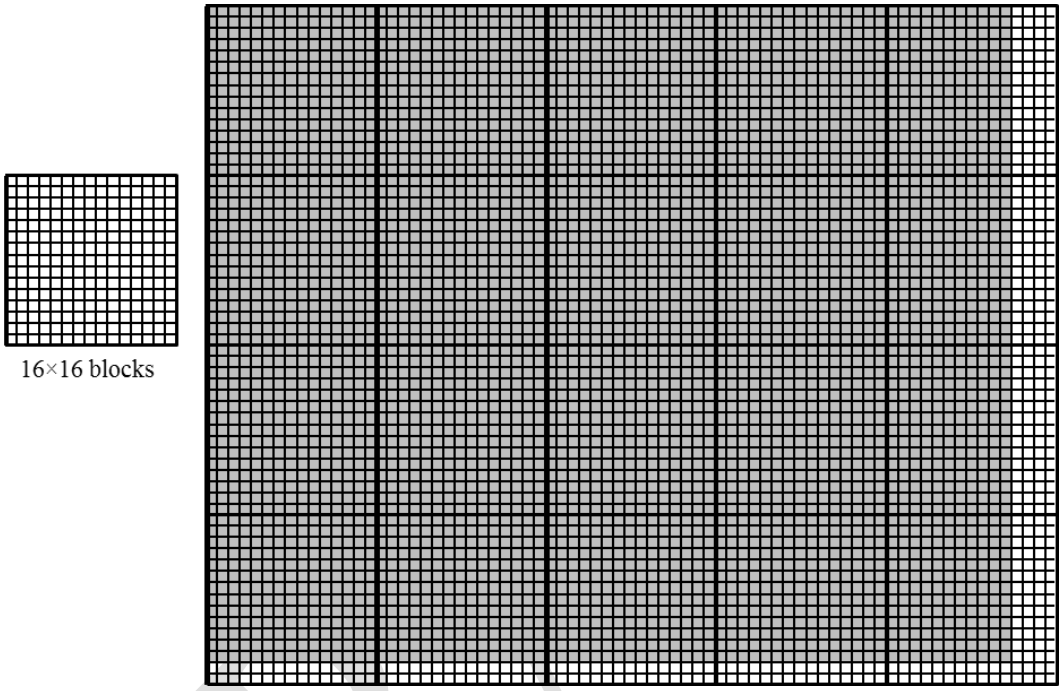


Figure 3.2 *Using a 2D thread grid to process a 76x62 picture P.*

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 the number of pixels in the y direction. We further assume that the input picture data has been copied to the device memory and can be accessed through a pointer variable d_Pin. The output picture has been allocated in the device memory and can be accessed through a pointer variable d_Pout. The following host code can be used to launch a 2D kernel 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. The dimensions of the grid, on the other hand, depend on the

dimensions of the picture. To process a 2,000×1,500 (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. A memory space is usually associated with each running application. The data to be processed by an application and instructions executed for the application are stored in locations in its memory space. Each location typically can accommodate a byte and has an address. Variables that require multiple bytes – 4 bytes for float and 8 bytes for double are stored in consecutive byte locations. The processor gives the starting address (address of the starting byte location) and the number of bytes needed when accessing a data value from the memory space.

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

Before we show the kernel code, we need to first understand how C statements access elements of dynamically allocated multi-dimensional arrays. Ideally, we would like to access d_Pin as a two-dimensional array where an element at the row j and column i can be accessed as d_Pin[j][i]. However, the ANSI C standard based on which CUDA C was developed requires that the number of columns in Pin be known at compile time for 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 why one uses dynamically allocated arrays 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 not known at compile time by design. As a result, programmers need to explicitly linearize, or "flatten", a dynamically allocated two-dimensional array into an equivalent one-dimensional array in the current CUDA C. Note that newer C99 standard allows multi-dimensional syntax for dynamically allocated arrays. It

is likely that future CUDA C versions may support multi-dimensional syntax for dynamically allocated arrays.

In reality, all multi-dimensional arrays in C are linearized. This is due to the use of a "flat" memory space in modern computers (see "Memory Space" sidebar). In the case of statically allocated arrays, the compilers allow the programmers to use higher dimensional indexing syntax such as d_Pin[j][i] to access their elements. Under the hood, the compiler linearizes them into an equivalent one-dimensional array and translates the multi-dimensional indexing syntax into a one-dimensional offset. In the case of dynamically allocated arrays, the current CUDA C compiler leaves the work of such translation to the programmers due to lack of dimensional information at compile time.

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

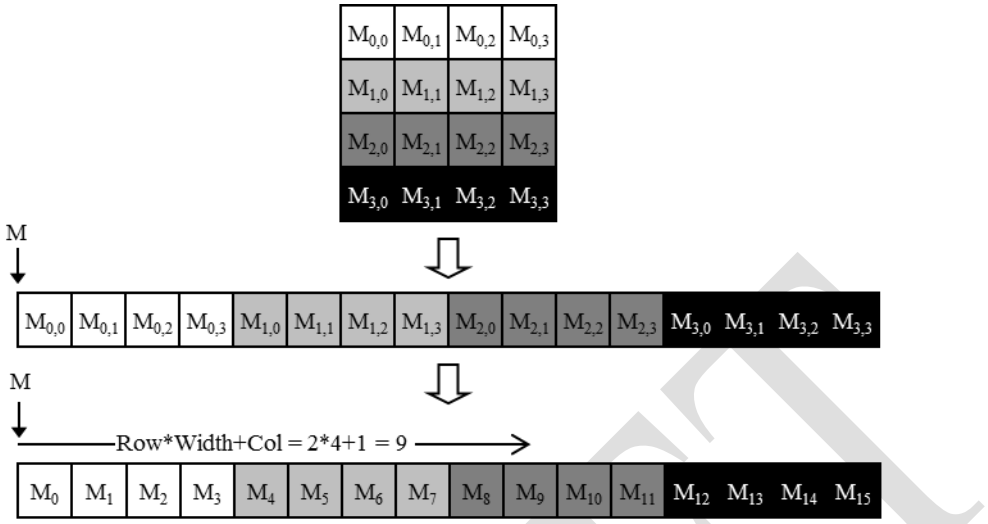


Figure 3.3 *Row-major layout for a 2D C Array. The result is an equivalent 1D array accessed by an index expression j*Width+i for an element that is in the j^{th} row and i^{th} column of an array of Width elements in each row.*

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

Linear Algebra Functions

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

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

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

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

to convert each color pixel to a its greyscale counterpart.

There are a total of blockDim.x*gridDim.x threads in the horizontal direction. As shown 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 also know that there are at least as many threads as the number of pixels in the vertical direction. Therefore, as long as we test and make sure only the threads with both Row and Col values within range, that is (Col<width) && (Row<height), we will be able to cover every pixel in the picture.

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colortoGreyscaleConvertion(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 gray scale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r =  Pin[rgbOffset     ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 1]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 2]; // 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*

Since there are width pixels in each row, we can generate the one-dimensional index for the pixel at row Row and column Col as Row*width+Col. This one-dimensional index greyOffset is the pixel index for Pout since each pixel in the output greyscale image is one byte (unsigned char). Using our 76x62 image example, the linearized one-dimensional index of the Pout pixel calculated by thread(0,0) of block(1,0) with the formula:

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

As for Pin, we need to multiple the grey pixel index by 3 since each pixel is stored as (r, g, b), each of which is one byte. The resulting rgbOffset is gives the starting location of the color pixel in the Pin array. We read the r, g, and b value from the three consecutive byte locations of the Pin array, perform the calculation of the greyscale pixel value, and write that value into the Pout array using greyOffset.

Using our 76x62 image example, the linearized one-dimensional index of the Pin pixel calculated by thread(0,0) of block(1,0) with the formula:

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

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

Figure 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 full into one of four different cases, shown as four shaded areas in Figure 3.5.

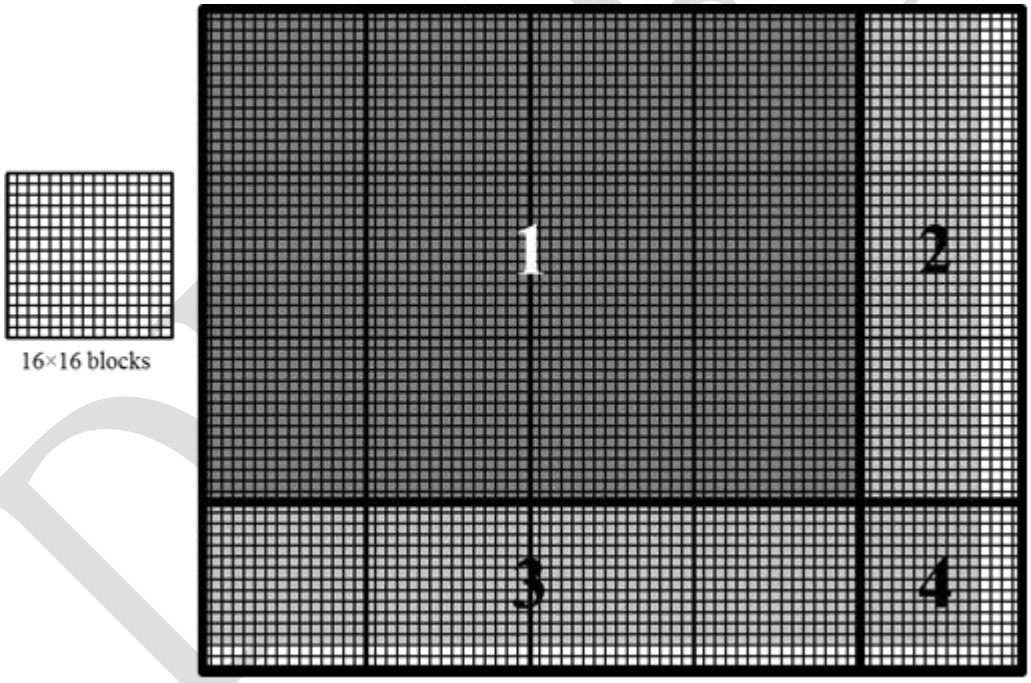


Figure 3.5 Covering a 76×62 picture with 16×blocks

The first area, marked as 1 in Figure 3.5, consists of the threads that belong to the 12 blocks covering the majority of pixels in the picture. Both Col and Row values of these threads are within range; all these threads will pass the if-statement test and process pixels in the dark shaded area of the picture. That is all 16×16=256 threads in each block will process pixels. The second area, marked as 2 in Figure 3.5,

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

The third area, marked as 3 in Figure 3.5, accounts for the 3 lower left blocks covering the medium shaded area of the picture. Although the Col values of these threads are always within range, the Row values of some of them exceed the m value (62). This is because the number of threads in the vertical direction is always multiples of the blockDim.y value chosen by the programmer (16 in this case). The smallest multiple of 16 to cover 62 is 64. As a result, 14 threads in each column will find their Row values within range and will process pixels. On the other hand, 2 threads in each column will fail the if-statement of area 2, and will not process any pixels. 16×14=224 out of the 256 threads will process pixels. The fourth area, marked as 4 in Figure 3.5, contains the threads that cover the lower right lightly shaded area of the picture. Similar to Area 2, 4 threads in each of the top 14 rows will find their Col values out of range. Similar to Area 3, the entire bottom two rows of this block will find their Row values out of range. So, only 14×12=168 of the 16×16=256 threads will process pixels.

We can easily extend our discussion of 2D arrays to 3D arrays by including another dimension when we linearize arrays. This is done by placing each “plane” of the array one after another into the address space. Assume that the programmer uses variables m and n to track the number of columns and rows in a 3D array. 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 in the form of P[Plane*m*n + Row * m + Col]. A kernel processing the 3D P array needs to check whether all the three global indices, Plane, Row, and Col fall within the valid range of the array.

3.3. Image Blur – A More Complex Kernel

We have studied vecAddkernel and colorToGreyscaleConversion where each thread performs only a small number of arithmetic operations on one array element. These kernels serve their purposes well: to illustrate the basic CUDA 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 each other. For the next few chapters, we are going to work on increasingly more complex examples that exhibit these characteristics. We will start with an image blurring function.

Figure 3.6 An original image and a blurred version.

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

Mathematically, an image blurring function calculates the value of an output image pixel as a weighted sum of a patch of pixels encompassing the pixel in the input image. As we will learn in Chapter 7, Parallel patterns: 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 NxN 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, as is common in a convolution blurring approach such as Gaussian Blur.

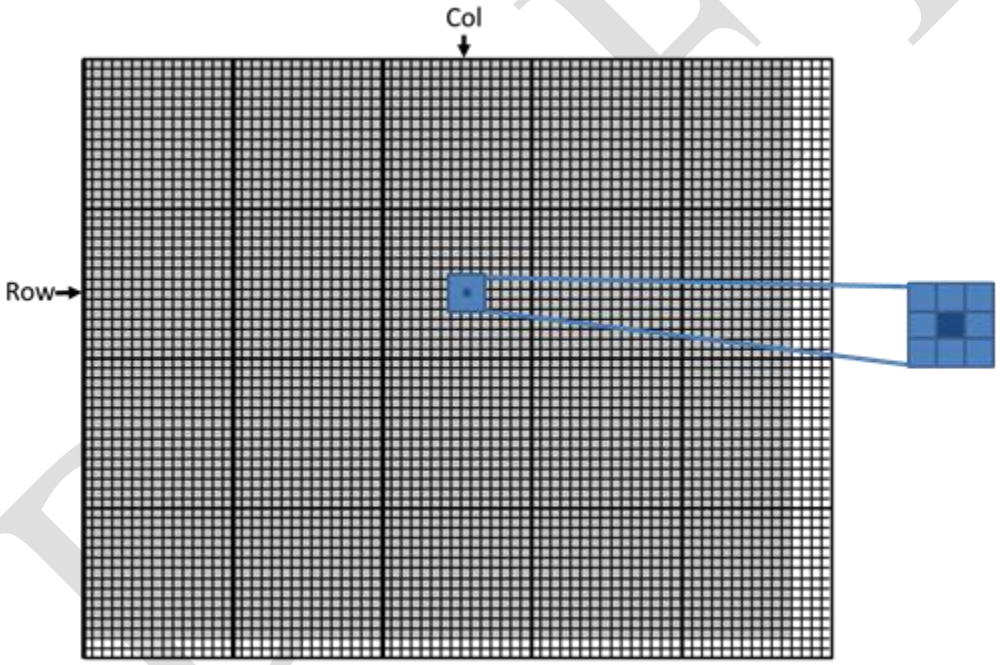


Figure 3.7 Each output pixel is the average of a patch of pixels in the input image.

Figure 3.7 shows an example using a 3x3 patch. When calculating an output pixel value at (Row, Col) position, we see that the patch is centered at the input pixel located at the (Row, Col) position. The 3x3 patch spans three rows (Row-1, Row, Row+1) and three columns (Col-1, Col, Col+1). For example, the coordinates of the

nine pixels for calculating the output pixel at (25, 50) are (24, 49), (24, 50), (24, 51), (25, 49), (25, 50), (25, 51), (26, 49), (26, 50), and (26, 51).

```
  __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.

Figure 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 indices. We also see the familiar if-statement that verifies that both Col and Row are within the valid range according to the height and width of the image. Only the threads whose Col and Row indices are both within value ranges will be allowed to participate in the execution.

As shown in Figure 3.7, the Col and Row values also gives the central pixel location of the patch used for calculating the output pixel for the thread. The nested for-loops at Lines 3 and 4 in Figure 3.8 iterate through all the pixels in the patch. We assume that the program has a defined constant BLUR_SIZE. The value of BLUR_SIZE is set such that 2*BLUR_SIZE gives the number of pixels on each side of the patch. For example, for a 3x3 patch, BLUR_SIZE is set to 1 whereas for a 7x7 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 3x3 patch example, the BLUR_SIZE is 1. For the thread that calculates output pixel (25, 50), during the first iteration of the outer loop, the CurRow variable is Row-BLUR_SIZE = (25-1) = 24. Thus, during the first iteration of the outer loop, the inner loop iterates through the patch pixels in row 24. The inner loop iterates from column Col-BLUR_SIZE = 50-1 = 49 to Col+BLUR_SIZE = 51 using the CurCol variable. Therefore, the pixels processed in the first iteration of the outer loop are (24, 49), (24, 50), and (24, 51). The reader should verify that in the second iteration of the outer loop, the inner loop iterates through pixels (25, 49), (25, 50), and (25, 51). Finally, the in the third iteration of the outer loop, the inner loop iterates through pixels (26, 49), (26, 50) and (26, 51).

Line 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 fact that one more pixel value has been added into the running sum by incrementing the pixels variable. After all the pixels in the patch are processed, Line 10 calculates the average value of the pixels in the patch by diving 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 Figure 3.9 assuming 3x3 patches. In Case 1, the pixel at the upper left corner is being blurred. Five out of the nine pixels in the intended patch do not exist in the input image. In this case, the Row and Col value of the output pixel is 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), (1,1). Note that for the five pixels that are outside the image, at least one of the values is less than 0. The CurRow<0 and CurCol<0 conditions of the if-statement catch these values and skips 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.

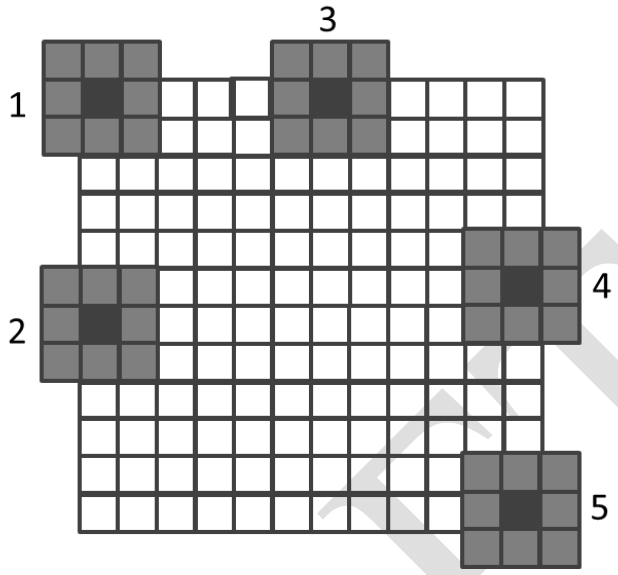


Figure 3.9 Handling boundary conditions for pixels near the edges of the image

The readers should work through the other cases in Figure 3.9 and analyze the execution behavior of the nested loop in the blurKernel. Note that most of the threads will find all the pixels in their assigned 3x3 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. These variations necessitate keeping track of the actual number of pixels accumulated with variable pixels.

3.4. Synchronization and Transparent Scalability

Thus far, we have discussed 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 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 ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase.

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. For example, assume that four friends go to a shopping mall in a car. They can all go to different stores to shop for their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit all the stores of interest. However, barrier synchronization is needed before they leave the mall. They have to wait until all four friends have returned to the car before they can leave. The ones that finish ahead of others need to wait for those who finish later. Without the barrier synchronization, one or more persons can be left in the mall when the car leaves, which can seriously damage their friendship!

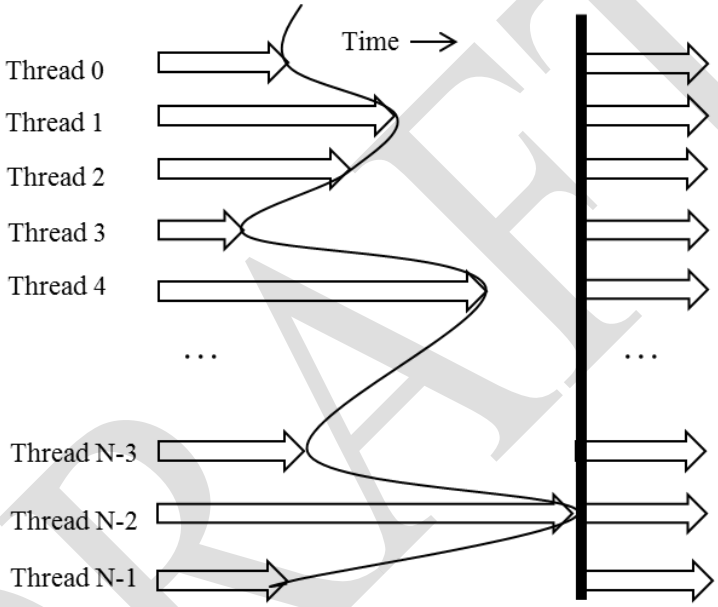


Figure 3.10 An example execution timing of barrier synchronization

Figure 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 that 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 threads in a block execute the path that includes the __syncthreads() or none of them does. For an `if-the-else` statement, if each path has a __syncthreads() statement, either all threads in a block execute the then-path or all of them execute

the else-path. The two __syncthreads() are different barrier synchronization points. If a thread in a block executes the then-path and another executes the else-path, they would be waiting at different barrier synchronization points. They would end up waiting for each other forever. It is the responsibility of the programmers to write their code so that these requirements are satisfied.

The ability to synchronize also imposes execution constraints on threads within a block. These threads should execute in close time proximity with each other to avoid excessively long waiting times. In fact, one needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier. Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to wait forever. CUDA run-time 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 run-time system has secured all the resources needed for all threads in the block to complete execution. When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures the time proximity of all threads in a block and prevents excessive or indefinite waiting time during barrier synchronization.

This leads us to 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 run-time system can execute blocks in any order relative to each other since none of them need to wait for each other. This flexibility enables scalable implementations as shown in Figure 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 at the same time; portrayed as executing two blocks a time on the left hand side of Figure 3.11. In a high-end implementation with more execution resources, one can execute a large number of blocks at the same time; shown as four blocks at a time on the right hand side of Figure 3.11.

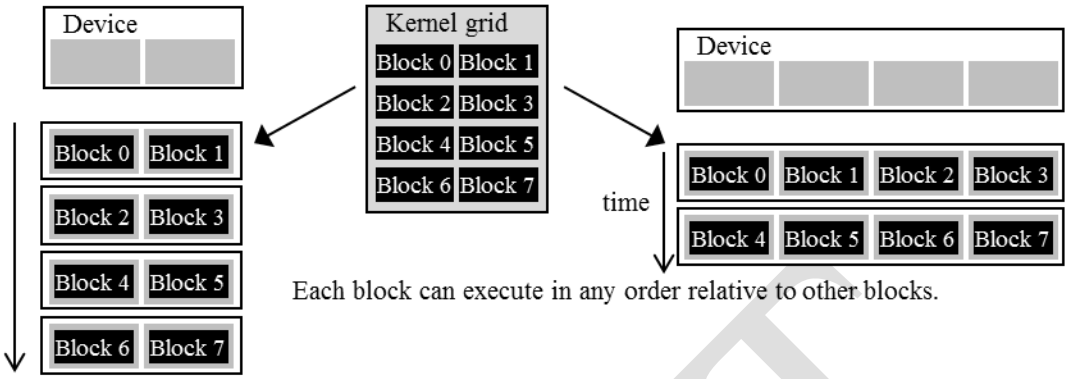


Figure 3.11 Lack of synchronization constraints between blocks enables transparent scalability for CUDA programs

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

3.5. Resource Assignment

Once a kernel is launched, the CUDA run-time system generates the corresponding grid of threads. As we discussed in the previous section, these threads are assigned to execution resources on a block-by-block basis. In the current generation of hardware, the execution resources are organized into Streaming Multiprocessors (SMs). Figure 3.12 illustrates that multiple thread blocks can be assigned to each SM. Each device has a limit on the number of blocks that can be assigned to each SM. For example, a CUDA device may allow up to 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 under the limit. With limited numbers of SMs and limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in a CUDA device. Most grids contain many more blocks than this number. The run-time system maintains a list of blocks that need

to execute and assigns new blocks to SMs as previously assigned blocks complete execution.

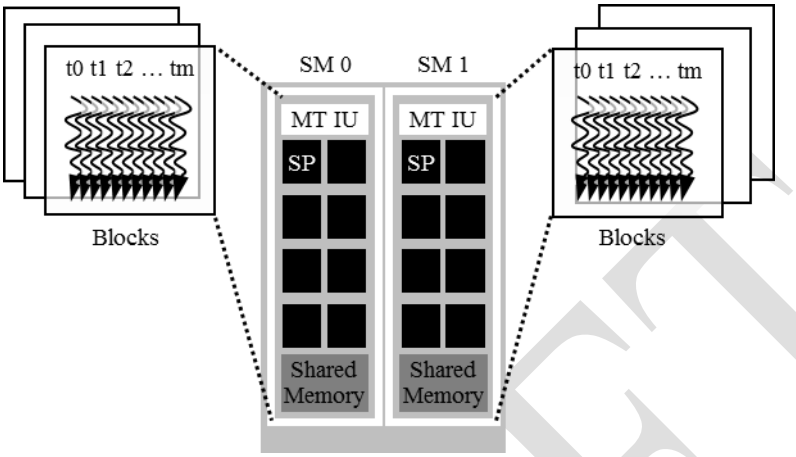


Figure 3.12 Thread block assignment to Streaming Multiprocessors (SMs)

Figure 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 indices 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. In more recent CUDA device designs, up to 8 blocks and 1,536 threads can be assigned to each SM. This could be in the form of 6 blocks of 256 threads each, 3 blocks of 512 threads each, etc. If the device only allows up to 8 blocks in an SM, it should be obvious that 12 blocks of 128 threads each is not a viable option. If a CUDA device has 30 SMs and each SM can accommodate up to 1,536 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 find out the number of SMs in a device and the number of blocks and threads that can be assigned to each SM? Obviously, there are also other resources that we have not discussed so far but can be relevant to the execution of a CUDA application. In general, many modern applications are designed to execute on a wide variety of hardware systems. There is often a need for the application to *query* the available resources and capabilities of the

Resource and Capability Queries

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

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

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

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

underlying hardware in order to take advantage of the more capable systems while compensating for the less capable systems.

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

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

While it may not be obvious, a modern PC system often have two or more CUDA devices. This is because many PC systems come with one or more “integrated” GPUs. These GPUs are the default graphics units and provide rudimentary capabilities and hardware resources to perform minimal graphics functionalities for modern window-based user interfaces. Most CUDA applications will not perform very well on these integrated devices. This would be a reason for the host code to

iterate through all the available devices, query their resources and capabilities, and choose the ones that have enough resources to execute the application with satisfactory performance.

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

```
cudaDeviceProp  dev_prop;

for (i = 0; i < dev_count; i++) {
   cudaGetDeviceProperties( &dev_prop, i);

 //decide if device has sufficient resources and capabilities

}
```

The built-in type cudaDeviceProp is a C struct type with fields that represent the properties of a CUDA device. The reader is referred to the CUDA C Programming Guide for all the fields of the type. We will discuss a few of these fields that are particularly relevant to the assignment of execution resources to threads. We assume that the properties are returned in the 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 gives the maximal number of threads allowed in a block in the queried device. Some devices allow up to 1024 threads in each block and other devices allow fewer. It is possible that future devices may even allow more than 1024 threads per block. Therefore, it is a good idea to query the available devices and determine which ones will allow sufficient number of threads in each block as far as the application is concerned.

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 much larger number of SMs, e.g. 30. If the application requires a large number of SMs to achieve satisfactory performance, it should definitely check this property of the prospective device. Furthermore, the clock frequency of the device

is in dev_prop.clockRate. The combination the clock rate and the number of SMs gives a good indication of the hardware execution capacity of the device.

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

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

The warp is the unit of thread scheduling in SMs. Figure 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 second warp, and so on. In this example, there are three blocks, Block 1, Block 2 and Block 3, all assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.

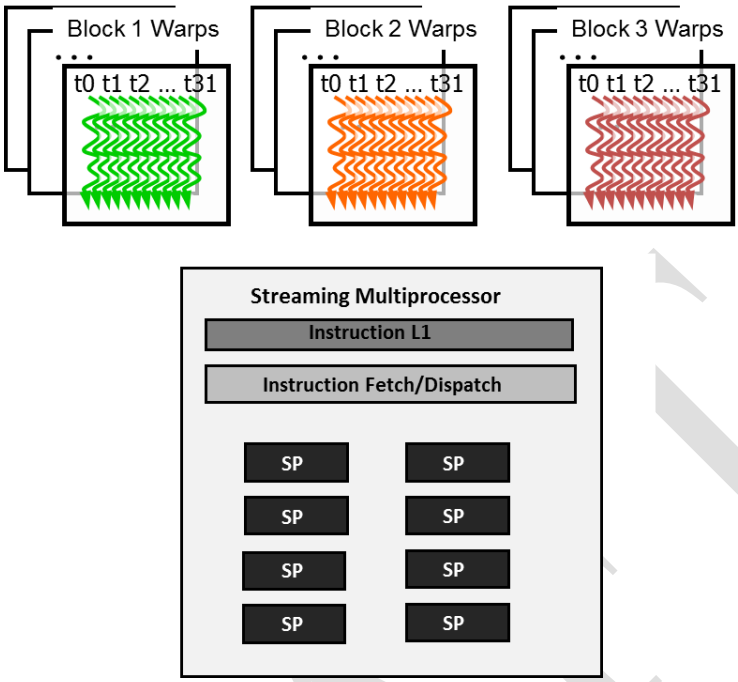


Figure 3.13 Blocks are partitioned into warps for thread scheduling

We can calculate the number of warps that reside in a SM for a given block size and a given number of blocks assigned to each SM. For example, in Figure 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*3 = 24 warps in each SM.

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

Figure 3.13 also shows a number of hardware Streaming Processors (SPs) that actually execute instructions. In general, there are fewer streaming processors than threads assigned to each SM. That is, each SM has only enough hardware to execute instructions from a small subset of all threads assigned to the SM at any point in time. In earlier GPU design, each SM can execute only one instruction for a single warp at any given instant. In more recent designs, each SM can execute

instructions for a small number of warps at any given point in time. In either case, the hardware can execute instructions for a small subset of all warps in the SM. A legitimate question is why 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).

Latency Tolerance

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

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

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

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

Note that warp scheduling is also used for tolerating other types of operation latencies such as pipelined floating-point arithmetic and branch instructions. With

enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long latency operations. The selection of ready warps for execution does not introduce any idle 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 operation latencies is the main reason why GPUs do not dedicate nearly as much chip area to cache memories and branch prediction mechanisms as CPUs. As a result, GPUs can dedicate more of its chip area to floating-point execution resources.

We are now ready to do a simple exercise[3]. Assume that a CUDA device allows up to 8 blocks and 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, since there is a limitation of up to 8 blocks in each SM, we will end up with only 64×8 = 512 threads in each SM. This means that the SM execution resources will likely be underutilized because there will be fewer warps to schedule around long latency operations.

The 16×16 blocks give 256 threads per block. This means that each SM can take 1024/256 = 4 blocks. This is within the 8-block limitation. This is a good configuration since we will have full thread capacity in each SM and 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 the device. Only 16x16 blocks allow maximal number of threads 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 programmer's responsibility to use these variables in kernel functions so that the threads can properly identify the portion of the data to process. This model of programming compels the

[3] Note that this is an over-simplified exercise. As we will explain in Chapter 4, 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.

programmer to organize threads and their data into hierarchical and multi-dimensional organizations.

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

Threads are assigned to SMs for execution on a block-by-block basis. Each CUDA device imposes a potentially different limitation on the amount of resource available in each SM. For example, each CUDA device has 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 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 number. 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 number. Write a matrix-vector multiplication kernel and the host stub function that can be called with four parameters: pointer to the output matrix, pointer to the input matrix, pointer to the input vector, and the number of elements in each dimension. Use one thread to calculate an output vector element.

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

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
   (C) 2048
   (D) 2096

5. If 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 400x900 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. For 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 micro seconds) to execute the sections: 2.0, 2.3, 3.0, 2.8, 2.4, 1.9, 2.6, 2.9 and spend the rest of their time waiting for the barrier. What percentage of the threads' total execution time is spent waiting for the barrier?

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 1024X1024 matrices using a tiled matrix multiplication code with 32×32 thread blocks. He is using a CUDA device that allows up to 512 threads per block and up to 8 blocks per SM. He further mentioned that each thread in a thread block calculates one element of the result matrix. What would be your reaction and why?

Chapter 4

Memory and Data Locality

Keywords: memory bandwidth, memory-bound, on-chip memory, tiling, strip-mining, shared memory, private memory, scope, lifetime, occupancy

CHAPTER OUTLINE

So far, we have learned how to write a CUDA kernel function and how to configure and coordinate its execution by a massive number of threads. In this chapter, we will begin to study how one can organize and position the data for efficient access by a massive number of threads. We learned in Chapter 2 that the data are first transferred from the host memory to the device global memory. In Chapter 3, we studied how to direct the threads to access their portion of the data from the global memory using their block indexes and thread indexes. We have also learned more details about resource assignment and thread scheduling. Although this is a very good start, the CUDA kernels that we have learned so far will likely achieve only a tiny fraction of the potential speed of the underlying hardware. The poor performance is due to the fact that global memory, which is typically implemented with DRAM, tends to have long access latencies (hundreds of clock cycles) and finite access bandwidth. While having many threads available for execution can theoretically tolerate long memory access latencies, one can easily run into a situation where traffic congestion in the global memory access paths prevents all but very few threads from making progress, thus rendering some of the Streaming Multiprocessors (SMs) idle. In order to circumvent such congestion, CUDA provides a number of additional resources and methods for

accessing memory that can remove the majority of traffic to and from the global memory. In this chapter, you will learn to use different memory types to boost the execution efficiency of CUDA kernels.

4.1. Importance of Memory Access Efficiency

We can illustrate the effect of memory access efficiency by calculating the expected performance level of the most executed portion of the image blur kernel code in Figure 3.8, replicated in Figure 4.1. The most important part of the kernel in terms of execution time is the nested for-loop that performs pixel value accumulation with the blurring patch.

```
      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
          }
        }
      }
```

Figure 4.1*: The most executed part of the image blurring kernel in Figure 3.8.*

In every iteration of the inner loop, one global memory access is performed for one floating-point addition. The global memory access fetches an in[] array element. The floating-point add operation accumulates the value of the in[] array element into pixVal. Thus, the ratio of floating-point calculation to global memory access operation is 1 to 1, or 1.0. We will refer to this ratio as the *compute-to-global-memory-access ratio*, defined as the number of floating-point calculations performed for each access to the global memory within a region of a program.

The compute-to-global-memory-access ratio has major implications on the performance of a CUDA kernel. In a high-end device today, the global memory bandwidth is around 1000 GB/s, or 1 TB/s. With four bytes in each single-precision floating-point value, one can expect to load no more than 1000/4=250 giga single-precision operands per second. With a compute-to-global-memory ratio of 1.0, the execution of the image blur kernel will be limited by the rate at which the operands (e.g., the elements of in[])can be delivered to the GPU. We will refer to programs whose execution speed is limited by memory access throughput as *memory bound* programs. In our example, the kernel will achieve no more than 250 giga floating-point operations per second (GFLOPS).

While 250 GFLOPS is a respectable number, it is only a tiny fraction (2%) of the peak single-precision performance of 12 TFLOPS or higher for these high-end devices. In order to achieve a higher level of performance for the kernel, we need to increase the ratio by reducing the number of global memory accesses. To achieve the peak 12 TFLOPS rating of the processor, we need a ratio of 48 or higher. In general, the desired ratio has been increasing in the past few generations of devices as computational throughput has been increasing faster than memory bandwidth. The rest of this chapter introduces a commonly used technique for reducing the number of global memory accesses.

4.2. Matrix Multiplication

Matrix-matrix multiplication, or matrix multiplication in short, between an i×j (i rows by j columns) matrix M and a j×k matrix N produces an i×k matrix P. Matrix multiplication is an important component of the Basic Linear Algebra Subprograms (BLAS) standard (see "Linear Algebra Functions" sidebar in Chapter 3). It is the basis of many linear algebra solvers such as LU decomposition. As we will see, matrix multiplication presents opportunities for reduction of global memory accesses that can be captured with relatively simple techniques. The execution speed of matrix multiplication functions can vary by orders of magnitude depending on the level of reduction of global memory accesses. Therefore, matrix multiplication provides an excellent initial example for such techniques.

When performing a matrix multiplication, each element of the output matrix P is an inner product of a row of M and a column of N. We will continue to use the convention where $P_{Row,Col}$ is the element at Rowth position in the vertical direction and Col$^{th\ position}$ in the horizontal direction. As shown in Figure 4.2, $P_{Row,\ Col}$ (the small square in P) is the inner product of the vector formed from the Rowth row of M (shown as a horizontal strip in M) and the vectorformed from the Colth column of N (shown as a vertical strip in N). The inner product, sometimes called the dot product, of two vectors is the sum of products of the individual vector elements. That is, $P_{Row,Col}=\sum M_{Row,k} * N_{k,Col,}$ for k = 0, 1, … Width-1. For example,

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

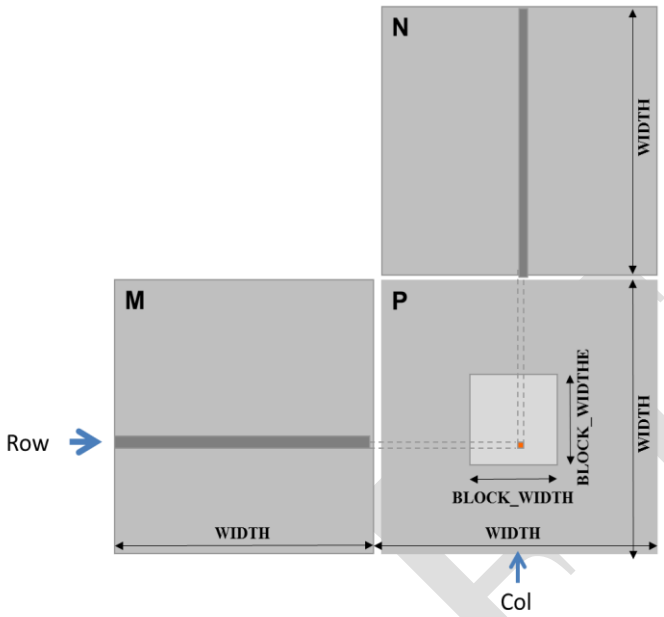


Figure 4.2: *Matrix multiplication using multiple blocks by tiling P*

In our initial matrix multiplication implementation, we map threads to elements of P with the same approach that we used for colorToGreyscaleConversion. That is, each thread is responsible for calculating one P element. The row and column indices for the P element to be calculated by each thread are:

Row = blockIdx.y*blockDim.y+threadIdx.y

and

Col = blockIdx.x*blockDim.x+threadIdx.x.

With this one-to-one mapping, the Row and Col thread indices are also the row and column indices for output array. Figure 4.3 shows the source code of the kernel based on this thread-to-data mapping. The reader should immediately see the familiar pattern of calculating Row, Col, and the if-statement testing if Row and Col are both within range. These statements are almost identical to their counter parts in colorToGreyscaleConversion. The only significant difference is that we are assuming square matrices for matrixMulKernel so we replace both width and height with Width.

The thread-to-data mapping effectively divides P into tiles, one of which is shown as a large square in Figure 4.2. Each block is responsible for calculating one of these tiles.

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

Figure 4.3: *A simple matrix multiplication kernel using one thread to compute one P element*

We now turn our attention to the work done by each thread. Recall that $P_{Row, Col}$ is the inner product of the Rowth row of M and the Colth column of N. In Figure 4.3, we use a for-loop to perform this inner product operation. Before we enter the loop, we initialize a local variable Pvalue to 0. Each iteration of the loop accesses an element from the Rowth row of M and one from the Colth column of N, multiplies the two elements together, and accumulates the product into Pvalue.

Let's first focus on accessing the M element within the for-loop. Recall that M is linearized into an equivalent 1D array where the rows of M are placed one after another in the memory space, starting with the 0th row. Therefore, the beginning element of the 1st row is M[1*Width] because we need to account for all elements of the 0th row. In general, the beginning element of the Rowth row is M[Row*Width]. Since all elements of a row is placed in consecutive locations, the kth element of the Rowth row is at M[Row*Width+k]. This is what we used in Figure 4.3.

We now turn our attention to N. As shown in Figure 4.3, the beginning element of the Colth column is the Colth element of the 0th row, which is N[Col]. Accessing each additional element in Colth column requires skipping over entire rows. This is because the next element of the same column is actually the same element in the next row. Therefore, the kth element of the Colth column is N[k*Width+Col].

After the execution exits the for-loop, all threads have their P element value in the Pvalue variables. Each thread then uses the one-dimensional equivalent index expression

Row*Width+Col to write its P element. Again, this index pattern is similar to that used in the colorToGreyscaleConversion kernel.

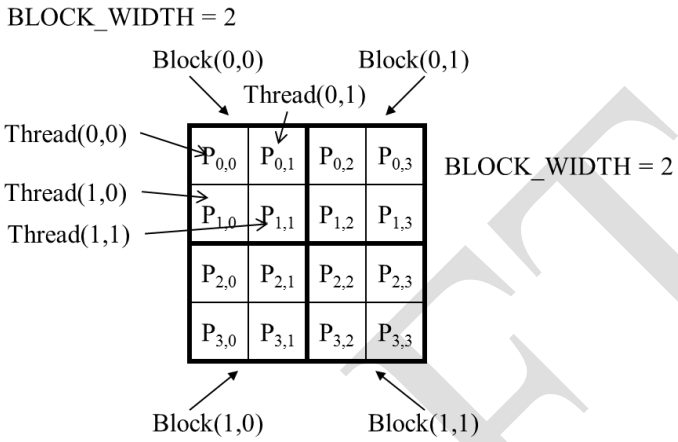


Figure 4.4: *A small execution example of matrixMulKernel*

We now use a small example to illustrate the execution of the matrix multiplication kernel. Figure 4.4 shows a 4×4 P with BLOCK_WIDTH = 2. The small sizes allow us to fit the entire example in one picture. The P matrix is now divided into four tiles and each block calculates one tile. We do so by creating blocks that are 2x2 arrays of threads, with each thread calculating one P element. In the example, thread(0,0) of block(0,0) calculates $P_{0,0}$ whereas thread(0,0) of block(1,0) calculates $P_{2,0}$.

Row and Col in the matrixMulKernel identify the P element to be calculated by a thread. Row also identifies the row of M and Col identifies the column of N as input values for the thread. Figure 4.5 illustrates the multiplication actions in each thread block. For the small matrix multiplication example, threads in block (0,0) produce four dot products. The Row and Col variables of thread(1,0) in block(0,0) are 0*0+1=1 and 0*0+0=0. It maps to $P_{1,0}$ and calculates the dot product of row 1 of M and column 0 of N.

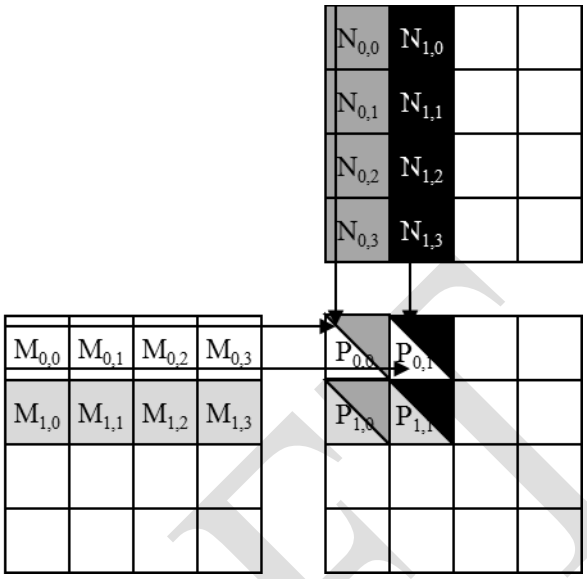


Figure 4.5: *Matrix multiplication actions of one thread block.*

Let's walk through the execution of the for-loop of Figure 4.3 for thread(0,0) in block(0,0). During the 0th iteration (k=0), Row*Width+k = 0*4+0 = 0 and k*Width+Col = 0*4+0 = 0. Therefore the we are accessing M[0] and N[0], which according to Figure 3.3 are the 1D equivalent of $M_{0,0}$ and $N_{0,0}$. Note that these are indeed the 0th elements of row 0 of M and column 0 of N. During the 1st iteration (k=1), Row*Width+k = 0*4+1 = 1 and k*Width+Col = 1*4+0 = 4. We are accessing M[1] and N[4], which according to Figure 3.3 are the 1D equivalent of $M_{0,1}$ and $N_{1,0}$. These are the 1st elements of row 0 of M and column 0 of N.

During the 2nd iteration (k=2), Row*Width+k = 0*4+2 = 2 and k*Width+Col = 8, which results in M[2] and N[8]. Therefore, the elements accessed are the 1D equivalent of $M_{0,2}$ and d_$N_{2,0}$. Finally, during the 3rd iteration (k=3), Row*Width+k = 0*4+3 and k*Width+Col = 12, which results in M[3] and N[12], the 1D equivalent of $M_{0,3}$ and $N_{3,0}$. We now have verified that the for-loop performs inner product between the 0th row of M and the 0th column of N. After the loop, the thread writes P[Row*Width+Col], which is P[0]. This is the 1D equivalent of $P_{0,0}$ so thread(0,0) in block(0,0) successfully calculated the inner product between the 0th row of M and the 0th column of N and deposited the result in $P_{0,0}$.

We will leave it as an exercise for the reader to hand-execute and verify the for-loop for other threads in block(0,0) or in other blocks.

Note that matrixMulKernel can handle matrices of up to 16×65,535 elements in each dimension. In the situation where matrices larger than this limit are to be multiplied, one can divide up the P matrix into sub-matrices whose size can be covered by a grid. We can then use the host code to iteratively launch kernels and complete the P matrix. Alternatively, we can change the kernel code so that each thread calculates more P elements.

We can estimate the effect of memory access efficiency by calculating the expected performance level of the matrix multiplication kernel code in Figure 4.3. The dominating part of the kernel in terms of execution time is the for-loop that performs inner product calculation:

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

In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition. One global memory access fetches an M element and the other fetches an N element. One floating-point operation multiplies the M and N elements fetched and the other accumulates the product into Pvalue. Thus, the compute-to-global-memory-access ratio of the loop is 1.0. From our discussion in Chapter 3, this ratio will likely result in less than 3% utilization of the peak execution speed of the modern GPUs. We need to increase it by at least an order of magnitude in order to achieve good utilization of the computation throughput of modern devices. In the next section, we will show that we can use the special memory types in CUDA devices to accomplish this goal.

4.3. CUDA Memory Types

A CUDA device contains several types of memory that can help programmers to improve compute-to-global-memory-access ratio and thus achieve high execution speed. Figure 4.6 shows these CUDA device memories. At the bottom of the picture, we see global memory and constant memory. These types of memory can be written (W) and read (R) by the host by calling API functions[1]. We have already introduced global memory in Chapter 2. The global memory can be written and read by the device. The constant memory supports short-latency, high bandwidth **read-only access** by the device.

Registers and shared memory in Figure 4.6 are on-chip memories. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner. Registers are allocated to individual threads; each thread can only access its own registers. A kernel function typically uses registers to hold frequently accessed variables that are private to each thread. Shared memory locations are allocated to thread blocks; all threads

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

in a block can access shared-memory variables allocated to the block. Shared memory is an efficient means for threads to cooperate by sharing their input data and intermediate results. By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable.

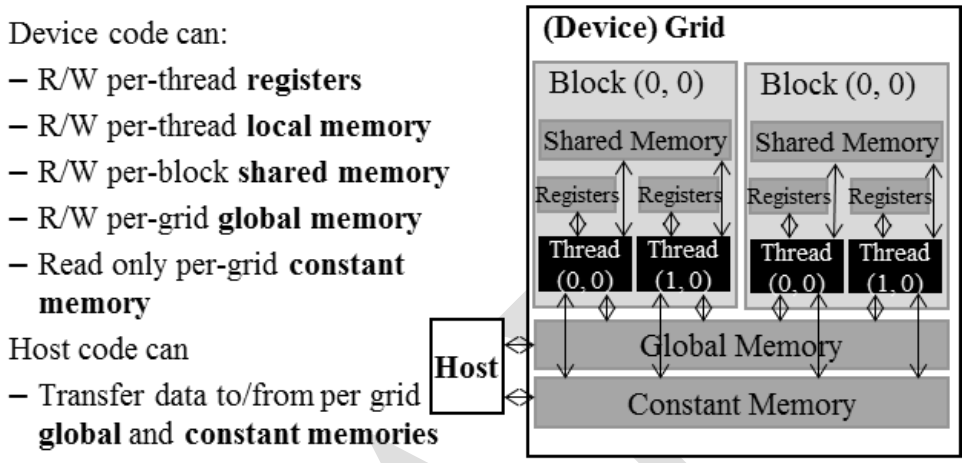


Figure 4.6: *Overview of the CUDA device memory model*

In order to fully appreciate the difference between registers, shared memory and global memory, we need to go into a little more details of how these different memory types are realized and used in modern processors. Virtually all modern processors find their root in the model proposed by John von Neumann in 1945, which is shown in Figure 4.7. The CUDA devices are no exception. The Global Memory in a CUDA device maps to the Memory box in Figure 4.7. The processor box corresponds to the processor chip boundary that we typically see today. The Global Memory is off the processor chip and is implemented with DRAM technology, which implies long access latencies and relatively low access bandwidth. The Registers correspond to the "Register File" of the von Neumann model. The Register File is on the processor chip, which implies very short access latency and drastically higher access bandwidth when compared to the global memory. In a typical device, the aggregated access bandwidth of the register files is at least two orders of magnitude higher than that of the global memory. Furthermore, whenever a variable is stored in a register, its accesses no longer consume off-chip global memory bandwidth. This will be reflected as an increased compute-to-global-memory-access ratio.

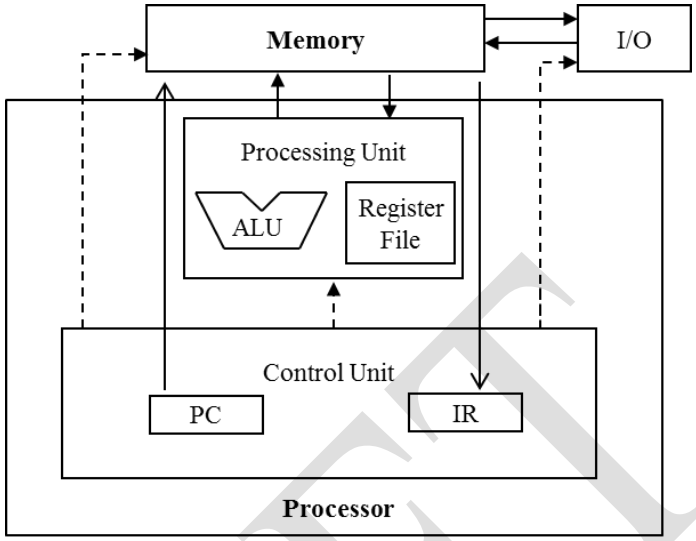


***Figure 4.7**: Memory vs. Registers in a modern computer based on the von Neumann model*

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

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

where r2 and r3 are the register numbers that specifies the location in the register file where the input operand values can be found. The location for storing the floating-point addition result value is specified by r1. Therefore, when an operand of an arithmetic instruction is in

The von Neumann Model

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

The von Neumann Model is illustrated in the Figure 4.7. The computer has an I/O (input/output) that allows both programs and data to be provided to and generated from the system. In order to execute a program, the computer first inputs the program and its data into the Memory.

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

a register, there is no additional instruction required to make the operand value available to the arithmetic and logic unit (ALU), where the arithmetic calculation is done.

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

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

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

Finally, there is another subtle reason why placing an operand value in registers is preferable. In modern computers, the energy consumed for accessing a value from the register file is at least an order of magnitude lower that for accessing a value from the global memory. We will look at more details of the speed and energy difference in accessing these two hardware structures in modern computers soon. However, as we will soon learn, the number of registers available to each thread (see "Processing Units and Threads" sidebar) is quite limited in today's GPUs. We need to be careful not oversubscribe to this limited resource.

Figure 4.8 shows the shared memory and registers in a CUDA device. Although both are on-chip memories, they differ significantly in functionality and cost of access. Shared memory is designed as part of the memory space that resides on the processor chip. When the processor accesses data that reside in the shared memory, it needs to perform a memory load operation, just like accessing data in the global memory. However, because shared memory resides on-chip, it can be accessed with much lower latency and much higher throughput than the global memory. Because of the need to perform a load operation, share memory has longer latency and lower bandwidth than registers. In computer architecture terminology, the shared memory is a form of *scratchpad memory*.

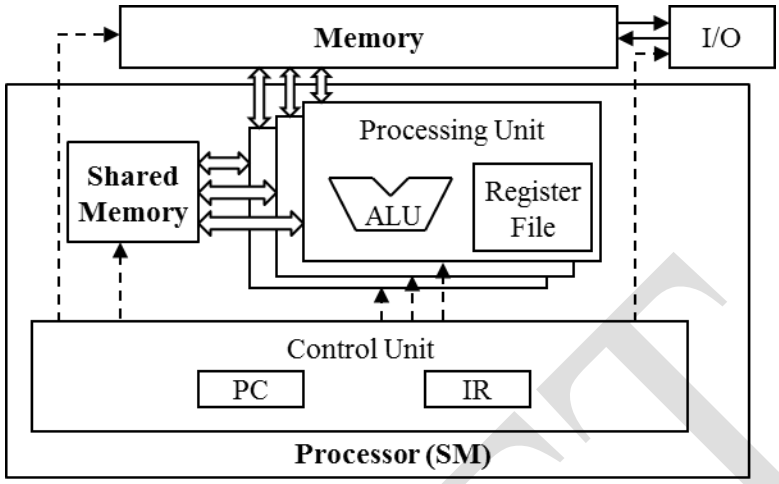


Figure 4.8: *Shared memory vs. registers in a CUDA device SM*

One important difference between the share memory and registers in CUDA is that variables that reside in the shared memory are accessible by all threads in a block. This is in contrast to register data, which is private to a thread. That is, shared memory is designed to support

Processing Units and Threads

Now that we have introduced the von Neumann model, we are ready to discuss how threads are implemented. A thread in modern computers is the state of executing a program on a von Neumann Processor. Recall that a thread consists of the code of a program, the particular point in the code that is being executed, and value of its variables and data structures.

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

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

Some processors provide multiple processing units, which allow multiple threads to make simultaneous progress. Figure 4.8 shows a Single-Instruction, Multiple-Data (SIMD) design style where multiple processing units share a PC and IR. Under this design, all threads make simultaneous progress by executing the same instruction in the program.

efficient, high-bandwidth sharing of data among threads in a block. As shown in Figure 4.8, a CUDA device SM typically employs multiple processing units, to allow multiple threads to make simultaneous progress (see "Processing Units and Threads" sidebar). Threads in a block can be spread across these processing units. Therefore, the hardware implementations of the shared memory in these CUDA devices are typically designed to allow multiple processing units to simultaneously access its contents to support efficient data sharing among threads in a block. We will be learning several important types of parallel algorithms that can greatly benefit from such efficient data sharing among threads.

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

Table 4.1: *CUDA Variable Type Qualifiers*

It should be clear by now that registers, shared memory and global memory all have different functionalities, latencies, and bandwidth. It is therefore important to understand how to declare a variable so that it will reside in the intended type of memory. Table 4.1 presents the CUDA syntax for declaring program variables into the various memory types. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: a single thread only, all threads of a block, or all threads of all grids. If a variable's scope is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable. For example, if a kernel declares a variable whose scope is a thread and it is launched with one million threads, one million versions of the variable will be created so that each thread initializes and uses its own version of the variable.

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

We refer to variables that are not arrays or matrices as *scalar* variables. As shown in Table 4.1, all automatic scalar variables declared in kernel and device functions are placed into registers. The scopes of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all its automatic variables also cease to exist. In Figure 4.1, variables blurRow, blurCol, curRow, curCol, pixels and pixVal are all automatic variables and fall into this category. Note that accessing these variables is extremely fast and parallel but one must be careful not to exceed the limited capacity of the register storage in the hardware implementations. Using a large number of registers can negatively impact the number of active threads assigned to each SM. We will address this point in Chapter 5.

Automatic array variables are not stored in registers[2]. Instead, they are stored into the global memory and may incur long access delays and potential access congestions. The scope of these arrays is, like automatic scalar variables, limited to individual threads. That is, a private version of each automatic array is created for and used by every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist. From our experience, one seldom needs to use automatic array variables in kernel functions and device functions.

If a variable declaration is preceded by the "__shared__" (each "__" consists of two "_" characters) keyword, it declares a shared variable in CUDA. One can also add an optional "__device__" in front of "__shared__" in the declaration to achieve the same effect. Such declaration typically resides within a kernel function or a device function. Shared variables reside in shared memory. The scope of a shared variable is within a thread block, that is, all threads in a block see the same version of a shared variable. A private version of the shared variable is created for and used by each thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel. When a kernel terminates its execution, the contents of its shared variables cease to exist. As we discussed earlier, shared variables are an efficient means for threads within a block to collaborate with each other. Accessing shared variables from the shared memory is extremely fast and highly parallel. CUDA programmers often use shared variables to hold the portion of global memory data that are heavily used in an execution phase of kernel. One may need to adjust the algorithms used in order to create execution phases that heavily focus on small portions of the global memory data, as we will demonstrate with matrix multiplication in Section 4.4.

If a variable declaration is preceded by keywords "__constant__" (each "__" consists of two "_" characters) it declares a constant variable in CUDA. One can also add an optional

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

"__device__" in front of "__constant__" to achieve the same effect. Declaration of constant variables must be outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution. Constant variables are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at 65,536 bytes. One may need to break up the input data volume to fit within this limitation, as we will illustrate in Chapter 7.

A variable whose declaration is preceded only by the keyword "__device__" (each "__'' consists of two "_'' characters), is a global variable and will be placed in the global memory. Accesses to a global variable are slow. Latency and throughput of accessing global variables have been improved with caches in more recent devices. One important advantage of global variables is that they are visible to all threads of all kernels. Their contents also persist through the entire execution. Thus, global variables can be used as a means for threads to collaborate across blocks. One must, however, be aware of the fact that there is currently no easy way to synchronize between threads from different thread blocks or to ensure data consistency across threads when accessing global memory other than terminating the current kernel execution[3]. Therefore, global variables are often used to pass information from one kernel invocation to another kernel invocation.

In CUDA, pointers are used to point to data objects in the global memory. There are two typical ways in which pointer usage arises in kernel and device functions. First, if an object is allocated by a host function, the pointer to the object is initialized by cudaMalloc and can be passed to the kernel function as a parameter. For example, the parameters M, N, and P in Figure 4.1 are such pointers. The second type of usage is to assign the address of a variable declared in the global memory to a pointer variable. For example, the statement {float* ptr = &GlobalVar;} in a kernel function assigns the address of GlobalVar into an automatic pointer variable ptr. The reader should refer to the CUDA Programming Guide for using pointers in other memory types.

4.4. Tiling for Reduced Memory Traffic

We have an intrinsic tradeoff in the use of device memories in CUDA: the global memory is large but slow whereas the shared memory is small but fast. A common strategy is to partition the data into subsets called *tiles* so that each tile fits into the shared memory. The term tile draws on the analogy that a large wall (i.e., the global memory data) can be covered

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

by tiles (i.e., subsets that each can fit into the shared memory). An important criterion is that the kernel computation on these tiles can be done independently of each other. Note that not all data structures can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example. Figure 4.5 shows a small example of matrix multiplication. It corresponds to the kernel function in Figure 4.3. We replicate the example in Figure 4.9 for convenient reference by the reader. For brevity, we abbreviate P[y*Width+x], M[y*Width+x], and N[y*Width+x] into $P_{y,x}$, $M_{y,x}$, and $N_{y,x}$. This example assumes that we use four 2×2 blocks to compute the P matrix. Figure 4.5 highlights the computation done by the four threads of block(0,0). These four threads compute $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, and $P_{1,1}$. The accesses to the M and N elements by thread(0,0) and thread(0,1) of block(0,0) are highlighted with black arrows. For example, thread(0,0) reads $M_{0,0}$ and $N_{0,0}$, followed by $M_{0,1}$ and $N_{1,0}$ followed by $M_{0,2}$ and $N_{2,0}$, followed by $M_{0,3}$ and $N_{3,0}$.

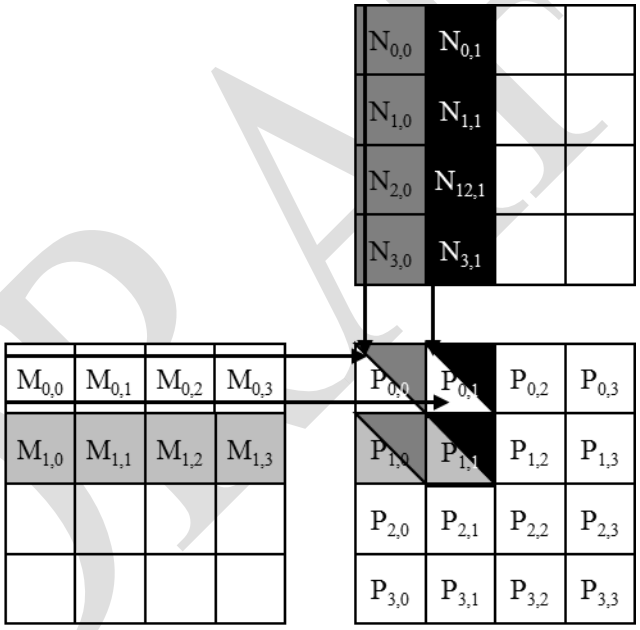


Figure 4.9 *A small example of matrix multiplication. For brevity, We show M[y*Width+x], N[y*Width+x], P[y*Width+x] as $M_{y,x}$, $N_{y,x}$ $P_{y,x}$.*

Figure 4.10 shows the global memory accesses done by all threads in $block_{0,0}$. The threads are listed in the vertical direction, with time of access increasing to the right in the horizontal direction. Note that each thread accesses four elements of M and four elements of N during its execution. Among the four threads highlighted, there is a significant overlap in the M and

N elements they access. For example, $thread_{0,0}$ and $thread_{0,1}$ both access $M_{0,0}$ as well as the rest of row 0 of M. Similarly, $thread_{0,1}$ and $thread_{1,1}$ both access $N_{0,1}$ as well as the rest of column 1 of N.

The kernel in Figure 4.3 is written so that both $thread_{0,0}$ and $thread_{0,1}$ access row 0 elements of M from the global memory. If we can somehow manage to have $thread_{0,0}$ and $thread_{1,0}$ to collaborate so that these M elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In fact, we can see that every M and N element is accessed exactly twice during the execution of $block_{0,0}$. Therefore, if we can have all four threads to collaborate in their accesses to global memory, we can reduce the traffic to the global memory by half.

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

Access order

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

Figure 4.10*: Global memory accesses performed by threads in* $block_{0,0}$

Traffic congestion obviously does not only arise in computing. Most of us have experienced traffic congestion in highway systems, as illustrated in Figure 4.11. The root cause of highway traffic congestion is that there are too many cars all squeezing through a road that is designed for a much smaller number of vehicles. When congestion occurs, the travel time for each vehicle is greatly increased. Commute time to work can easily double or triple during traffic congestion.

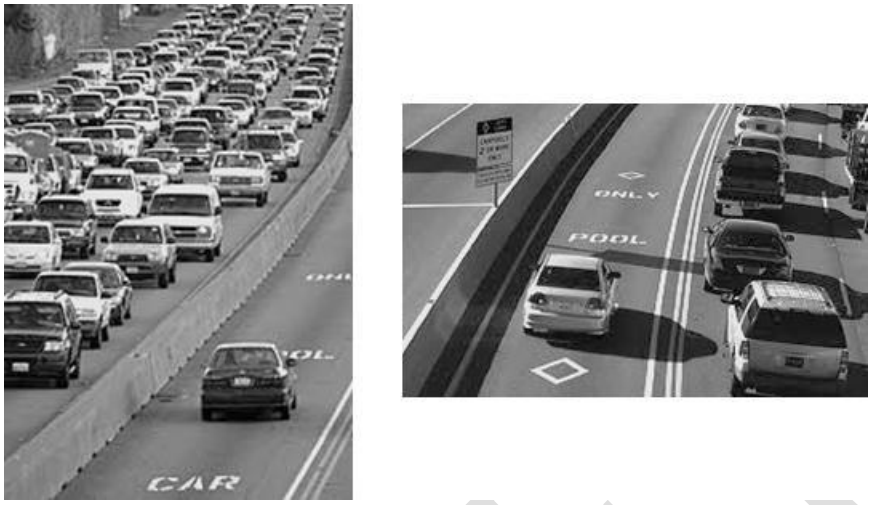


Figure 4.11*: Reducing traffic congestion in highway systems*

Most solutions for reduced traffic congestion involve reduction of cars on the road. Assuming that the number of commuters is constant, people need to share rides in order to reduce the number of cars on the road. A common way to share rides in the U.S. is carpooling, where a group of commuters take turns to drive the group to work in one vehicle. The government usually need to have policies to encourages carpooling. In some countries, the government simply disallows certain classes of cars to be on the road on a daily basis. For example, cars with odd license plates may not be allowed on the road on Monday, Wednesday, or Friday. This encourages people whose cars are allowed on different days to form a carpool group. There are also countries where the government makes gasoline so expensive that people form carpools to save money. In other countries, the government may provide incentives for behavior that reduces the number of cars on the road. In the U.S., some lanes of congested highways are designated as carpool lanes; only cars with more than 2 or 3 people are allowed to use these lanes. All these measures for encouraging carpooling are designed to overcome the fact that carpooling requires extra effort, as we show in Figure 4.12.

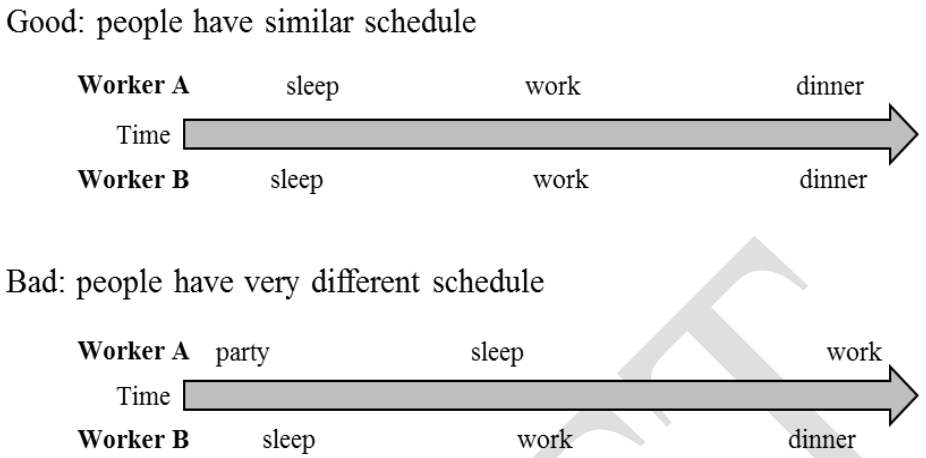


Figure 4.12: *Carpooling requires synchronization among people*

Carpooling requires workers who wish to carpool to compromise and agree on a common commute schedule. The top half of Figure 4.12 shows a good schedule pattern for carpooling. Time goes from left to right. Worker A and Worker B have similar schedule for sleep, work, and dinner. This allows these two workers to easily go to work and return home in one car. Their similar schedules allow them to more easily agree on a common departure time and return time. This is however, not the case of the schedules shown in the bottom half of Figure 4.12. Worker A and Worker B have very different habit in this case. Worker A parties until sunrise, sleeps during the day, and goes to work in the evening. Worker B sleeps at night, goes to work in the morning, and returns home for dinner at 6pm. The schedules are so wildly different that there is no way these two workers can coordinate a common time to drive to work and return home in one car. In order for these workers to form a carpool, they need to negotiate a common schedule similar what is shown in the top half of Figure 4.12.

Tiled algorithms are very similar to carpooling arrangements. We can think of threads accessing data values as commuters and DRAM access requests as vehicles. When the rate of DRAM requests exceeds the provisioned access bandwidth of the DRAM system, traffic congestion arises and the arithmetic units become idle. If multiple threads access data from the same DRAM location, they can potentially form a "carpool" and combine their accesses into one DRAM request. This, however, requires the threads to have similar execution schedule so that their data accesses can be combined into one. This is shown in Figure 4.13, where the cells in the middle of the picture depicts DRAM locations. An arrow going from a DRAM location to a thread represent an access by the thread to that location at the time marked by the head of the arrow. Note that the time goes from left to right. The top portion shows two threads that access the same data elements with similar timing. The bottom half shows two threads that access their common data in very different times. That is, the accesses

by Thread 2 lag significantly behind their corresponding accesses by Thread 1. The reason why the bottom is a bad arrangement is that data elements brought back from the DRAM need to be kept in the on-chip memory for a long time, waiting for Thread 2 to consume them. This will likely require a large number of data elements to be kept around, resulting in excessive on-chip memory requirements.

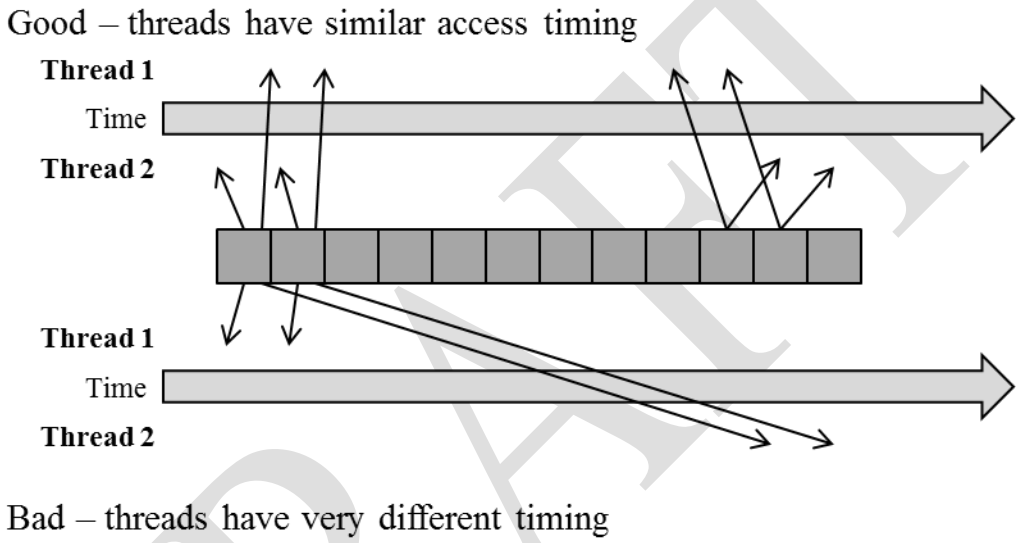


Figure 4.13 *Tiled Algorithms require synchronization among threads*

In the context of parallel computing, tiling is a program transformation technique that localizes the memory locations accessed among threads and the timing of their accesses. It breaks long access sequences of each thread into phases and use barrier synchronization to keep the timing of accesses to each section by the threads close to each other. It controls the amount of on-chip memory required by localizing the accesses both in time and in space. In terms of our carpool analogy, we keep the threads that form the "carpool" group to follow approximately the same execution timing.

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

is chosen so that they can fit into the shared memory. In the simplest form, the tile dimensions equal those of the block, as illustrated in Figure 4.11.

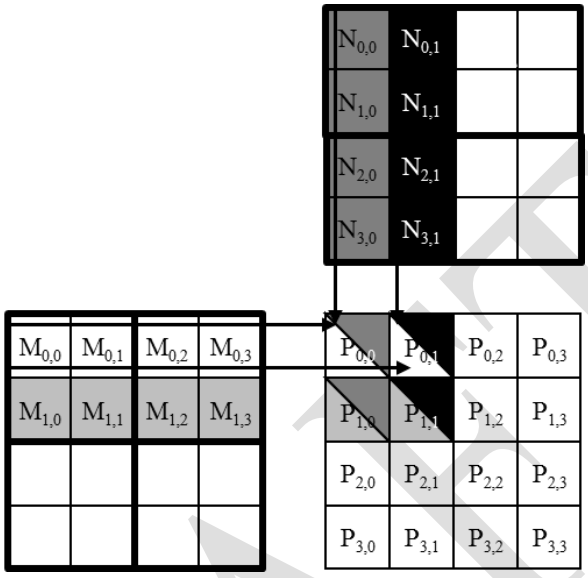


Figure 4.14 *Tiling M and N to utilize shared memory*

In Figure 4.14, we divide M and N into 2×2 tiles, as delineated by the thick lines. The dot product calculations performed by each thread are now divided into phases. In each phase, all threads in a block collaborate to load a tile of M and a tile of N into the shared memory. This can be done by having every thread in a block to load one M element and one N element into the shared memory, as illustrated in Figure 4.15. Each row of Figure 4.15 shows the execution activities of a thread. Note that time progresses from left to right. We only need to show the activities of threads in block$_{0,0}$; the other blocks all have the same behavior. The shared memory array for the M elements is called Mds. The shared memory array for the N elements is called Nds. At the beginning of Phase 1, the four threads of block$_{0,0}$ collaboratively load a tile of M into shared memory: thread$_{0,0}$ loads M$_{0,0}$ into Mds$_{0,0}$, thread$_{0,1}$ loads M$_{0,1}$ into Mds$_{0,1}$, thread$_{1,0}$ loads M$_{1,0}$ into Mds$_{1,0}$, and thread$_{1,1}$ loads M$_{1,1}$ into Mds$_{1,1}$. These loads are shown in the second column of Figure 4.15. A tile of N is also loaded in a similar manner, shown in the third column of Figure 4.15.

After the two tiles of M and N are loaded into the shared memory, these elements are used in the calculation of the dot product. Note that each value in the shared memory is used twice. For example, the M$_{1,1}$ value, loaded by thread$_{1,1}$ into Mds$_{1,1}$, is used twice, once by thread$_{1,0}$ and once by thread$_{1,1}$. By loading each global memory value into shared memory

so that it can be used multiple times, we reduce the number of accesses to the global memory. In this case, we reduce the number of accesses to the global memory by half. The reader should verify that the reduction is by a factor of N if the tiles are N×N elements.

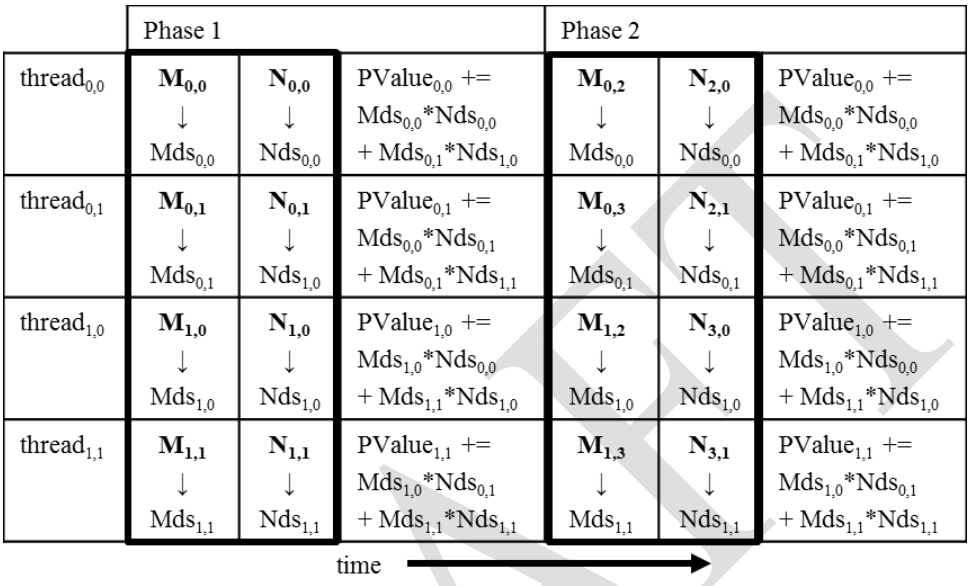


Figure 4.15: *Execution phases of a tiled matrix multiplication*

Note that the calculation of each dot product in Figure 4.3 is now performed in two phases, shown as Phase 1 and Phase 2 in Figure 4.15. In each phase, products of two pairs of the input matrix elements are accumulated into the Pvalue variable. Note that Pvalue is an automatic variable so a private version is generated for each thread. We added subscripts just to clarify that these are different instances of the Pvalue variable created for each thread. The first phase calculation is shown in the fourth column of Figure 4.15; the second phase the seventh column. In general, if an input matrix is of dimension Width and the tile size is TILE_WIDTH, the dot product would be performed in Width/TILE_WIDTH phases. The creation of these phases is key to the reduction of accesses to the global memory. With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy their overlapping input needs in the phase.

Note also that Mds and Nds are re-used to hold the input values. In each phase, the same Mds and Nds are used to hold the subset of M and N elements used in the phase. This allows a much smaller shared memory to serve most of the accesses to global memory. This is due to the fact that each phase focuses on a small subset of the input matrix elements. Such focused access behavior is called locality. When an algorithm exhibits locality, there is an

opportunity to use small, high-speed memories to serve most of the accesses and remove these accesses from the global memory. Locality is as important for achieving high-performance in multi-core CPUs as in many-thread GPUs We will return to the concept of locality in Chapter 5.

4.5. A Tiled Matrix Multiplication Kernel

We are now ready to present a tiled matrix multiplication kernel that uses shared memory to reduce traffic to the global memory. The kernel shown in Figure 4.16 implements the phases illustrated in Figure 4.15. In Figure 4.16, Line 1 and Line 2 declare Mds and Nds as shared memory variables. Recall that the scope of shared memory variables is a block. Thus, one pair of Mds and Nds will be created for each block and all threads of a block have access to the same Mds and Nds. This is important since all threads in a block must have access to the M and N elements loaded into Mds and Nds by their peers so that they can use these values to satisfy their input needs.

```
#define TILE_WIDTH 16

    __global__ void MatrixMulKernel(float* M, float* N, float* P,
      int Width) {

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

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

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

 7.   float Pvalue = 0;
      // Loop over the M and N tiles required to compute P element
 8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

        // Collaborative loading of M and N tiles into shared memory
 9.     Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

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

```
    }
```

Figure 4.16*: A tiled Matrix Multiplication Kernel using shared memory*

Lines 3 and 4 save the threadIdx and blockIdx values into automatic variables and thus into registers for fast access. Recall that automatic scalar variables are placed into registers. Their scope is in each individual thread. That is, one private version of tx, ty, bx, and by is created by the run-time system for each thread and will reside in registers that are accessible by the thread. They are initialized with the threadIdx and blockIdx values and used many times during the lifetime of thread. Once the thread ends, the values of these variables cease to exist.

Lines 5 and 6 determine the row index and column index of the P element that the thread is to produce. The code assumes that each thread is responsible for calculating one P element. As shown in Line 6, the horizontal (x) position, or the column index of the P element to be produced by a thread can be calculated as bx*TILE_WIDTH+tx. This is because each block covers TILE_WIDTH elements in the horizontal dimension. A thread in block bx would have bx blocks of threads, or (bx*TILE_WIDTH) threads, before it; they cover bx*TILE_WIDTH elements of P. Another tx threads within the same block would cover another tx elements. Thus the thread with bx and tx should be responsible for calculating the P element whose x index is bx*TILE_WIDTH+tx. This horizontal index is saved in the variable Col for the thread and is also illustrated in Figure 4.17.

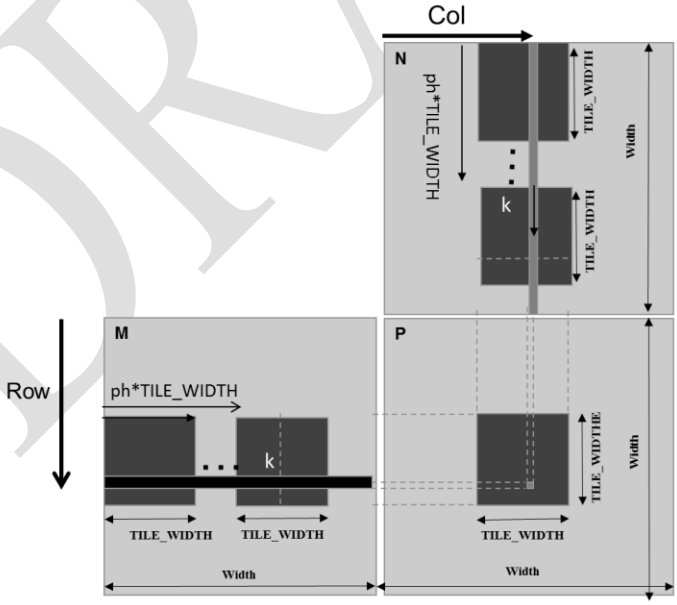


Figure 4.17*: Calculation of the matrix indices in tiled multiplication*

For the example in Figure 4.14, the x index of the P element to be calculated by $thread_{0,1}$ of $block_{1,0}$ is 0*2+1 = 1. Similarly, the y index can be calculated as by*TILE_WIDTH+ty. This vertical index is saved in the variable Row for the thread. Thus, as shown in Figure 4.17, each thread calculates the P element at the Col^{th} column and the Row^{th} row. Going back to the example in Figure 4.14, the y index of the P element to be calculated by $thread_{1,0}$ of $block_{0,1}$ is 1*2+0 = 2. Thus, the P element to be calculated by this thread is $P_{2,1}$.

Line 8 of Figure 4.16 marks the beginning of the loop that iterates through all the phases of calculating the P element. Each iteration of the loop corresponds to one phase of the calculation shown in Figure 4.15. The ph variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of M and one tile of N elements. Therefore, at the beginning of each phase, ph*TILE_WIDTH pairs of M and N elements have been processed by previous phases.

In each phase, Line 9 loads the appropriate M element into the shared memory. Since we already know the row of M and column of N to be processed by the thread, we now turn our focus to the column index of M and row index of N. As shown in Figure 4.17, each block has $TILE\_WIDTH^2$ threads that will collaborate to load $TILE\_WIDTH^2$ M elements into the shared memory. Thus, all we need to do is to assign each thread to load one M element. This is conveniently done using the blockIdx and threadIdx. Note that the beginning column index of the section of M elements to be loaded is ph*TILE_WIDTH. Therefore, an easy approach is to have every thread load an element that is tx (the threadIdx.x value) positions away from that beginning point.

This is precisely what we have in Line 9, where each thread loads M[Row*Width + ph*TILE_WIDTH + tx], where the linearized index is formed with the row index Row and column index ph*TILE_WIDTH + tx. Since the value of Row is a linear function of ty, each of the $TILE\_WIDTH^2$ threads will load a unique M element into the shared memory. Together, these threads will load a dark square subset of M in Figure 4.17. The reader should use the small example in Figure 4.14 and Figure 4.15 to verify that the address calculation works correctly for individual threads.

The barrier __syncthreads() in Line 11 ensures that all threads have finished loading the tiles of M and N into Mds and Nds before any of them can move forward. The loop in Line 12 then performs one phase of the dot product based on these tile elements. The progression of the loop for $thread_{ty,tx}$ is shown in Figure 4.17, with the access direction of the M and N elements along the arrow marked with k, the loop variable in Line 12. Note that these elements will be accessed from Mds and Nds, the shared memory arrays holding these M and N elements. The barrier __syncthreads() in Line 14 ensures that all threads have finished

using the M and N elements in the shared memory before any of them move on to the next iteration and load the elements from the next tiles. This way, none of the threads would load the elements too early and corrupt the input values for other threads.

The nested loop from Line 8 to Line 14 illustrates a technique called *strip-mining*, which takes a long-running loop and break it into phases. Each phase consists of an inner loop that executes a number of consecutive iterations of the original loop. The original loop becomes an outer loop whose role is to iteratively invoke the inner loop so that all the iterations of the original loop are executed in their original order. By adding barrier synchronizations before and after the inner loop, we force all threads in the same block to all focus their work on a section of their input data. Strip-mining is an important means to creating the phases needed by tiling in data parallel programs. [4]

After all phases of the dot product are complete, the execution exits the loop of Line 8. All threads write to their P element using the linearized index calculated from Row and Col.

The benefit of the tiled algorithm is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of TILE_WIDTH. If one uses 16×16 tiles, we can reduce the global memory accesses by a factor of 16. This increases the compute-to-global-memory-access ratio from 1 to 16. This improvement allows the memory bandwidth of a CUDA device to support a computation rate close to its peak performance. For example, this improvement allows a device with 150 GB/s global memory bandwidth to approach ((150/4)*16) = 600 GFLOPS!

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

4.6. Boundary Checks

We now extend the tiled matrix multiplication kernel to handle matrices with arbitrary width. The extensions will have to allow the kernel to correctly handle matrices whose width is not

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

a multiple of the tile width. Let's change the small example in Figure 4.14 to 3x3 M, N, and P matrices. The revised example is shown in Figure 4.18. Note that the width of the matrices is 3, which is not a multiple of the tile width (2). Figure 4.18 shows the memory access pattern during phase 1 of block$_{0,0}$. We see that thread$_{0,1}$ and thread$_{1,1}$ will attempt to load M elements that do not exist. Similarly, we see that thread$_{1,0}$ and thread$_{1,1}$ will attempt to access N elements that do not exist.

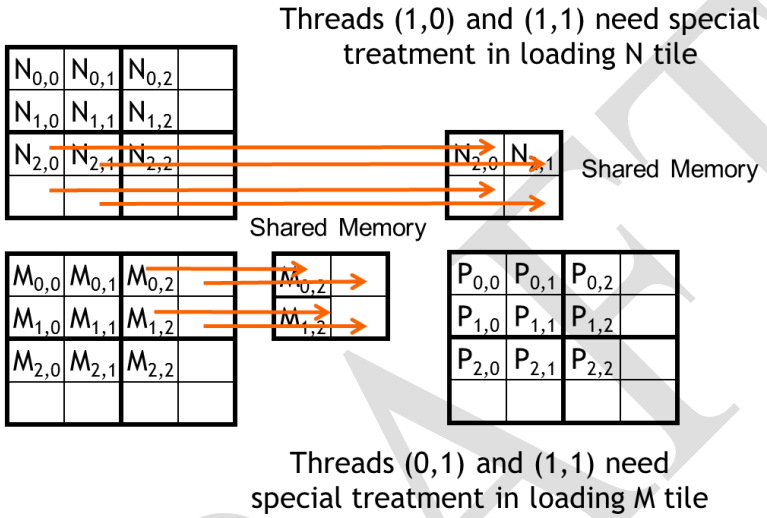


Figure 4.18: *Loading input matrix elements that are close to the edge – phase 1 of Block$_{0,0}$*

Accessing non-existing elements is problematic in two ways. In the case of accessing a non-existing elements past the end of a row (M accesses by thread$_{1,0}$ and thread$_{1,1}$ in Figure 4.18), these accesses will be done to incorrect elements. In our example, the threads will attempt to access M$_{0,3}$ and M$_{1,3}$, which do not exist. So, what will happen to these memory loads? In order to answer this question, we need to go back to the linearized layout of 2D matrices. The element after M$_{0,2}$ in the linearized layout is M$_{1,0}$. Although thead0,1 is attempting to access M0,3, it will end up getting M$_{1,0}$. The use of this value in the subsequent inner product calculation will obviously corrupt the output value.

A similar problem arises when accessing an element past the end of a column (N accesses by thread$_{1,0}$ and thread$_{1,1}$ in Figure 4.18). These accesses are to memory locations outside the allocated area for the array. In some systems, they will return random values from other data structures. Other systems will reject these accesses and cause the program to abort. Either way, the outcome of such accesses is undesirable.

From our discussion so far, it may seem that the problematic accesses only arise in the last phase of execution of the threads. This would suggest that we can deal with it by taking special actions during the last phase of the tiled kernel execution. Unfortunately this is not

true. Problematic accesses can arise in all phases. Figure 4.19 shows the memory access pattern of $block_{1,1}$ during phase 0. We see that $thread_{1,0}$ and $thread_{1,1}$ attempt to access non-existing M elements $M_{3,0}$ and $M_{3,1}$ whereas thread0,1and thread1,1 attempt to access $N_{0,3}$ and $N_{1,3}$, which do not exist.

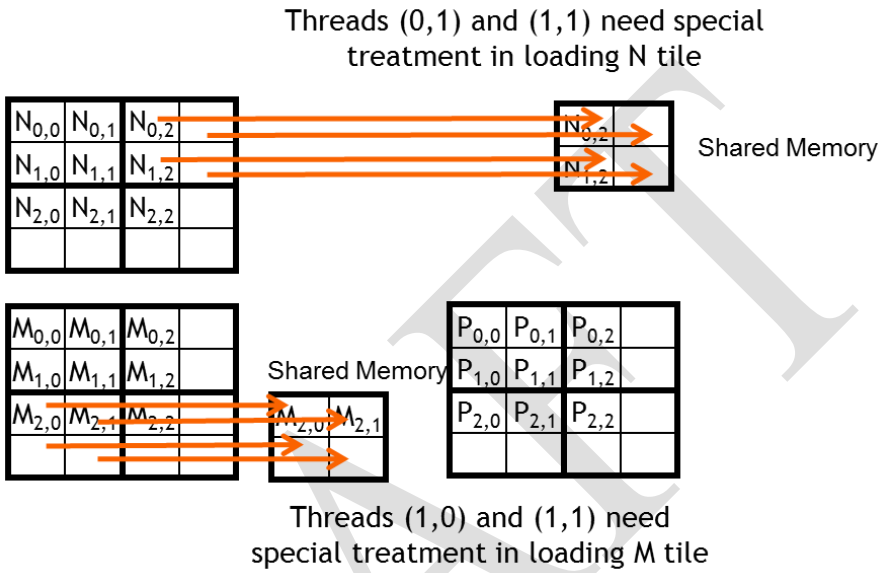


Figure 4.19: *Loading input elements during phase 0 of block1,0.*

Note that these problematic accesses cannot be prevented by excluding the threads that do not calculate valid P elements. For example, $thread_{1,0}$ in $block_{1,1}$ does not calculate any valid P element. However, it needs to load $M_{2,1}$ during phase 0. Further note that some threads that calculate valid P elements will attempt to access M or N elements that do not exist. For example, as we have seen in Figure 4.18, $thread_{0,1}$ of block 0,0 calculates a valid P element $P_{0,1}$. However, it attempts to access a non-existing $M_{0,3}$ during phase 1. These two facts indicate that we will need to use different boundary condition tests for loading M tiles, loading N tiles, and calculating/storing P elements.

Let's start with the boundary test condition for loading input tiles. When a thread is to load an input tile element, it should test whether the input element it is attempting to load is a valid element. This is easily done by examining the y and x indices. For example, at Line 9 in Figure 4.16, the linearized index is a derived from a y index of Row and an x index of ph*TILE_WIDTH + tx. The boundary condition test would be that both of indices are smaller than Width: (Row<Width) && (ph*TILE_WIDTH+tx)<Width. If the condition is true, the thread should go ahead and load the M element. The reader should verify that the condition test for loading the N element is (ph*TILE_WIDTH+ty)<Width && Col<Width.

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

Finally, a thread should only store its final inner product value if it is responsible for calculating a valid P element. The test for this condition is (Row<Width) && (Col<Width). The kernel code with the additional boundary condition checks is shown in Figure 4.20.

```
// Loop over the M and N tiles required to compute P element
8.   for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ++ph) {

        // Collaborative loading of M and N tiles into shared memory
9.      if ((Row< Width) && (ph*TILE_WIDTH+tx)< Width)
          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     if ((ph*TILE_WIDTH+ty)<Width && Col<Width)
                Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

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

Figure 4.20*: Tiled matrix multiplication kernel with boundary condition checks.*

With the boundary condition checks, the tile matrix multiplication kernel is just one more step away from being a general matrix multiplication kernel. In general, matrix multiplication is defined for rectangular matrices: a jxk M matrix multiplied with a kxl N matrix results in a jxl P matrix. Our kernel can only handle square matrices so far.

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

4.7. Memory as a Limiting Factor to Parallelism

While CUDA registers and shared memory can be extremely effective in reducing the number of accesses to global memory, one must be careful to stay within the capacity of these memories. These memories are forms of resources that are needed for thread execution. Each CUDA device offers a limited amount of resources, which limits the number threads that can simultaneously reside in the SM for a given application. In general, the more resources each thread requires, the fewer the number of threads can reside in each SM, and thus the fewer number of threads that can run in parallel in the entire device.

Let's use an example to illustrate the interaction between register usage of a kernel and the level of parallelism that a device can support. Assume that in a current generation device D, each SM can accommodate up to 1,536 threads and has 16,384 registers. While 16,384 is a large number, it only allows each thread to use a very limited number of registers considering the number of threads that can reside in each SM. In order to support 1,536 threads, each thread can use only 16,384/1,536= 10 registers. If each thread uses 11 registers, the number of threads that are able to be executed concurrently in each SM will be reduced. Such reduction is done at the block granularity. For example, if each block contains 512 threads, the reduction of threads will be done by reducing 512 threads at a time. Thus, the next lower number of threads from 1,536 would be 1,024, a 1/3 reduction of threads that can simultaneously reside in each SM. This can greatly reduce the number of warps available for scheduling, thus reducing the processor's ability to find useful work in the presence of long-latency operations.

Note that the number of registers available to each SM varies from device to device. An application can dynamically determine the number of registers available in each SM of the device used and choose a version of the kernel that uses the number of registers appropriate for the device. This can be done by calling the `cudaGetDeviceProperties` function, whose use was discussed in Section 3.6. Assume that variable `&dev_prop` is passed to the function for the device property, the field `dev_prop.regsPerBlock` gives the number of registers available in each SM. For current generation devices, the returned value for this field should be 16,384. The application can then divide this number by the target number of threads to reside in each SM to determine the number of registers that can be used in the kernel.

Shared memory usage can also limit the number of threads assigned to each SM. Assume device D mentioned above has 16,384 (16K) bytes of shared memory in each SM. Keep in mind that shared memory is allocated to thread blocks. Assume that each SM in D can accommodate up to 8 blocks. In order to reach this maximum, each block must not use more than 2K bytes of shared memory. If each block uses more than 2K bytes of memory, the number of blocks that can reside in each SM is reduced such that the total amount of shared memory used by these blocks does not exceed 16K bytes. For example, if each block uses 5K bytes of shared memory, no more than three blocks can be assigned to each SM.

For the matrix multiplication example, shared memory can become a limiting factor. For a tile size of 16 × 16, each block needs 16×16×4 = 1K bytes of storage for Mds. (Keep in mind that each element is a float type, which is 4 bytes.) Another 1KB is needed for Nds. Thus each block uses 2K bytes of shared memory. The 16K-byte shared memory allows 8 blocks to simultaneous reside in an SM. Since this is the same as the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size. In this case, the real limitation is the threading hardware limitation that only 1,536 threads are allowed in each SM. This limits the number of blocks in each SM to six. As a result, only 6*2KB=12KB of the shared memory will be used. These limits do change from device to device but can be determined at runtime with device queries.

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

Unfortunately, the kernel in Figure 4.16 does not support this. The declarations used in Figure 4.16 hardwire the size of its shared memory usage to a compile-time constant:

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

That is, the size of Mds and Nds is set to be $TILE\_WIDTH^2$ elements, whatever the value of TILE_WIDTH is set to be at compile time. For example, assume that the file contains

```
#define TILE_WIDTH 16
```

Both Mds and Nds will have 256 elements. If we want to change the size of Mds and Nds, we need to change the value of TILE_WIDTH and recompile the code. The kernel cannot easily adjust its shared memory usage at runtime without recompilation.

We can enable such adjustment with a different style of declaration in CUDA. We can add a C “extern” keyword in front of the shared memory declaration and omit the size of the array in the declaration. Based on this style, the declarations for Mds and Nds become:

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

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

At run time, when we launch the kernel, we can dynamically determine the amount of shared memory to be used according to the device query result and supply that as a **third** configuration parameter to the kernel launch. For example, the revised kernel could be launched with the following statements:

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

matrixMulKernel<<<dimGrid, dimBlock, size>>>(Md, Nd, Pd, Width);
```

where size_t is a built-in type for declaring a variable to holds the size information for dynamically allocated data structures. The size is expressed in number of bytes. In our matrix multiplication example, for a 16x16 tile, we have size to be 16x16x4=1024 bytes. We have omitted the details of the calculation for setting the value of size at run time.

4.8. Summary

In summary, the execution speed of a program in modern processors can be severely limited by the speed of the memory. To achieve good utilization of the execution throughput of CUDA devices, one needs to achieve a high compute-to-global-memory-access ratio in the kernel code. If the ratio is low, the kernel is memory-bound. That is, its execution speed is limited by the rate at which its operands are accessed from memory.

CUDA defines registers, shared memory, and constant memory. These memories are much smaller than the global memory but can be accessed at much higher speed. Using these memories effectively requires re-design of the algorithm. We use matrix multiplication as an example to illustrate tiling, a popular strategy to enhance locality of data access and enable effective use of shared memory. In parallel programming, tiling forces multiple threads to jointly focus on a subset of the input data at each phase of the execution so that the subset data can be placed into these special memory types to enable much higher access speed. We demonstrate that with 16×16 tiling, global memory accesses are no longer the major limiting factor for matrix multiplication performance.

It is, however, important for CUDA programmers to be aware of the limited sizes of these types of memory. Their capacities are implementation dependent. Once their capacities are exceeded, they limit the number of threads that can be simultaneously executing in each SM.

The ability to reason about hardware limitations when developing an application is a key aspect of computational thinking.

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

Our goal for this chapter is to introduce the concept of locality, tiling, and different CUDA memory types. We introduced a tiled matrix multiplication kernel using shared memory. We have not discussed the use of registers and constant memory in tiling. We will explain the use of these memory types in tiled algorithms when we discussed parallel algorithm patterns.

4.9. Exercises

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

2. Draw the equivalent of Figure 4.14 for a 8×8 matrix multiplication with 2×2 tiling and 4×4 tiling. Verify that the reduction in global memory bandwidth is indeed proportional to the dimension size of the tiles.

3. What type of incorrect execution behavior can happen if one forgot to use one or both __syncthreads() in the kernel of Figure 4.16?

4. Assuming capacity were not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.

5. For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?
   (A) 1/8 of the original usage
   (B) 1/16 of the original usage
   (C) 1/32 of the original usage

(D) 1/64 of the original usage

6. Assume that a CUDA kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?
   (A) 1
   (B) 1000
   (C) 512
   (D) 512000

7. In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?
   (A) 1
   (B) 1000
   (C) 512
   (D) 51200

8. Consider performing a matrix multiplication of two input matrices with dimensions N×N. How many times is each element in the input matrices requested from global memory when:

   a) There is no tiling?

   b) Tiles of size TxT are used?

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

   a) Peak FLOPS = 200 GFLOPS, Peak Memory Bandwidth = 100 GB/s

   b) Peak FLOPS = 300 GFLOPS, Peak Memory Bandwidth = 250 GB/s

10. To manipulate tiles, a new CUDA programmer has written the following device kernel which will transpose each tile in a matrix. The tiles are of size BLOCK_WIDTH by BLOCK_WIDTH, and each of the dimensions of matrix A is known to be a multiple of BLOCK_WIDTH. The kernel invocation and code are shown below. BLOCK_WIDTH is known at compile-time, but could be set anywhere from 1 to 20.

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

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

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

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

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

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

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

Chapter 5
Performance Considerations

Keywords: compute-bound, memory-bound, bottleneck, memory bandwidth, DRAM burst, memory coalescing, corner turning, memory bank, memory channel, SIMD, control flow, control divergence, dynamic resource partition, instruction mix, thread granularity

CHAPTER OUTLINE

The execution speed of a parallel program can vary greatly depending on the resource constraints of the computing hardware. While managing the interaction between parallel code and hardware resource constraints is important for achieving high performance in virtually all parallel programming models, it is a practical skill that is best learnt with hands-on exercises in a parallel programming model designed for high-performance. In this chapter, we will discuss the major types of resource constraints in a CUDA device and how they can affect the kernel execution performance. In order to achieve his/her goals, a programmer often has to find ways to achieve a required level of performance that is higher than that of an initial version of the application. In different applications, different constraints may dominate and become the limiting factors, commonly referred to as *bottlenecks*. One can often dramatically improve the performance of an application on a particular CUDA device, by trading one resource usage for another. This strategy works well if the resource constraint thus alleviated was actually the dominating constraint before the strategy was applied and the one thus exacerbated does not have negative effects on parallel execution. Without such understanding, performance tuning would be a guess

work; plausible strategies may or may not lead to performance enhancements. Beyond insights into these resource constraints, this chapter further offers principles and case studies designed to cultivate intuition about the type of algorithm patterns that can result in high performance execution. It is also establishes idioms and ideas that will likely lead to good performance improvements during your performance tuning efforts.

5.1. Global Memory Bandwidth

One of the most important factors of CUDA kernel performance is accessing data in the global memory. CUDA applications exploit massive data parallelism. Naturally, CUDA applications tend to process a massive amount of data from the global memory within a short period of time, In Chapter 4, we studied tiling techniques that utilize shared memories to reduce the total amount of data that must be accessed from the global memory by a collection of threads in each thread block. In this chapter, we will further discuss memory coalescing techniques that can more effectively move data from the global memory into shared memories and registers. Memory coalescing techniques are often used in conjunction with tiling techniques to allow CUDA devices to reach their performance potential by more efficiently utilizing the global memory bandwidth.[1]

Why are DRAMs so slow?

The following figure shows a DRAM cell and the path for accessing its content. The decoder is an electronic circuit that uses a transistor to drive a line connected to the outlet gates of thousands of cells. It can take a long time for the line to be fully charged or discharged to the desired level.

A more formidable challenge is for the cell to drive the vertical line to the sense amplifiers and allow the sense amplifier to detect its content. This is based on electrical charge sharing. The gate lets out the tiny amount of electrical charge stored in the cell. If the cell content is "1", the tiny amount of charge must raise the electrical potential of the large capacitance of the long bit line to a sufficiently high level that can trigger the detection mechanism of the sense amplifier. A good analogy would be for someone to hold a small cup of coffee at one end of a long hall way for another person to smell the aroma propagated through the hall way to determine the flavor of the coffee.

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

The global memory of a CUDA device is implemented with DRAMs. Data bits are stored in DRAM cells that are small capacitors, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1. Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to a sensor and set off its detection mechanism that determines whether a sufficient amount of charge is present in the capacitor to qualify as a "1" (see "Why is DRAM so slow?" sidebar). This process takes 10s of nanoseconds in modern DRAM chips. This is in sharp contrast with the sub-nanosecond clock cycle time of modern computing devices. Because this is a very slow process relative to the desired data access speed (sub-nanosecond access per byte), modern DRAMs use parallelism to increase their rate of data access, commonly referred to as memory access throughput.

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

Recognizing the burst organization of modern DRAMs, current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. That is, the most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations. In this case, the hardware combines, or *coalesces*, all these accesses into a consolidated access to consecutive DRAM locations. For example, for a given load instruction of a warp, if thread 0 accesses global memory location N[2], thread 1 location N+1, thread 2 location N+2, and so on, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAMs. Such coalesced access allows the DRAMs to deliver data as a burst.[3]

[2] Different CUDA devices may also impose alignment requirements on N. For example, in some CUDA devices, N is required to be aligned to 16-word boundaries. That is, the lower 6 bits of N should all be 0 bits. Such alignment requirements have been relaxed in recent CUDA devices due to the presence of second-level caches.

[3] Note that modern CPUs also recognize the DRAM burst organization in their cache memory design. A CPU cache line typically maps to one or more DRAM bursts. Applications that make full use of bytes in

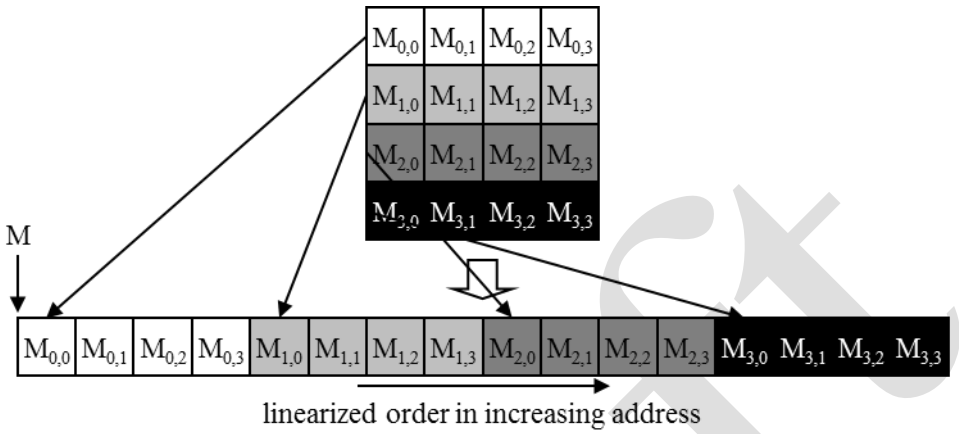


Figure 5.1: *Placing matrix elements into linear order*

In order to understand how to effectively use the coalescing hardware, we need to review how the memory addresses are formed in accessing C multi-dimensional array elements. Recall from Chapter 3 (Figure 3.3, replicated as Figure 5.1 for convenience) that multi-dimensional array elements in C and CUDA are placed into the linearly addressed memory space according to the row major convention. The term *row major* refers to the fact that the placement of data preserves the structure of rows: all adjacent elements in a row are placed into consecutive locations in the address space. In Figure 5.1, the four elements of row 0 are first placed in their order of appearance in the row. Elements in row 1 are then placed, followed by elements of row 2, followed by elements of row 3. It should be clear that $M_{0,0}$ and $M_{1,0}$, though appear to be consecutive in the two dimensional matrix, are placed four locations away in the linearly addressed memory.

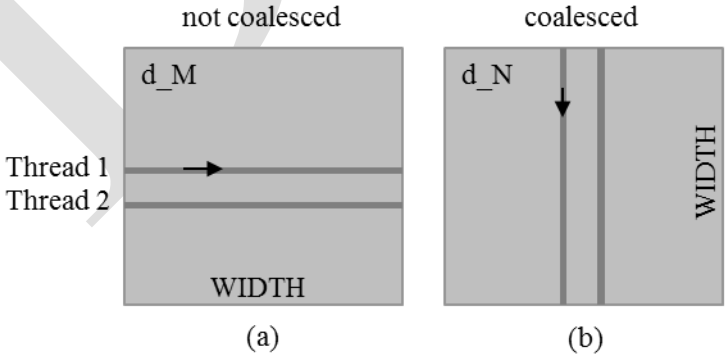


Figure 5.2*: Memory access patterns in C 2D arrays for coalescing*

each cache line they touch tend to achieve much higher performance than those that randomly access memory locations. The techniques presented in this chapter can be adapted to help CPU programs to achieve high performance.

Figure 5.2 illustrates the favorable vs. unfavorable CUDA kernel 2D row-major array data access patterns for memory coalescing. Recall from Figure 4.7 that in our simple matrix multiplication kernel, each thread accesses a row of the M array and a column of the N array. The reader should review Section 4.3 before continuing. Part (a) of Figure 5.2 illustrates the data access pattern the M array, where threads in a warp read adjacent rows. That is, during iteration 0, threads in a warp read element 0 of rows 0 through 31. During iteration 1, these same threads read element 1 of rows 0 through 31. None of the accesses will be coalesced. A more favorable access pattern is shown in Figure 5.2(b), where each thread reads a column of N. During iteration 0, threads in warp 0 reads element 1 of columns 0 through 31. All these accesses will be coalesced.

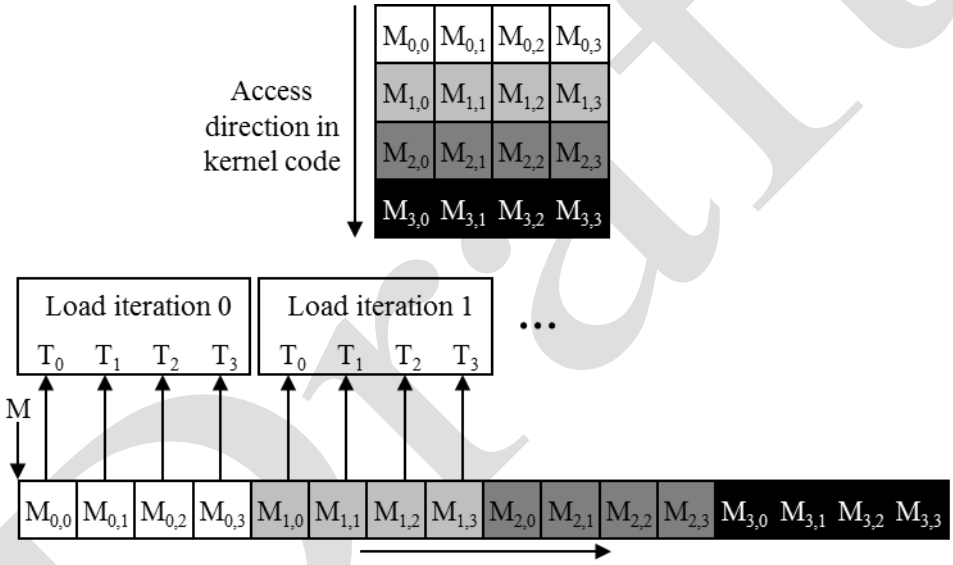


Figure 5.3: *A coalesced access pattern*

In order to understand why the pattern in 5.2(b) is more favorable than that in 5.2(a), we need to review how these matrix elements are accessed in more detail. Figure 5.3 shows a small example of the favorable access pattern in accessing a 4×4 matrix. The arrow in the top portion of Figure 5.3 shows the access pattern of the kernel code. This access pattern is generated by the access to N in Figure 4.3:

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

Within a given iteration of the k loop, the k*Width value is the same across all threads. Recall that Col = blockIdx.x*blockDim.x+threadIdx.x. Since the value of blockIndx.x and blockDim.x are of the same value for all threads in the same block, the only part of k*Width+Col that vary across a thread block is threadIdx.x. Since adjacent threads have

consecutive threadIdx.x values, their accessed elements will have consecutive addresses. For example, in Figure 5.3, assume that we are using 4×4 blocks and that the warp size is 4. That is, for this toy example, we are using only 1 block to calculate the entire P matrix. The values of Width, blockDim.x, blockIdx.x are 4, 4, and 0 for all threads in the block. In iteration 0, the k value is 0. The index used by each thread for accessing N is

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

That is, within this thread block, the index for accessing N is simply the value of threadIdx.x. The N elements accessed by T_0, T_1, T_2, T_3 are N[0], N[1], N[2], and N[3]. This is illustrated with the "Load iteration 0" box of Figure 5.3. These elements are in consecutive locations in the global memory. The hardware detects that these accesses are made by threads in a warp and to consecutive locations in the global memory. It coalesces these accesses into a consolidated access. This allows the DRAMs to supply data at high rate.

During the next iteration, the k value is 1. The index used by each thread for accessing N becomes:

```
N[k*Width+Col]      = N[k*Width+blockIdx.x*blockDim.x+threadIdx.x]
                    = N[1*4 + 0*4 + threadidx.x]
                    = N[4+threadIdx.x]
```

The N elements accessed by T_0, T_1, T_2, T_3 in this iteration are N[5], N[6], N[7], and N[8], as shown with the "Load iteration 1" box in Figure 5.3. All these accesses are again coalesced into a consolidated access for improved DRAM bandwidth utilization.

Figure 5.4 shows an example of matrix data access pattern that is not coalesced. The arrow in the top portion of the figure shows that the kernel code for each thread accesses elements of a row in sequence. The arrow in the top portion of Figure 6.9 shows the access pattern of the kernel code for one thread. This access pattern is generated by the access to M in Figure 4.3: M[Row*Width + k].

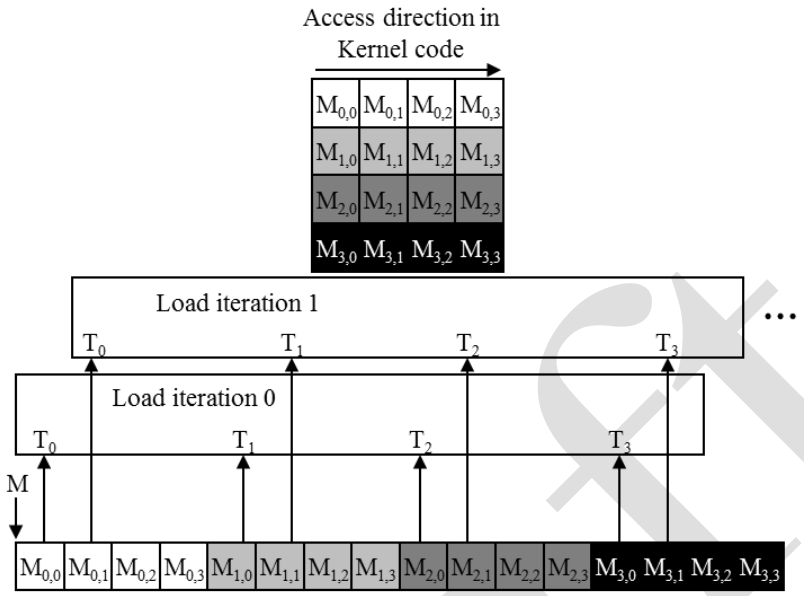


Figure 5.4: *An un-coalesced access pattern*

Within a given iteration of the k loop, k*Width value is the same across all threads. Recall from Figure 4.3 that Row = blockIdx.y*blockDim.y + threadIdx.y. Since the value of blockIndx.y and blockDim.y are of the same value for all threads in the same block, the only part of Row*Width+k that can vary across a thread block is threadIdx.y. In Figure 5.4, we assume again that we are using 4×4 blocks and that the warp size is 4. The values of Width, blockDim.y, blockIdx.y are 4, 4, and 0 for all threads in the block. In iteration 0, the k value is 0. The index used by each thread for accessing M is

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

That is, the index for accessing M is simply the value of threadIdx.x*4. The M elements accessed by T_0, T_1, T_2, T_3 are M[0], M[4], M[8], and M[12]. This is illustrated with the "Load iteration 0" box of Figure 5.4. These elements are not in consecutive locations in the global memory. The hardware cannot coalesce these accesses into a consolidated access.

During the next iteration, the k value is 1. The index used by each thread for accessing M becomes:

```
M[Row*Width+k]      = M[(blockIdx.y*blockDim.y+threadIdx.y)*Width+k]
```


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

The M elements accessed by T_0, T_1, T_2, T_3 are M[1], M[5], M[9], and M[13], as shown with the "Load iteration 1" box in Figure 5.4 Again, these accesses cannot be coalesced into a consolidated access.

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

If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the shared memory to enable memory coalescing. The technique, called *corner turning*, is illustrated in Figure 5.5 for matrix multiplication. Each thread reads a row from M, a pattern that cannot be coalesced. Fortunately, a tiled algorithm can be used to enable coalescing. As we discussed in Chapter 4, threads of a block can first cooperatively load the tiles into the shared memory. **Care must be taken to ensure that these tiles are loaded in a coalesced pattern.** Once the data is in shared memory, they can be accessed either on a row basis or a column basis with much less performance variation because the shared memories are implemented as intrinsically high-speed on-chip memory that does not require coalescing to achieve high data access rate.

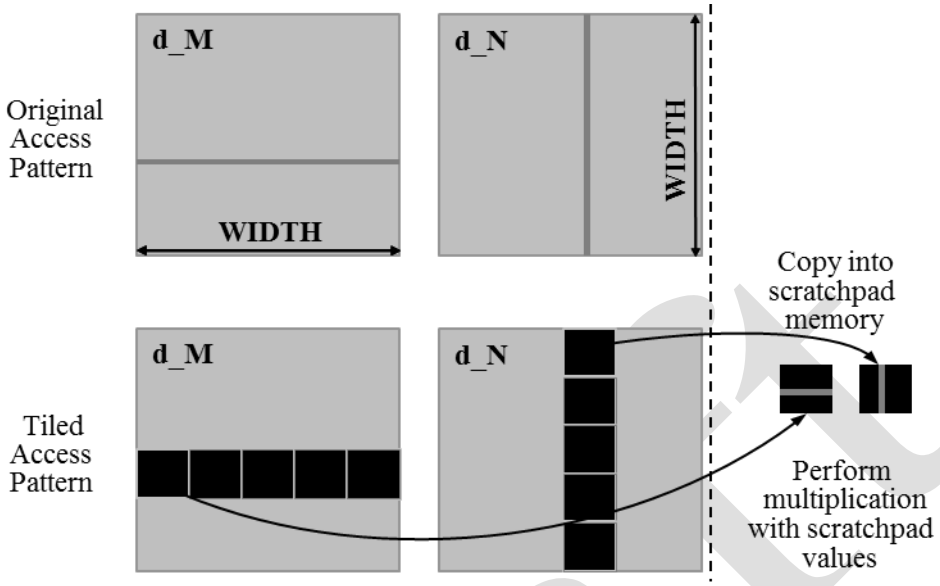


Figure 5.5: *Using shared memory to enable coalescing*

We replicate Figure 4.16 here as Figure 5.6, where the matrix multiplication kernel loads two tiles of matrix M and N into the shared memory. Recall that at the beginning of each phase (Lines 9-11) each thread in a thread block is responsible for loading one M element and one N element into Mds and Nds. Note that there are TILE_WIDTH^2 threads involved in each tile. The threads use threadIdx.y and threadIdx.y to determine the elements to load.

```
    __global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

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

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

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

 7.   float Pvalue = 0;
      // Loop over the M and N tiles required to compute the P element
 8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

        // Collaborative loading of M and N tiles into shared memory
 9.     Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11.      __syncthreads();

12.     for (int k = 0; k < TILE_WIDTH; ++k) {
13.       Pvalue += Mds[ty][k] * Nds[k][tx];
```

```
        }
14.     __syncthreads();
      }
15.   P[Row*Width + Col] = Pvalue;
    }
```

Figure 5.6: *Tiled Matrix Multiplication Kernel using shared memory*

The M elements are loaded in line 9, where the index calculation for each thread uses ph to locate the left end of the tile. The linearized index calculation is equivalent to the two-dimensional array access expression M[Row][ph*TILE_SIZE+tx]. Note that the column index used by the threads only differ in terms of threadIdx. The Row Index is determined by blockIdx.y and threadIdx.y (Line 5), which means that threads in the same thread block with identical blockIdx.y/threadIdx.y and adjacent threadIdx.x values will access adjacent M elements. That is, each row of the tile is loaded by TILE_WIDTH threads whose threadIdx are identical in the y dimension and consecutive in the x dimension. The hardware will coalesce these loads.

In the case of N, the row index ph*TILE_SIZE+ty has the same value for all threads with the same threadIdx.y value. The question is whether threads with adjacent threadIdx.x values access adjacent N elements of a row. Note that the column index calculation for each thread, Col = bx*TILE_SIZE+tx (see line 4). The first term, bx*TILE_SIZE, is the same for all threads in the same block. The second term, tx, is simply the threadIdx.x value. Therefore, threads with adjacent threadIdx.x values access adjacent N elements in a row. The hardware will coalesce these loads.

Note that in the simple matrix multiplication algorithm, threads with adjacent threadIdx.x values access vertically adjacent elements that are not physically adjacent in the row major layout. The tiled algorithm "transformed" this into a different access pattern where threads with adjacent threadIdx.x values access horizontally adjacent elements. That is we turned a vertical access pattern into a horizontal access pattern, which is sometimes referred to as *corner turning*. Corner turning could be also used to turn a horizontal access pattern into a vertical access pattern, which is beneficial in languages such as FORTRAN where 2D arrays are laid out in column-major order.

In the tiled algorithm, loads to both the M and N elements are coalesced. Therefore, the tiled matrix multiplication algorithm has two advantages over the simple matrix multiplication. First, number of memory loads are reduced due to the reuse of data in the shared memory. Second, the remaining memory loads are coalesced so the DRAM bandwidth utilization is further improved. These two improvements have multiplicative effect on each other and result in very significant increased execution speed of the kernel. On a current generation device, the tiled kernel can run more than 30x faster than the simple kernel.

Lines 5, 6, 9, 10 in Figure 5.6 form a frequently used programming pattern for loading matrix elements into shared memory in tiled algorithms. We would also like to encourage the reader to analyze the data access pattern by the dot-product loop in lines 12 and 13. Note that the threads in a warp do not access consecutive location of Mds. This is not a problem since Mds is in shared memory, which does not require coalescing to achieve high speed data access.

5.2. More on Memory Parallelism

As we explained in Section 5.1, DRAM bursting is a form of parallel organization: multiple locations around are accessed in the DRAM core array in parallel. Modern However, bursting alone is not sufficient to realize the level of DRAM access bandwidth required by modern processors. DRAM systems typically employ two more forms of parallel organization – banks and channels. At the highest level, a processor contains one or more channels. Each channel is a memory controller with a bus that connects a set of DRAM banks to the processor. Figure 5.7 illustrates a processor that contains four channels, each with a bus that connects four DRAM banks to the processor. In real systems, a processor typically have one to eight channels and each channel is connected a large number of banks.

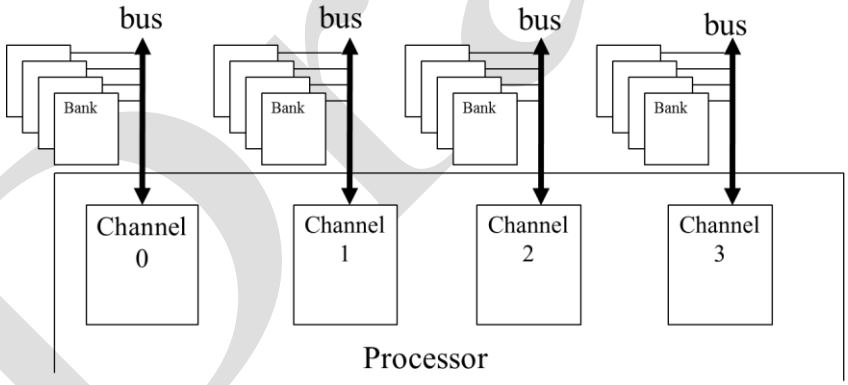


Figure 5.7: *Channels and banks in DRAM systems*

The data transfer bandwidth of a bus is defined by its width and clock frequency. Modern *double data rate* (DDR) busses perform two data transfers per clock cycle, one at the rising edge and one at the falling edge of each clock cycle. For example, a 64-bit DDR bus with a clock frequency of 1 GHz has a bandwidth of 8B * 2 * 1GHz = 16GB/sec. This seems to be a large number but is often too small for modern CPUs and GPUs. A modern CPU might require a memory bandwidth of at least 32 GB/sec whereas a modern GPU might require 128 GB/s. For this example, the CPU would require 2 channels and the GPU would require 8 channels.

For each channel, the number of banks connected to it is determined by the number of banks required to fully utilize the data transfer bandwidth of the bus. This is illustrated in Figure 5.8. Each bank contains an array of DRAM cells, the sensing amplifiers for accessing these cells, and the interface for delivering bursts of data to the bus (Section 5.1).

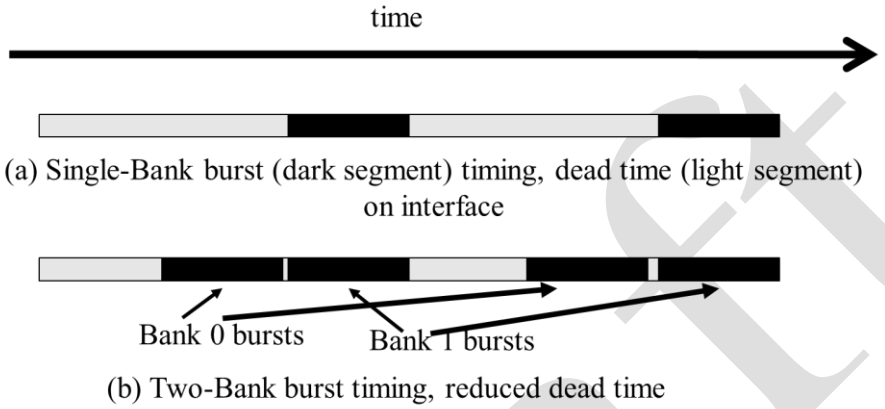


Figure 5.8: *Banking improves the utilization of data transfer bandwidth of a channel*

Figure 5.8(a) illustrates the data transfer timing when a single bank is connected to a channel. It shows the timing of two consecutive memory read accesses to the DRAM cells in the bank. Recall from Section 5.1 that each access involves a long latency for the decoder to enable the cells and for the cells to share their stored charge with the sensing amplifier. This latency is shown as the light grey section at the left end of the time frame. Once the sensing amplifier completes its work, the burst data is delivered through the bus. The time for transferring the burst data through the bus is shown as the left dark section of the time frame in Figure 5.8. The second memory read access will incur a similar long access latency (light section between the dark sections of the time frame) before its burst data can be transferred (right dark section).

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

When two banks are connected to a channel bus, an access can be initiated in the second bank while the first bank is serving another access. Therefore, one can overlap the latency for accessing the DRAM cell arrays. Figure 5.8(b) shows the timing of a two-bank

organization. We assume that the bank 0 started at a time earlier than the window shown in Figure 5.8. Shortly after the first bank starts accessing its cell array, the second bank also starts access its cell array. When the access in bank 0 is complete, it transfers the burst data (leftmost dark section of the time frame). Once bank 0 completes is data transfer, bank 1 can transfer its burst data (second dark section). This pattern repeats for the next accesses.

From Figure 5.8(b), we can see that by having two banks, we can potentially double the utilization of the data transfer bandwidth of the channel bus. In general, if the ratio of the cell array access latency and data transfer time is R, we need to have at least R+1 banks if we hope to fully utilize the data transfer bandwidth of the channel bus. For example, if the ratio is 20, we will need at least 21 banks connected to each channel bus. In reality, the number of banks connected to each channel bus needs to be larger than R for two reasons. One is that having more banks reduces the probability of multiple simultaneous accesses targeting the same bank, a phenomenon called bank conflict. Since each bank can serve only one access at a time, the cell array access latency can no longer be overlapped for these conflicting accesses. Having a larger number of banks increases the probability that these accesses will be spread out among multiple banks. The second reason is that the size of each cell array is set to achieve reasonable latency and manufacturability. This limits the number of cells that each bank can provide. One may need a large number of banks just to be able to support the memory size required.

There is an important connection between the parallel execution of threads and the parallel organization of the DRAM system. In order to achieve the memory access bandwidth specified for device, there must be a sufficient number of threads making simultaneous memory accesses. Furthermore, these memory accesses must be evenly distributed to the channels and banks. Of course, each access to a bank must also be a coalesced access, as we studied in Section 5.1.

Figure 5.9 shows a toy example of distributing array M elements to channels and banks. We assume a small burst size of two elements (eight bytes). The distribution is done by hardware design. The addressing of the channels and backs are such that the first eight bytes of the array (M[0] and M[1]) are stored in bank 0 of channel 0, the next eight bytes (M[2] and M[3]) in bank 0 of channel 1, the next eight bytes (M[4] and M[5]) in bank 0 of channel 2, and the next eight bytes (M[6] and M[7]) in bank 0 of channel 3.

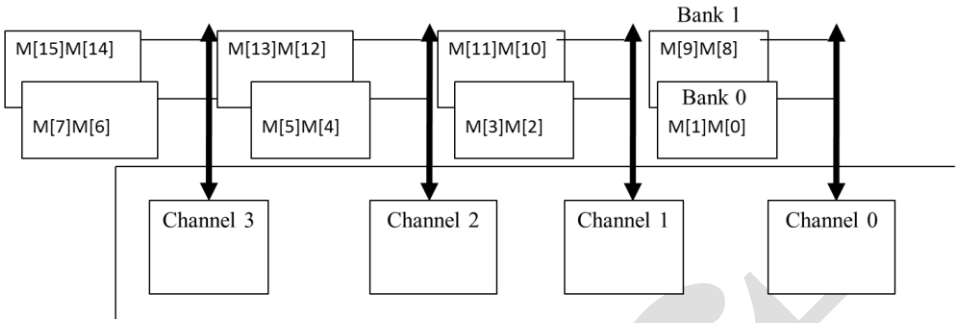


Figure 5.9: *Distributing array elements into channels and banks.*

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

The distribution scheme illustrated in Figure 5.9, often referred to as *interleaved data distribution*, spreads the elements across the banks and channels in the system. This scheme ensures that even relatively small arrays are spread out nicely. That is, we only assign enough elements to fully utilize the DRAM burst of bank 0 of channel before moving on to bank 0 of channel 1. In our toy example, as long as we have at least 16 elements, the distribution will involve all the channels and banks for storing the elements.

We now illustrate the interaction between parallel thread execution and the parallel memory organization. We will use the example in Figure 4.9, replicated as Figure 5.10. We assume that the multiplication will be performed with 2x2 thread blocks and 2x2 tiles.

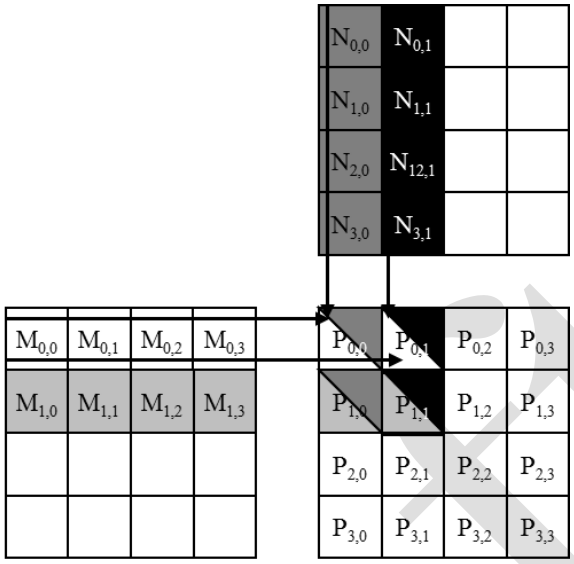


Figure 5.10: *A small example of matrix multiplication (replicated from Figure 4.9).*

During the phase 0 of the kernel's execution, all four thread blocks will be loading their first tile. The M elements involved in each tile are shown in Figure 5.11. Row 2 shows the M elements accessed in Phase 0, with their 2D indices. Row 3 shows the same M elements with their linearized indices. Assume that all thread blocks are executed in parallel. We see that each block will make two coalesced accesses.

Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[19], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

Figure 5.11*: M elements loaded by thread blocks in each phase.*

According to the distribution in Figure 5.9, these coalesced accesses will be made to the two banks in channel 0 as well as the two banks in channel 2. These four accesses will be done in parallel to take advantage of two channels as well as improving the utilization of the data transfer bandwidth of each channel.

We also see that $Block_{0,0}$ and $Block_{0,1}$ will load the same M elements. Most of the modern devices are equipped with a caches that will combine these accesses into one as long as the execution timing of these blocks are sufficiently close to each other. In fact, the cache memories in GPU devices are mainly designed to combine such accesses and reduce the number of accesses to the DRAM system.

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

The reader is invited to verify that multiplying two larger matrices, such as 8x8 with the same 2x2 thread block configuration, will make use all the four channels in Figure 5.9. Also, an increased DRAM burst size would require multiplication of even larger matrices to fully utilize the data transfer bandwidth of all the channels.

5.3. Warps and SIMD Hardware

We now turn our attention to aspects of the thread execution that can limit performance. Recall that launching a CUDA kernel generates a grid of threads that are organized as a two-level hierarchy. At the top level, a grid consists of a one-, two-, or three dimensional array of blocks. At the bottom level, each block, in turn, consists of a one-, two-, or three-dimensional array of threads. In Chapter 3, we saw that blocks can execute in any order relative to each other, which allows for transparent scalability across different devices. However, we did not say much about the execution timing of threads within each block.

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

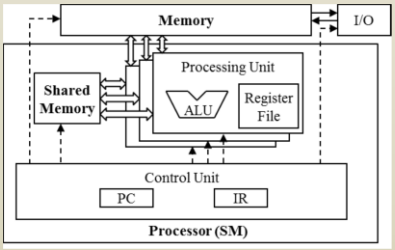
As we discussed in Chapter 3, each thread block is partitioned into *warps*. The execution of warps is done by an SIMD hardware (see "Warps and SIMD Hardware" sidebar). This implementation technique helps to reduce hardware manufacturing cost, lower run-time operation electricity cost, and enable coalescing of memory accesses. In the foreseeable future, we expect that warp partitioning will remain as a popular implementation technique. However, the size of warp can easily vary from implementation to implementation. Up to this point in time, all CUDA devices have used similar warp configurations where each warp consists of 32 threads.

Warps and SIMD Hardware

The motivation for executing threads as warps is illustrated in the following picture (Same as Figure 4.8). The processor has only one control unit that fetches and decodes instructions. The same control signal goes to multiple processing units, each of which executes one of the threads in a warp. Since all processing units are controlled by the same instruction, their execution differences are due to the different data operand values in the register files. This is called Single-Instruction-Multiple-Data (SIMD) in processor design. For example, although all processing units are controlled by an instruction

add r1, r2, r3

the r2 and r3 values are different in different processing units.



Control units in modern processors are quite complex, including sophisticated logic for fetching instructions and access ports to the instruction memory. They include on-chip instructions caches to reduce the latency of instruction fetch. Having multiple processing units to share a control unit can result in significant reduction in hardware manufacturing cost and power consumption.

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

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

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

Figure 5.12 shows an example of placing threads of a two-dimensional block into linear order. The upper part shows the two-dimensional view of the block. The reader should recognize that the similarity with the row-major layout of two-dimensional arrays. Each thread is shown as $T_{y,x}$, x being threadIdx.x and y being threadIdx.y. The lower part of Figure 5.12 shows the linearized view of the block. The first four threads are those threads whose threadIdx.y value is 0; they are ordered with increasing threadIdx.x values. The next four threads are those threads whose threadIdx.y value is 1. They are also placed with increasing threadIdx.x values. For this example, all 16 threads form half a warp. The warp will be padded with another 16 threads to complete a 32-thread warp. Imagine a two-dimensional block with 8×8 threads. The 64 threads will form two warps. The first warp starts from $T_{0,0}$ and ends with $T_{3,7}$. The second warp starts with $T_{4,0}$ and ends with $T_{7,7}$. It would be a useful exercise to draw out the picture as an exercise.

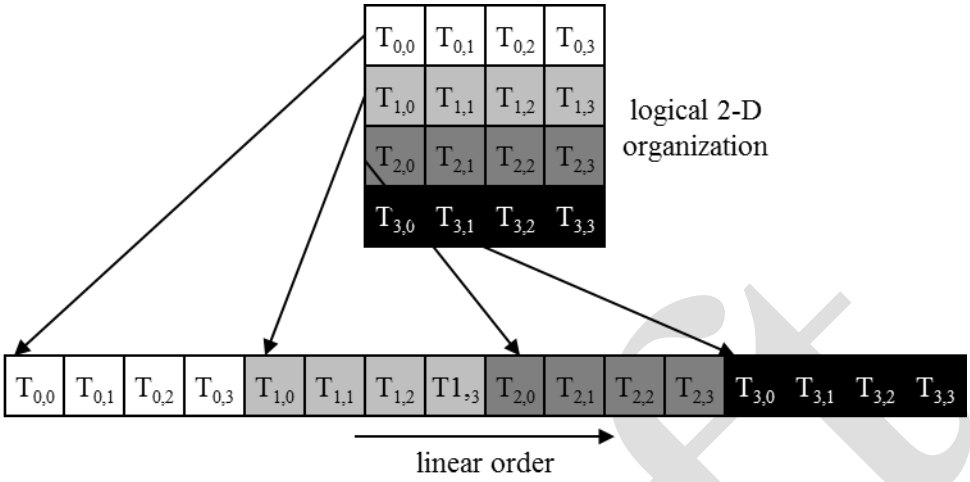


Figure 5.12: *Placing 2D threads into linear order*

For a three-dimensional block, we first place all threads whose threadIdx.z value is 0 into the linear order. Among these threads, they are treated as a two-dimensional block as shown in Figure 5.12. All threads whose threadIdx.z value is 1 will then be placed into the linear order, and so on. A three-dimensional thread block of dimensions 2×8×4 (four in the x dimension, eight in the y dimension, and two in the z dimension), the 64 threads will be partitioned into two warps, with $T_{0,0,0}$ through $T_{0,7,3}$ in the first warp and $T_{1,0,0}$ through $T_{1,7,3}$ in the second warp.

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

The multi-pass approach to divergent warp execution extends the SIMD hardware's ability to implement the full semantics of CUDA threads. While the hardware executes the same instruction for all threads in a warp, it selectively let the threads to take effect in only each pass, allowing every thread to take its own control flow path. This preserves the independence of threads while taking advantage of the reduced cost of SIMD hardware.

When threads in the same warp follow different execution paths, we say that these threads *diverge* in their execution. In the if-else example, divergence arises if some threads in a

warp take the then path and some the else path. The cost of divergence is the extra pass the hardware needs to take in order to allow the threads in a warp to make their own decisions.

Divergence also can arise in other constructs. For example, if threads in a warp execute a `for` loop which can iterate six, seven, or eight times for different threads. All threads will finish the first six iterations together. Two passes will be used to execute the 7th iteration, one for those that take the iteration and one for those that do not. Two passes will be used to execute the 8th iteration, one for those that take the iteration and one for those that do not.

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

A prevalent reason for using a control construct with thread divergence is handling boundary conditions when mapping threads to data. This is usually because the total number of threads needs to be a multiple and an integer (the thread block size) whereas the size of the data can be an arbitrary number. Starting with our vector addition kernel in Figure 2.12, we had an if (i<n) statement in addVecKernel. This is because not all vector lengths can be expressed as multiples of the block size. For example, let's assume that the vector length is 1,003. Assume that we picked 64 as block size. One would need to launch 16 thread blocks to process all the 1,003 vector elements. However, the 16 thread blocks would have 1,024 threads. We need to disable the last 21 threads in thread block 15 from doing work not expected/allowed by the original program. Keep in mind that these 16 blocks are partitioned into 32 warps. Only the last warp will have control divergence.

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

For two-dimensional data, such as the color-to-greyscale conversion example, if-statements are also used to handle the boundary conditions for threads that operate at the

edge of the data. In Figure 3.2, to process the 76x62 picture, we used 20=5*4 two-dimensional blocks that consist of 16x16 threads each. Each block will be partitioned into 8 warps, each one consists of two rows of a block. There are a total 160 warps (8 warps per block) involved.

To analyze the impact of control divergence, refer to Figure 3.5. None of the warps in the 12 blocks in region 1 will have control divergence. There are 12*8 = 96 warps in region 1. For region 2, all the 24 warps will have control divergence. For region three, note that all the bottom warps are mapped to data that are completely outside the picture. As result, none of them will pass the if-condition. The reader should verify that these warps would have had control divergence if the picture had an odd number of pixels in the vertical dimension. Since they all follow the same control flow path, none of these 32 warps will have control divergence! In region 4, the first seven warps will have control divergence but the last warp will not. All in all, 31 out of the 160 warps will have control divergence.

Once again, the performance impact of control divergence decreases as the number of pixels in the horizontal dimension increases. For example, if we process a 200x150 picture with 16x16 blocks, there will be a total 130 = 13*10 thread blocks or 1040 warps. The number of warps in regions 1 through 4 will be 864 (12*9*8), 72 (9*8), 96 (12*8), and 8 (1*8). Only 80 of these warps will have control divergence. Thus, the performance impact of control divergence will be less than 8%. Obviously, if we process a realistic picture with more than 1,000 pixels in the horizontal dimension, the performance impact of control divergence will be less than 2%.

Control divergence also naturally arise in some important parallel algorithms where the number of threads participating in the computation vary over time. We will use a reduction algorithm to illustrate such behavior.

A reduction algorithm derives a single value from an array of values. The single value could be the sum, the maximal value, the minimal value, etc. among all elements. All these types of reductions share the same computation structure. A reduction can be easily done by sequentially going through every element of the array. When an element is visited, the action to take depends on the type of reduction being performed. For a sum reduction, the value of the element being visited at the current step, or the current value, is added to a running sum. For a maximal reduction, the current value is compared to a running maximal value of all the elements visited so far. If the current value is larger than the running maximal, the current element value becomes the running maximal value.

For a minimal reduction, the value of the element currently being visited is compared to a running minimal. If the current value is smaller than the running minimal, the current element value becomes the running minimal. The sequential algorithm ends when all the

elements are visited. The sequential reduction algorithm is work efficient in that every element is only visited once and only a minimal amount of work is performed when each element is visited. Its execution time proportional to the number of elements involved. That is, the computational complexity of the algorithm is O(N), where N is the number of elements involved in the reduction.

The time needed to visit all elements of a large array motivates parallel execution. A parallel reduction algorithm typically resembles the structure of a soccer tournament. In fact, the elimination process of the World Cup is a reduction of "maximal" where the maximal is defined as the team that "beats" all other teams. The tournament "reduction" is done in multiple rounds. The teams are divided into pairs. During the first round, all pairs play in parallel. Winners of the first round advance to the second round, whose winners advance to the third round, etc. With 16 teams entering a tournament, eight winners will emerge from the first round, four from the second round, two from the third round, and one final winner from the fourth round.

It should be easy to see that even with 1024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough soccer fields to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on. With enough fields, even with sixty thousand teams, we can determine the final winner in just 16 rounds. Of course, one would need to have enough soccer fields and enough officials to accommodate the thirty thousand games in the first round, etc.

Figure 5.13 shows a kernel function that performs parallel sum reduction. The original array is in the global memory. Each thread block reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction on these elements. The code that loads the elements of the input array `X` from global memory into the shared memory. The reduction is done *in place*, which means some of the elements in the shared memory will be replaced by partial sums. Each iteration of the while-loop in the kernel function implements a round of reduction.

```
1.    __shared__ float partialSum[SIZE];
      partialSum[threadIdx.x] = X[threadIdx.x];
2.    unsigned int t = threadIdx.x;
3.    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
4.    {
5.      __syncthreads();
6.      if (t % (2*stride) == 0)
7.        partialSum[t] += partialSum[t+stride];
8.    }
```

Figure 5.13: *A simple sum reduction kernel*

The __syncthreads() statement (Line 5) in the for-loop ensures that all partial sums for the previous iteration have been generated and before any one of threads is allowed to begin the current iteration. This way, all threads that enter the second iteration will be using the values produced in the first iteration. After the first round, the even elements will be replaced by the partial sums generated in the first round. After the second round, the elements whose indices are multiples of four will be replaced with the partial sums. After the final round, the total sum of the entire section will be in partialSum[0].

In Figure 5.13, Line 3 initializes the `stride` variable to 1. During the first iteration, the if-statement in Line 6 is used to select only the even threads to perform addition between two neighboring elements. The execution of the kernel is illustrated in Figure 5.14. The threads and the array element values are shown in the horizontal direction. The iterations taken by the threads are shown in the vertical direction with time progressing from top to bottom. Each row of Figure 5.14 shows the contents of the array elements after an iteration of the for-loop.

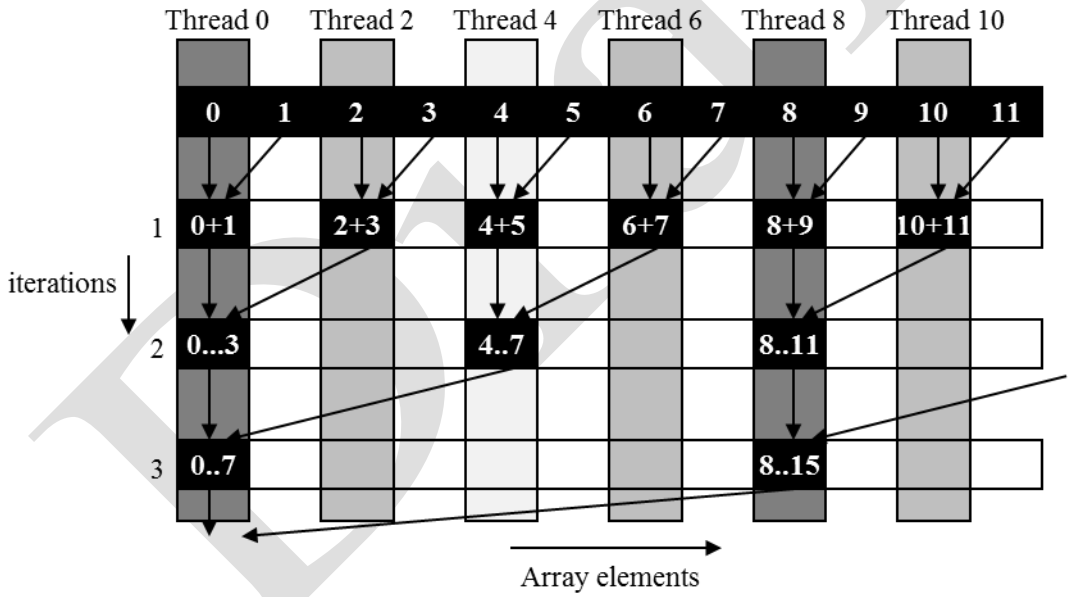


Figure 5.14: *Execution of the sum reduction kernel*

As shown in Figure 5.16, the even elements of the array hold the pair-wise partial sums after iteration 1. Before the second iteration, the value of the stride variable is doubled to 2. During the second iteration, only those threads whose indices are multiples of four will execute the add-statement in Line 8. Each thread generates a partial sum of four elements, as shown in row 2. With 512 elements in each section, the kernel function will generate

the sum of the entire section after 9 iterations. By using blockDim.x as the loop bound in Line 4, the kernel assumes that it is launched with the same number of threads as the number of elements in the section. That is, for section size of 512, the kernel needs to be launched with 512 threads.[4]

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

The kernel in Figure 5.13 clearly has thread divergence. During the first iteration of the loop, only those threads whose threadIdx.x are even will execute the add statement. One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute Line 8. In each successive iteration, fewer threads will execute Line 8 but two passes will be still needed to execute all the threads during each iteration. This divergence can be reduced with a slight change to the algorithm.

```
1.   __shared__ float partialSum[SIZE];
    partialSum[threadIdx.x] = X[threadIdx.x];

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

Figure 5.15: *A kernel with fewer thread divergence*

Figure 5.15 shows a modified kernel with a slightly different algorithm for sum reduction. Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other. It does so by initializing the stride to be half the size of the section. All pairs added during the first round are half the section size away from each other. After the first iteration, all the pair-wise sums are stored in the first half of the array, as shown in Figure 5.16. The loop divides the stride by 2 before entering the next iteration. Thus for the second iteration, the stride variable value is quarter of the section size. That

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

is, the threads add elements that are quarter a section away from each other during the second iteration.

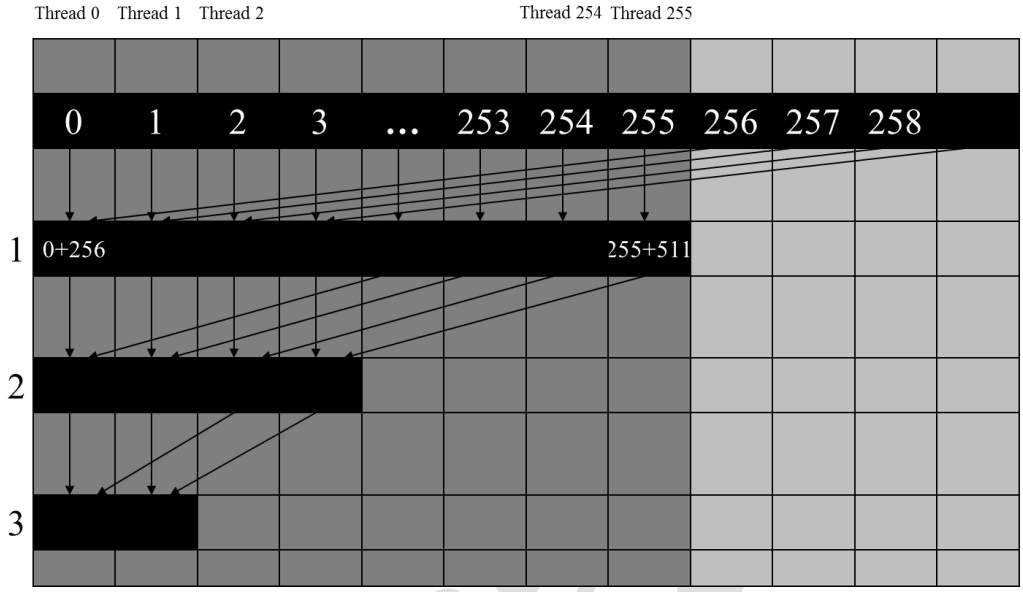


Figure 5.16: *Execution of the revised algorithm*

Note that the kernel in Figure 5.15 still has an if-statement (Line 6) in the loop. The number of threads that execute Line 7 in each iteration is the same as in Figure 5.13. So, why should there be a performance difference between the two kernels? The answer lies in the positions of threads that execute Line 7 relative to those that do not.

Figure 5.16 illustrates the execution of the revised kernel in Figure 5.15. During the first iteration, all threads whose threadIdx.x value are less than half of the size of the section execute Line 7. For a section of 512 elements, Threads 0 through 255 execute the add statement during the first iteration while threads 256 through 511 do not. The pair-wise sums are stored in elements 0 through 255 after the first iteration. Since the warps consists of 32 threads with consecutive threadIdx.x values, all threads in warps 0 through warp 7 execute the add statement whereas warp 8 through warp 15 execute all skip the add statement. Since all threads in each warp take the same path, there is no thread divergence!

The kernel in Figure 5.15 does not completely eliminate the divergence caused by the if-statement. The reader should verify that starting with the 5^{th} iteration, the number of threads that execute Line 7 will fall below 32. That is, the final five iterations will have only 16, 8, 4, 2, and 1 thread(s) performing the addition. This means that the kernel execution will still

have divergence in these iterations. However, the number of iterations of the loop that has divergence is reduced from ten to five.

The difference between Figure 5.13 and Figure 5.15 is small but has very significant performance impact. It requires someone with clear understanding of the execution of threads on the SIMD hardware of the device to be able to confidently make such adjustments.

5.4. Dynamic Partitioning of Resources

The execution resources in an SM include registers, shared memory, thread block slots, and thread slots. These resources are dynamically partitioned and assigned to threads to support their execution. In Chapter 3, we have seen that current generation of devices have 1,536 thread slots. These thread slots are partitioned and assigned to thread blocks during runtime. If each thread block consists of 512 threads, the 1,536 threads slots are partitioned and assigned to three blocks. In this case, each SM can accommodate up to three thread blocks due to limitations on thread slots.

If each thread block contains 256 threads, the 1,536 thread slots are partitioned and assigned to 6 thread blocks. The ability to dynamically partition the thread slots among thread blocks makes SMs versatile. They can either execute many thread blocks each having few threads or execute few thread blocks each having many threads. This is in contrast to a fixed partitioning method where each block receives a fixed amount of resource regardless of their real needs. Fixed partitioning results in wasted thread slots when a block has few threads and fails to support blocks that require more thread slots than the fixed partition allows.

Dynamic partitioning of resources can lead to subtle interactions between resource limitations, which can cause underutilization of resources. Such interactions can occur between block slots and thread slots. For example, if each block has 128 threads, the 1536 thread slots can be partitioned and assigned to 12 blocks. However, since there are only 8 block slots in each SM, only 8 blocks will be allowed. This means that in the end, only 1024 of the thread slots will be utilized. Therefore, to fully utilize both the block slots and thread slots, one needs at least 256 threads in each block.

As we mentioned in Chapter 4, the automatic variables declared in a CUDA kernel are placed into registers. Some kernels may use lots of automatic variables and others may use few of them. Thus, one should expect that some kernels require many registers and some require fewer. By dynamically partitioning the registers among blocks, the SM can accommodate more blocks if they require few registers and fewer blocks if they require

more registers. One does, however, need to be aware of potential interactions between register limitations and other resource limitations.

In the matrix multiplication example, assume that each SM has 16,384 registers and the kernel code uses 10 registers per thread. If we have 16×16 thread blocks, how many threads can run on each SM? We can answer this question by first calculating the number of registers needed for each block, which is 10*16* 16= 2,560. The number of registers required by six blocks is 15,360, which is under the 16,384 limit. Adding another block would require 17,920 registers, which exceeds the limit. Therefore, the register limitation allows six blocks that altogether have 1,536 threads to run on each SM, which also fits within the limit of 8 block slots and 1,536 thread slots.

Now assume that the programmer declares another two automatic variables in the kernel and bumps the number of registers used by each thread to 12. Assuming the same 16x16 blocks, each block now requires 12*16*16 = 3,072 registers. The number of registers required by six blocks is now 18,432, which exceeds the register limitation for some CUDA hardware. The CUDA runtime system deals with this situation by reducing the number of blocks assigned to each SM by one, thus reducing the number of registered required to 15,360. This, however, reduces the number of threads running on an SM from 1,536 to 1,280. That is, by using two extra automatic variables, the program saw a 1/6 reduction in the warp parallelism in each SM. This is sometimes a referred to as a "performance cliff" where a slight increase in resource usage can result in significant reduction in parallelism and performance achieved [RRS2008].

Shared memory is another resource that is dynamically partitioned at run time. Tiled algorithms often require a large amount of shared memory to be effective. Unfortunately, large shared memory usage can reduce the number of thread blocks running on an SM. As we discussed in Section 5.3, reduced thread parallelism can negatively affect the utilization of the memory access bandwidth of the DRAM system. The reduced memory access throughput, in turn, can further reduce the thread execution throughput. This is a pitfall that can result in disappointing performance of tiled algorithms and should be carefully avoided.

It should be clear to the reader that the constraints of all the dynamically partitioned resources interact with each other in a complex manner. Accurate determination of the number of threads running in each SM can be difficult. The reader is referred to the CUDA Occupancy Calculator [NVIDIA] which is a downloadable Excel sheet that calculates the actual number of threads running on each SM for a particular device implementation given the usage of resources by a kernel.

5.5. Thread Granularity

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

Figure 5.17 illustrates such an opportunity in matrix multiplication. The tiled algorithm in Figure 5.6 uses one thread to compute one element of the output P matrix. This requires a dot product between one row of M and one column of N.

The opportunity of thread granularity adjustment comes from the fact that multiple blocks redundantly load each M tile. This was also demonstrated in Figure 5.11. As shown in Figure 5.17, the calculation of two P elements in adjacent tiles uses the same M row. With the original tiled algorithm, the same M row is redundantly loaded by the two blocks assigned to generate these two Pd tiles. One can eliminate this redundancy by merging the two thread blocks into one. Each thread in the new thread block now calculates two P elements. This is done by revising the kernel so that two dot-products are computed by the innermost loop of the kernel. Both dot products use the same Mds row but different Nds columns. This reduces the global memory access by ¼. The reader is encouraged to write the new kernel as an exercise.

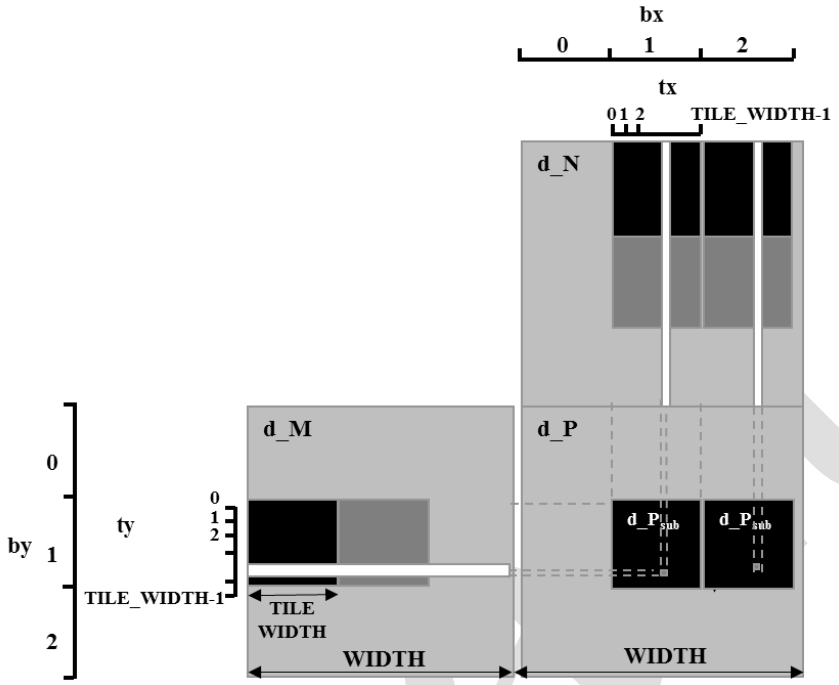


Figure 5.17: *Increased thread granularity with rectangular tiles*

The potential downside is that the new kernel now uses even more registers and shared memory. As we discussed in the previous section, the number of blocks that can be running on each SM may decrease. For a given matrix size, tt also reduces the total number of thread blocks by half, which may result in insufficient amount of parallelism for matrices of smaller dimensions. In practice, combining up to four adjacent horizontal blocks to compute adjacent horizontal tiles significantly improves the performance of large (2048x2048 or more) matrix multiplication.

5.6. Summary

In this chapter, we reviewed the major aspects of application performance on a CUDA device: global memory access coalescing, memory parallelism, control flow divergence, dynamic resource partitioning and instruction mixes. Each of this aspects is rooted in the hardware limitations of the devices. With these insights, the reader should be able to reason about the performance of any kernel code he/she comes across.

More importantly, we need to be able to convert poor performing code into well performing code. As a starting point, we presented practical techniques for creating good program patterns for these performance aspects. We will continue to study practical applications of

these techniques in the parallel computation patterns and application case studies in the next few chapters.

References

[NVIDIA] CUDA Occupancy Calculator. Web search using keywords “CUDA Occupancy Calculator”

[NVIDIA2012] CUDA C Best Practices Guide v. 4.2, January 2012.

[RRS2008] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, W. Hwu, “Program Optimization Space Pruning for a Multithreaded GPU,”Proceedings of the 6th ACM/IEEE International Symposium on Code Generation and Optimization, April 6-9, 2008.

[RRB2008] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu , “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA,” Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2008.

5.7 Exercises

1. The kernels in Figure 5.13 and 5.15 are wasteful in their use of threads; half of the threads in each block never execute. Modify the kernels to eliminate such waste. Give the relevant execute configuration parameter values at the kernel launch. Is there a cost in terms of extra arithmetic operation needed? Which resource limitation can be potentially address with such modification? (Hint: (1) Line 2 and/or Line 4 can be adjusted in each case. (2) The number of elements in a section may need to increase.)

2. Compare the modified kernels you wrote for Exercise 5.1. Which kernel incurred fewer additional arithmetic operations from the modification?

3. Write a complete kernel based on Exercise 5.1 by (1) adding the statements that load a section of the input array from global memory to shared memory, (2) using blockIdx.x to allow multiple blocks to work on different sections of the input array, (3) write the reduction value for the section to a location according to the blockIdx.x so that all

blocks will deposit their section reduction value to the lower part of the input array in global memory.

4. Design a reduction program based on the kernel you wrote for Exercise 6.3. The host code should (1) transfer a large input array to the global memory, (2) use a loop to repeatedly invoke the kernel you wrote for Exercise 5.3 with adjusted execution configuration parameter values so that the reduction result for the input array will eventually be produced.

5. For the tiled matrix multiplication kernel in Figure 5.6, draw the access patterns of threads in a warp of Lines 9 and 10 for a small 16×16 matrix size. Calculate the tx values and ty values for each thread in a warp and used these values in the M and N index calculations in Lines 9 and 10. Show that the threads indeed access consecutive M and N locations in global memory during each iteration.

6. For the simple matrix multiplication (P=M*N) based on row-major layout, which input matrix will have coalesced accesses?
   (A) M
   (B) N
   (C) M, N
   (D) Neither

7. For the tiled matrix-matrix multiplication (MxN) based on row-major layout, which input matrix will have coalesced accesses?
   (A) M
   (B) N
   (C) M, N
   (D) Neither

8. For the simple reduction kernel, if the block size is 1024 and warp size is 32, how many warps in a block will have divergence during the 5^{th} iteration?
   (A) 0
   (B) 1
   (C) 16
   (D) 32

9. For the improved reduction kernel, if the block size is 1024 and warp size is 32, how many warps will have divergence during the 5^{th} iteration?

(A) 0
(B) 1
(C) 16
(D) 32

10. Write a matrix multiplication kernel function that corresponds to the design illustrated in Figure 5.17.

11. For tiled matrix multiplication out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will the kernel completely avoid un-coalesced accesses to global memory? (You need to consider only square blocks.)

12. In an attempt to improve performance, a bright young engineer changed the reduction kernel into the following. (A) Do you believe that the performance will improve? Why or why not? (B) Should the engineer receive a reward or a lecture? Why?

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

Chapter 7

Parallel Patterns: Convolution

An introduction to stencil computation

Keywords: convolution, stencil, tiling, ghost cells, halo cells, tiling efficiency, constant memory, cache, finite difference method

CHAPTER OUTLINE

In the next several chapters, we will discuss a set of important patterns of parallel computation. These patterns are the basis of a wide range of parallel algorithms that appear in many parallel applications. We will start with convolution, which is a popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision. In these application areas, convolution is often performed as a filter that transforms signals and pixels into more desirable values. Our image blur kernel is such a filter that smooths out the signal values so that one can see the big-picture trend. For another example, Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images.

In high-performance computing, the convolution pattern is often referred to as stencil computation, which appears widely in numerical methods for solving differential equations. It also forms the basis of many force calculation algorithms in simulation models. Convolution typically involves a significant number of arithmetic operations on each data element. For large data sets such as high-definition images and videos, the amount of computation can be very large. Each output data element can be calculated independently of each other, a desirable trait for parallel computing. On the other hand, there is substantial

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

7.1. Background

Convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements. The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the *convolution kernel*. Since there is an unfortunate name conflict between the CUDA kernel functions and convolution kernels, we will refer to these mask arrays as *convolution masks* to avoid confusion. The same convolution mask is typically used for all elements of the array.

In audio digital signal processing, the input data are in 1D form and represent sampled signal volume as a function of time. Figure 7.1 shows a convolution example for 1D data where a 5-element convolution mask array M is applied to a 7-element input array N. We will follow the C language convention where N and P elements are indexed from 0 to 6 and M elements are indexed from 0 to 4. The fact that we use a 5-element mask M means that each P element is generated by a weighted sum of the N element at the corresponding position, two N elements to the left and two N elements to the right.

For example, the value of P[2] is generated as the weighted sum of N[0] (i.e., N[2-2]) through N[4] (i.e., N[2+2]). In this example, we arbitrarily assume that the values of the N elements are 1, 2, 3, …,7. The M elements define the weights, whose values are 3, 4, 5, 4, 3 in this example. Each weight value is multiplied to the corresponding N element values before the products are summed together. As shown in Figure 7.1, the calculation for P[2] is as follows:

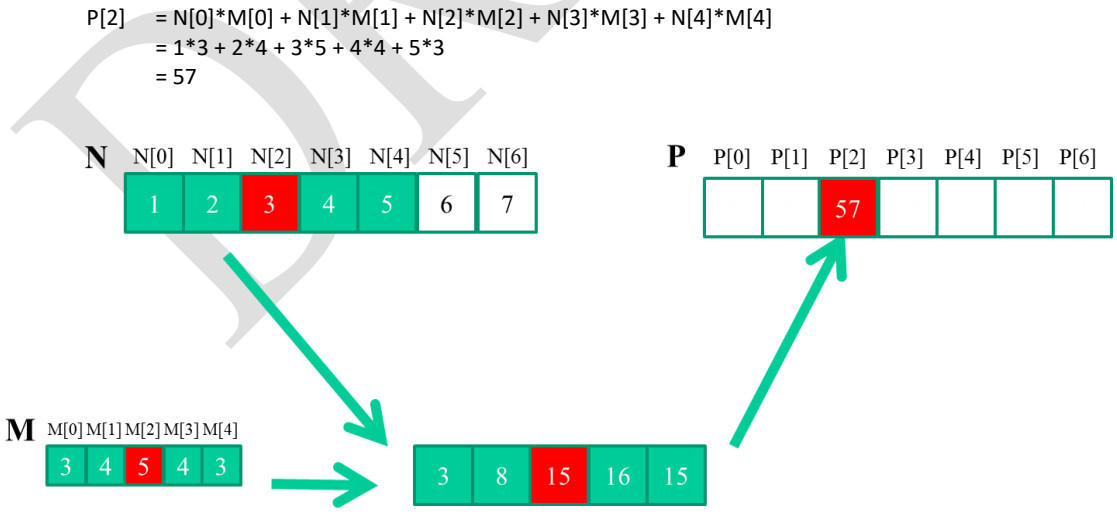


Figure 7.1 A 1D convolution example, inside elements.

In general, the size of the mask tends to be an odd number, which makes the weighted sum calculation symmetric around the element being calculated. That is, an odd number of mask elements defines the weighted sum to include the same number of elements on each side of the element being calculated. In Figure 7.1, the mask size is 5 elements. Each output element is calculated as the weighted sum of the corresponding input element, two elements on the left, and two elements on the right.

In Figure 7.1, the calculation for P[i] can be viewed as an inner product between the sub array of N that starts at N[i-2] and the M array. Figure 7.2 shows the calculation for P[3]. The calculation is shifted by one N element from that of Figure 7.1. That is the value of P[3] is the weighted sum of N[1] (i.e., N[3-2]), through N[5] (i.e., N[3+2]). We can think of the calculation for P[3] is as follows:

```
P[3]  = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]
      = 2*3 + 3*4 + 4*5 + 5*4 + 6*3
      = 76
```

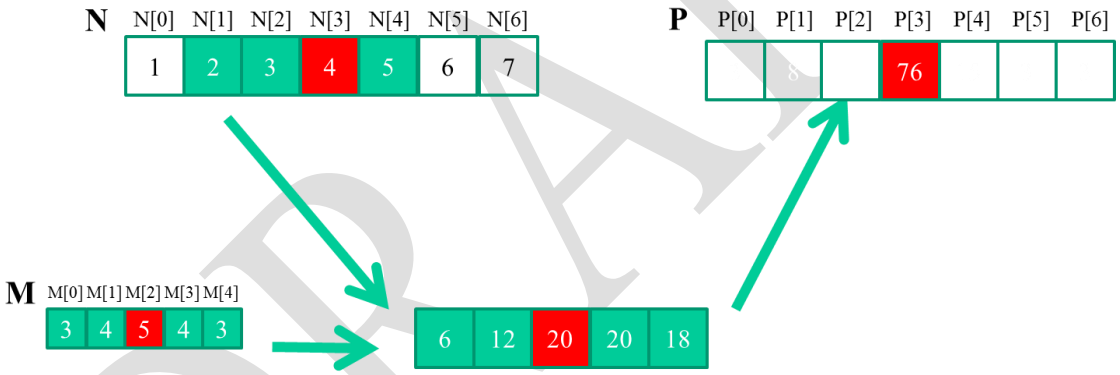


Figure 7.2 1D convolution, calculation of P[3]

Because convolution is defined in terms of neighboring elements, boundary conditions naturally arise for output elements that are close to the ends of an array. As shown in Figure 7.3, when we calculate P[1], there is only one N element to the left of N[1]. That is, there are not enough N elements to calculate P[1] according to our definition of convolution. A typical approach to handling such boundary condition is to define a default value to these missing N elements. For most applications, the default value is 0, which is what we used in Figure 7.3. For example, in audio signal processing, we can assume that the signal volume is 0 before the recording starts and after it ends. In this case, the calculation of P[1] is as follows.

```
P[1]  = 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4]
      = 0 * 3 + 1*4 + 2*5 + 3*4 + 4*3
      = 38
```

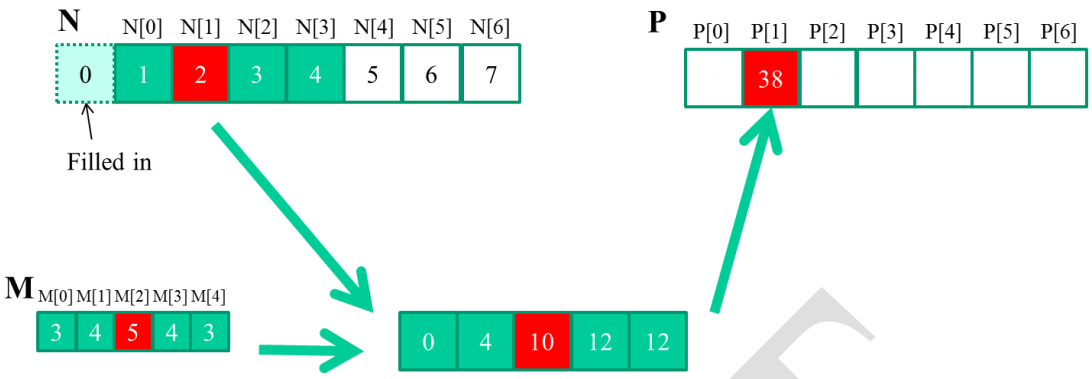


Figure 7.3 1D convolution boundary condition.

The N element that does not exist in this calculation is illustrated as a dashed box in Figure 7.3. It should be clear that the calculation of P[0] will involve two missing N elements, both will be assumed to be 0 for this example. We leave the calculation of P[0] as an exercise. These missing elements are typically referred to as "ghost cells" or "halo cells" in literature. There are also other types of ghost cells due to the use of tiling in parallel computation. These ghost cells can have significant impact on the effectiveness and/or efficiency of tiling. We will come back to this point soon.

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

For image processing and computer vision, input data is typically two-dimensional arrays, with pixels in an x-y space. Image convolutions are therefore 2D convolution, as illustrated in Figure 7.4. In a 2D convolution, the mask M is a 2D array. Its x- and y-dimensions determine the range of neighbors to be included in the weighted sum calculation. In Figure 7.4, we use a 5×5 mask for simplicity. In general, the mask does not have to be a square array. To generate an output element, we take the sub-array whose center is at the corresponding location in the input array N. We then perform pairwise multiplication between elements of the mask array and those of the image array. For our example, the result is shown as the 5×5 product array below N and P in Figure 7.4. The value of the output element is the sum of all elements of the product array.

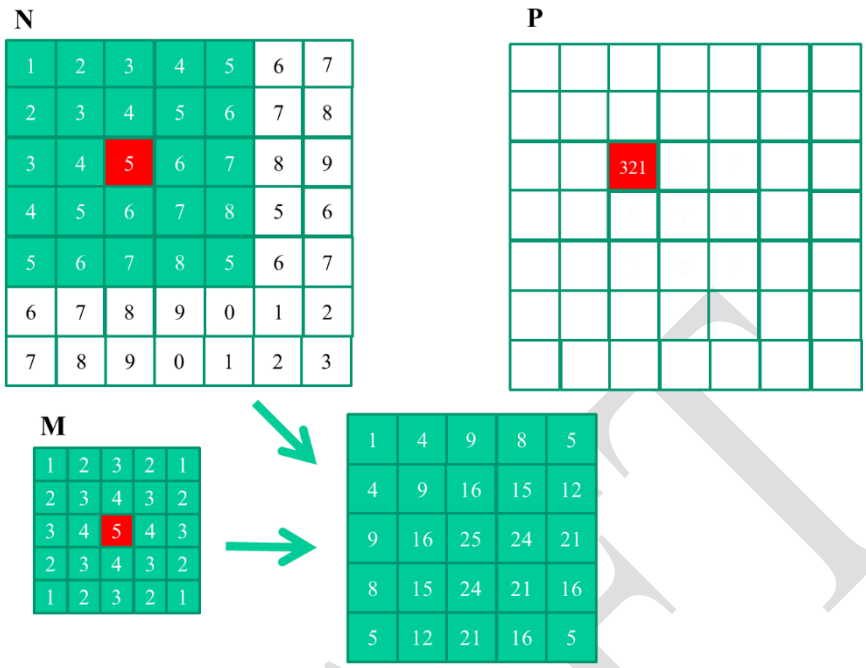


Figure 7.4 A 2D convolution example.

The example in Figure 7.4 shows the calculation of $P_{2,2}$. For brevity, we will use $N_{y,x}$ to denote N[y][x] in addressing a C array. Since N and P are most likely dynamically allocated arrays, we will be using linearized indices in our actual code examples. The sub-array of N for calculating the value of $P_{2,2}$ span from $N_{0,0}$ to $N_{0,4}$ in the x or horizontal direction and $N_{0,0}$ to $N_{4,0}$ in the y or vertical direction. The calculation is as follows:

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

Like 1D convolution, 2D convolution must also deal with boundary conditions. With boundaries in both the x and y dimensions, there are more complex boundary conditions: the calculation of an output element may involve boundary conditions along a horizontal boundary, a vertical boundary, or both. Figure 7.5 illustrates the calculation of a P element

that involves both boundaries. From Figure 7.5, the calculation of $P_{1,0}$ involves two missing columns and one missing horizontal row in the sub-array of N. Like in 1D convolution, different applications assume different default values for these missing N elements. In our example, we assume that the default value is 0. These boundary conditions also affect the efficiency of tiling. We will come back to this point soon.

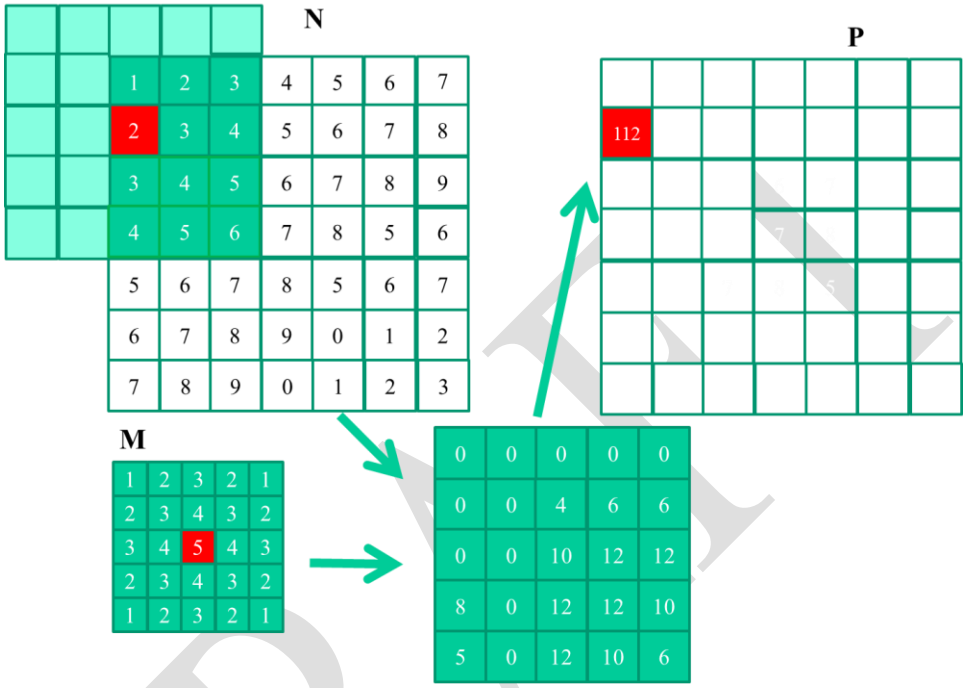


Figure 7.5 A 2D convolution boundary condition.

7.2. 1D Parallel Convolution – A Basic Algorithm

As we mentioned in Section 7.1, the calculation of all output (P) elements can be done in parallel in a convolution. This makes convolution an ideal problem for parallel computing. Based on our experience in matrix-matrix multiplication, we can quickly write a simple parallel convolution kernel. For simplicity, we will start with 1D convolution.

The first step is to define the major input parameters for the kernel. We assume that the 1D convolution kernel receives five arguments: pointer to input array N, pointer to input mask M, pointer to output array P, size of the mask Mask_Width, and size of the input and output arrays Width. Thus, we have the following set up.

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {
  // kernel body
}
```

The second step is to determine and implement the mapping of threads to output elements. Since the output array is 1D, a simple and good approach is to organize the threads into a 1D grid and have each thread in the grid to calculate one output element. The reader should recognize that this is the same arrangement as the vector addition example as far as output elements are concerned. Therefore, we can use the following statement to calculate an output element index from the block index, block dimension, and thread index for each thread:

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Once we determined the output element index, we can access the input N elements and the mask M elements using offsets to the output element index. For simplicity, we assume that Mask_Width is an odd number and the convolution is symmetric, i.e., Mask_Width is 2*n+1 where n is an integer. The calculation of P[i] will use N[i-n], N[i-n+1],…, N[i-1], N[i], N[i+1], N[i+n-1], N[i+n]. We can use a simple loop to do this calculation in the kernel:

```
float Pvalue = 0;
int N_start_point = i - (Mask_Width/2);
for (int j = 0; j < Mask_Width; j++) {
  if (N_start_point + j >= 0 && N_start_point + j < Width) {
    Pvalue += N[N_start_point + j]*M[j];
  }
}
P[i] = Pvalue;
```

The variable Pvalue will allow all intermediate results be accumulated in a register to save DRAM bandwidth. The for-loop accumulates all the contributions from the neighboring elements to the output P element. The if statement in the loop tests in any of the input N elements used are ghost cells, either on the left side or the right side of the N array. Since we assume that 0 values will be used for ghost cells, we can simply skip the multiplication and accumulation of the ghost cell element and its corresponding N element. After the end of the loop, we release the Pvalue into the output P element. We now have a simple kernel in Figure 7.6.

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;
}
```

Figure 7.6 A 1D convolution kernel with boundary condition handling.

We can make two observations about the kernel in Figure 7.6. First, there will be control flow divergence. The threads that calculate the output P elements near the left end or the right end of the P array will handle ghost cells. As we showed in Section 7.1, each of these neighboring threads will encounter a different number of ghost cells. Therefore, they will all be somewhat different decisions in the if statement. The thread that calculates P[0] will skip the multiply-accumulate statement about half of the time whereas the one that calculates P[1] will skip one fewer times, and so on. The cost of control divergence will depend on Width the size of the input array and Mask_Width the size of the masks. For large input arrays and small masks, the control divergence only occurs to a small portion of the output elements, which will keep the effect of control divergence small. Since convolution is often applied to large images and spatial data, we typically expect that the effect of convergence to be modest or insignificant.

A more serious problem is memory bandwidth. The ratio of floating-point arithmetic calculation to global memory accesses is only about 1.0 in the kernel. As we have seen in the matrix-matrix multiplication example, this simple kernel can only be expected to run at a small fraction of the peak performance. We will discuss two key techniques for reducing the number of global memory accesses in the next two sections.

7.3. Constant Memory and Caching

There are three interesting properties of the way the mask array M is used in convolution. First, the size of the M array is typically small. Most convolution masks are less than 10 elements in each dimension. Even in the case of a 3D convolution, the mask typically contains only less than 1000 elements. Second, the contents of M are not changed throughout the execution of the kernel. Third, all threads need to access the mask elements. Even better, all threads access the M elements in the same order, starting from M[0] and move by one element a time through the iterations of the for loop in Figure 7.6. These two properties make the mask array an excellent candidate for constant memory and caching (Figure 7.7).

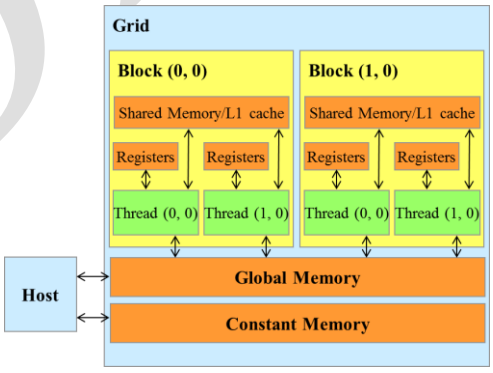


Figure 7.7 A review of the CUDA Memory Model

As we discussed in Chapter 5, the CUDA programming model allows programmers to declare a variable in the constant memory. Like global memory variables, constant memory variable are also visible to all thread blocks. The main difference is that a constant memory variable cannot be changed by threads during kernel execution. Furthermore, the size of the constant memory is quite small, currently at 64KB.

In order to use constant memory, the host code needs to allocate and copy constant memory variables in a different way than global memory variables. To declare an M array in constant memory, the host code declares it as a global variable as follows:

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

This is a global variable declaration and should be outside any function in the source file. The keyword __constant__ (two underscores on each side) tells the compiler that array M should be placed into the device constant memory.

Assume that the host code has already allocated and initialized the mask in a mask M_h array in the host memory with Mask_Width elements. The contents of the M_h can be transferred to M in the device constant memory as follows:

```
cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

Note that this is a special memory copy function that informs the CUDA runtime that the data being copied into the constant memory will not be changed during kernel execution. In general, the use of cudaMemcpyToSymble() function is as follows:

```
cudaMemcpyToSymbol(dest, src, size)
```

where dest is a pointer to the destination location in the constant memory, src is a pointer to the source data in the host memory, and size is the number of bytes to be copied.

Kernel functions access constant memory variables as global variables. Thus, their pointers do not need to be passed to the kernel as parameters. We can revise our kernel to use the constant memory as shown in Figure 7.8. Note that the kernel looks almost identical to that in Figure 7.6. The only difference is that M is no longer accessed through a pointer passed in as a parameter. It is now accessed as a global variable declared by the host code. Keep in mind that all the C language scoping rules for global variables apply here. If the host code and kernel code are in different files, the kernel code file must include the relevant external declaration information to ensure that the declaration of M is visible to the kernel.

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int
Mask_Width,
  int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;
}
```

Figure 7.8 A 1D convolution kernel using constant memory for M.

Like global memory variables, constant memory variables are also located in DRAM. However, because the CUDA runtime knows that constant memory variables are not modified during kernel execution, it directs the hardware to aggressively cache the constant memory variables during kernel execution. In order to understand the benefit of constant memory usage, we need to first understand more about modern processor memory and cache hierarchies.

As we discussed in Chapter 5, Performance considerations, the long latency and limited bandwidth of DRAM has been a major bottleneck in virtually all modern processors. In order to mitigate the effect of memory bottleneck, modern processors commonly employ on-chip cache memories, or caches, to reduce the number of variables that need to be accessed from the main memory (DRAM), as shown in Figure 7.9.

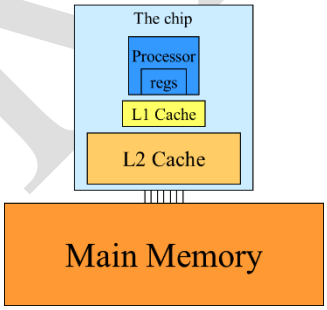


Figure 7.9 A simplified view of the cache hierarchy of modern processors.

Unlike CUDA shared memory, or scratch memories in general, caches are “transparent” to programs. That is, in order to use CUDA shared memory, a program needs to declare variables as __shared__ and explicitly move a global memory variable into a shared memory variable. On the other hand, when using caches, the program simply accesses the original variables. The processor hardware will automatically retain some of the most recently or frequently used variables in the cache and remember their original DRAM address. When one of the retained variables is used later, the hardware will detect from their addresses that

a copy of the variable is available in cache. The value of the variable will then be provided from the cache, eliminating the need to access DRAM.

There is a tradeoff between the size of a memory and the speed of a memory. As a result, modern processors often employ multiple levels of caches. The numbering convention for these cache levels reflects the distance to the processor. The lowest level, L1 or Level 1, is the cache that is directly attached to a processor core. It runs at a speed very close to the processor in both latency and bandwidth. However, an L1 cache is small in size, typically between 16KB and 64KB. L2 caches are larger, in the range of 128KB to 1MB, but can take tens of cycles to access. They are typically shared among multiple processor cores, or SMs in a CUDA device. In some high-end processors today, there are even L3 caches that can be of several MB in size.

A major design issue with using caches in a massively parallel processor is cache coherence, which arises when one or more processor cores modify cached data. Since L1 caches are typically directly attached to only one of the processor cores, changes in its contents are not easily observed by other processor cores. This causes a problem if the modified variable is shared among threads running on different processor cores. A *cache coherence mechanism* is needed to ensure that the contents of the caches of the other processor cores are updated. Cache coherence is difficult and expensive to provide in massively parallel processors. However, their presence typically simplifies parallel software development. Therefore, modern CPUs typically support cache coherence among processor cores. While modern GPUs provide two levels of caches, they typically do without cache coherence to maximize hardware resources available to increase the arithmetic throughput of the processor.

Constant memory variables play an interesting role in using caches in massively parallel processors. Since they are not changed during kernel execution, there is no cache coherence issue during the execution of a kernel. Therefore, the hardware can aggressively cache the constant variable values in L1 caches. Furthermore, the design of caches in these processors is typically optimized to broadcast a value to a large number of threads. As a result, when all threads in a warp access the same constant memory variable, as is the case of `M`, the caches can provide tremendous amount of bandwidth to satisfy the data needs of threads. Also, since the size of `M` is typically small, we can assume that all `M` elements are effectively always accessed from caches. Therefore, we can simply assume that no DRAM bandwidth is spent on `M` accesses. With the use of constant memory and caching, we have effectively doubled the ratio of floating-point arithmetic to memory access to 2.

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

7.4. Tiled 1D Convolution with Halo Cells

We will now address the memory bandwidth issue in accessing `N` array element with a tiled convolution algorithm. Recall that in a tiled algorithm, threads collaborate to load input

elements into an on-chip memory and then access the on-chip memory for their subsequent use of these elements. For simplicity, we will continue to assume that each thread calculates one output P element. With up to 1024 threads in a block we can process up to 1024 data elements. We will refer to the collection of output elements processed by each block as an *output tile*. Figure 7.10 shows a small example of 16-element 1D convolution using four thread blocks of four threads each. In this example, there are four output tiles. The first output tile covers N[0] through N[3], the second tile N[4] through N[7], the third tile N[8] through N[11], and the fourth tile N[12] through N[15]. Keep in mind that we use four threads per block to keep the example small. In practice, there should be at least 32 threads per block for the current generation of hardware. From this point on, we will assume that M elements are in the constant memory.

We will discuss two input data tiling strategy for reducing the total number of global memory accesses. The first one is the most intuitive and involves loading all input data elements needed for calculating all output elements of a thread block into the shared memory. The number of input elements to be loaded depends on the size of the mask. For simplicity, we will continue to assume that the mask size is an odd number equal to 2*n+1. That is each output element P[i] is a weighted sum of the input element at the corresponding input element N[i], the n input elements to the left (N[i-n], … N[i-1]), and the n input elements to the right (N[i+1], … N[i+n]). Figure 7.10 shows an example where Mask_Width=5 and n=2.

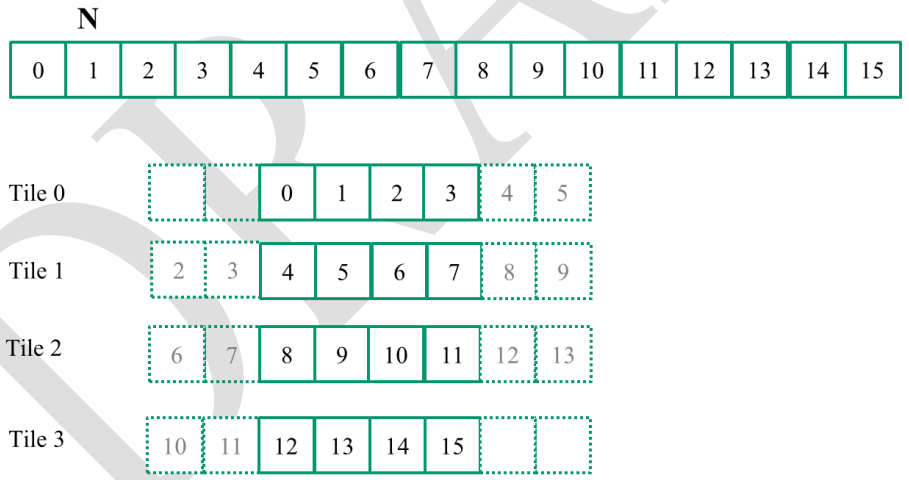


Figure 7.10 A 1D tiled convolution example.

Threads in the Block 0 calculate output elements P[0] through P[3]. They collectively require input elements N[0] through N[5]. Note that the calculation also requires two ghost cell elements to the left of N[0]. This is shown as two dashed empty elements on the left end of Tile 0 of Figure 7.6. These ghost elements will be assumed have default value of 0. Tile 3 has a similar situation at the right end of input array N. In our discussions, we will refer to

tiles like Tile 0 and Tile 3 as boundary tiles since they involve elements at or outside the boundary of the input array N.

Threads in Block 1 calculate output elements P[4] through P[7]. They collectively require input elements N[2] through N[9], also shown in Figure 7. Note that elements N[2] and N[3] belong to two tiles and are loaded into the shared memory twice, once to the shared memory of Block 0 and once to the shared memory of Block 1. Since the contents of shared memory of a block are only visible to the threads of the block, these elements need to be loaded into the respective shared memories for all involved threads to access them. The elements that are involved in multiple tiles and loaded by multiple blocks are commonly referred to as *halo cells* or *skirt cells* since they "hang" from the side of the part that is used solely by a single block. We will refer to the center part of an input tile that solely by a single block the *internal cells* of that input tile. Tile 1 and Tile 2 are commonly referred to as *internal tiles* since they do not involve any ghost elements at our outside the boundaries of the input array N.

We now show the kernel code that loads the input tile into shared memory. We first declare a shared memory array N_ds to hold the N tile for each block. The size of the shared memory array must be large enough to hold the left halo cells, the center cells, and the right halo cells of an input tile. We assume that Mask_Width is an odd number. The total is TILE_SIZE + MAX_MASK_WIDTH -1, which is used in the following declaration in the kernel.

```
__shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
```

We then load the left halo cells, which include the last n = Mask_Width/2 center elements of the previous tile. For example, in Figure 7.6, the left halo cells of Tile 1 consist of the last 2 center elements of Tile 0. In C, assuming that Mask_Width is an odd number, the expression Mask_Width/2 will result in an integer value that is the same as (Mask_Wdith-1)/2. We will use the last (Mask_Width/2) threads of the block to load the left halo element. This is done with the following two statements.

```
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
if (threadIdx.x >= blockDim.x - n) {
  N_ds[threadIdx.x - (blockDim.x - n)] =
    (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

In the first statement, we map the thread index to element index into the previous tile with the expression (blockIdx.x-1)*blockDim.x+threadIdx.x. We then pick only the last n threads to load the needed left halo elements using the condition in the if statement. For example, in Figure 7.6, blockDim.x equals 4 and n equals 2; only thread 2 and thread 3 will be used. Thread 0 and thread 1 will not load anything due to the failed condition.

For the threads used, we also need to check if their halo cells are actually ghost cells. This can be checked by testing if the calculated halo_index_left value is negative. If so, the halo cells are actually ghost cells since their N indices are negative, outside the valid range of the N indices. The conditional

C assignment will choose 0 for threads in this situation. Otherwise, the conditional statement will use the halo_index_left to load the appropriate N elements into the shared memory. The shared memory index calculation is such that left halo cells will be loaded into the shared memory array starting at element 0. For example, in Figure 7.6, blockDim.x-n equals 2. So for block 1, thread 2 will load the left most halo element into N_ds[0] and thread 3 will load the next halo element into N_ds[1]. However, for block 0, both thread 2 and thread 3 will load value 0 into N_ds[0] and N_ds[1].

The next step is to load the center cells of the input tile. This is done by mapping the blockIdx.x and threadIdx.x values into the appropriate N indices, as shown in the following statement. The reader should be familiar with the N index expression used.

```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
```

Since the first n elements of the N_ds array already contain the left halo cells, the center elements need to be loaded into the next section of N_ds. This is done by adding n to threadIdx.x as the index for each thread to write its loaded center element into N_ds.

We now load the right halo elements, which is quite similar to loading the left halo. We first map the blockIdx.x and threadIdx.x to the elements of next output tile. This is done by adding (blockIdx.x+1)*blockDim.x to the thread index to form the N index for the right halo cells. In this case, we are loading the beginning n elements of the next tile.

```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
if (threadIdx.x < n) {
  N_ds[n + blockDim.x + threadIdx.x] =
    (halo_index_right >= Width) ? 0 : N[halo_index_right];
}
```

Now that all the input tile elements are in N_ds, each thread can calculate their output P element value using the N_ds elements. Each thread will use a different section of the N_ds. Thread 0 will use N_ds[0] through N_ds[Mask_Width-1]; thread 1 will use N_ds[1] through N[Mask_Width]. In general, each thread will use N_ds[threadIdx.x] through N[threadIdx.x+Mask_Width-1]. This is implemented in the following for loop for calculate the P element assigned to the thread:

```
float Pvalue = 0;
for(int j = 0; j < Mask_Width; j++) {
  Pvalue += N_ds[threadIdx.x + j]*M[j];
}
P[i] = Pvalue;
```

However, one must not forget to do a barrier synchronization using __syncthreads() to make sure that all threads in the same block have completed loading their assigned N elements before anyone should start using them from the shared memory.

```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int
Mask_Width,
  int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
  int n = Mask_Width/2;
  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }
  __syncthreads();
  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;
}
```

Figure 7.11 A tiled 1D convolution kernel using constant memory for M.

Note that the code for multiply and accumulate is simpler than the base algorithm. The conditional statements for loading the left and right halo cells have placed the 0 values into the appropriate N_ds elements for the first and last thread block.

The tiled 1D convolution kernel is significantly longer and more complex than the basic kernel. We introduced the additional complexity in order to reduce the number of DRAM accesses for the N elements. The goal is improve the arithmetic to memory access ratio so that the achieved performance is not limited or less limited by the DRAM bandwidth. We will evaluate improvement by comparing the number of DRAM accesses performed by each thread block for the kernels in Figure 7.8 and Figure 7.11.

In Figure 7.8, there are two cases. For thread blocks that do not handle ghost cells, the number of N elements accessed by each thread is Mask_Width. Thus, the total number of N elements accessed by each thread block is blockDim.x*Mask_Width or blockDim.x*(2n+1). For example, if Mask_Width is equal to 5 and each block contains 1024 threads, each block access a total of 5120 N elements.

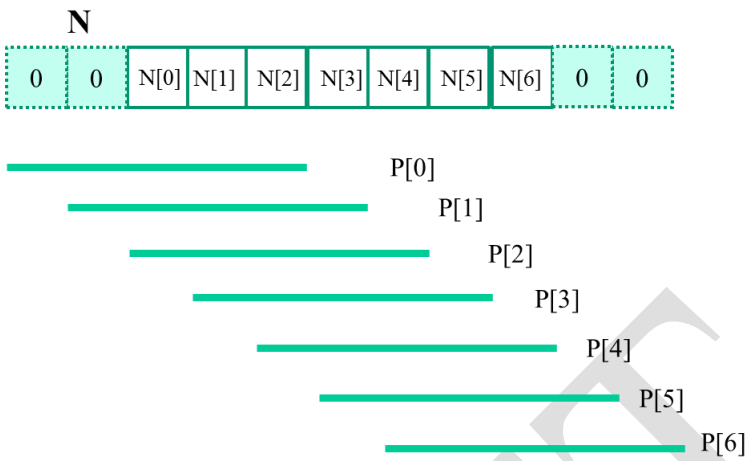


Figure 7.12 A small example of accessing N elements and ghost cells.

For the first and the last blocks, the threads that that handle ghost cells do not perform any memory access for the ghost cells. This reduces the number of memory accesses. We can calculate the reduced number of memory accesses by enumerating the number of threads that use each ghost cell. This is illustrated with a small example in Figure 7.12. The leftmost ghost cell is used by one thread. The second left ghost cell is used by two threads. In general, the number of ghost cells is n and the number of threads that use each of these ghost cells, from left to right is 1, 2, … n. This is a simple series with sum n(n+1)/2, which is the total number of accesses that were avoided due to ghost cells. For our simple example where Mask_Width is equal to 5 and n is equal to 2, the number of accesses avoided due to ghost cells is 2*3/2 = 3. A similar analysis gives the same results for the right ghost cells. It should be clear that for large thread blocks, the effect of ghost cells for small mask sizes will be insignificant.

We now calculate the total number of memory accesses for N elements by the tiled kernel in Figure 7.11. All the memory accesses have been shifted to the code that loads the N elements into the shared memory. In the tiled kernel, each N element is only loaded by one thread. However, 2n halo cells will also be loaded, n from the left and n from the right, for blocks that do not handle ghost cells. Therefore, we have the blockDim.x+2n elements loaded for the internal thread blocks and blockDim+n for boundary thread blocks.

For internal thread blocks, the ratio of memory accesses between the basic and the tiled 1D convolution kernel is:

```
(blockDim.x*(2n+1)) / (blockDim.x+2n)
```

whereas the ratio for boundary blocks is:

```
(blockDim.x*(2n+1) – n(n+1)/2) / (blockDim.x+n)
```


For most situations, blockDim.x is much larger than n. Both ratios can be approximated by eliminating the small terms n(n+1)/2 and n:

```
(blockDim.x*(2n+1)/ blockDim.x = 2n+1 = Mask_Width
```

This should be quite an intuitive result. In the original algorithm, each N element is redundantly loaded by approximately Mask_Width threads. For example, in Figure 7.12, N[2] is loaded by the 5 threads that calculate P[2], P[3], P[4], P[5], and P[6]. That is, the ratio of memory access reduction is approximately proportional to the mask size.

However, in practice, the effect of the smaller terms may be significant and cannot be ignored. For example, if blockDim.x is 128 and n is 5, the ratio for the internal blocks is

```
(128*11 – 10) / (128 + 10) = 1398 / 138 = 10.13
```

whereas the approximate ratio would be 11. It should be clear that as the blockDim.x becomes smaller, the ratio also becomes smaller. For example, if blockDim is 32 and n is 5, the ratio for the internal blocks becomes

```
(32*11 – 10) / (32+10) = 8.14
```

The readers should always be careful when using smaller block and tile sizes. They may result in significantly less reduction in memory accesses than expected. In practice, smaller tile sizes are often used due to insufficient amount of on-chip memory, especially for 2D and 3D convolution where the amount of on-chip memory needed grow quickly with the dimension of the tile.

7.5. A Simpler Tiled 1D Convolution - General Caching

In Figure 7.11, much of the complexity of the code has to do with loading the left and right halo cells in addition to the internal elements into the shared memory. More recent GPUs such as Fermi provide general L1 and L2 caches, where L1 is private to each streaming multiprocessor and L2 is shared among all streaming multiprocessors. This leads to an opportunity for the blocks to take advantage of the fact that their halo cells may be available in the L2 cache.

Recall that the halo cells of a block are also internal cells of a neighboring block. For example, in Figure 7.10, the halo cells N[2] and N[3] of Tile 1 are also internal elements of Tile 0. There is a significant probability that by the time Block 1 needs to use these halo cells, they are already in L2 cache due to the accesses by Block 0. As a result, the memory accesses to these halo cells may be naturally served from L2 cache without causing additional DRAM traffic. That is, we can leave the accesses to these halo cells in the original N elements rather than loading them into the N_ds. We now present a simpler tiled 1D

convolution algorithm that only loads the internal elements of each tile into the shared memory.

In the simpler tiled kernel, the shared memory N_ds array only needs to hold the internal elements of the tile. Thus, it is declared with the TILE_SIZE, rather than TILE_SIZE+Mask_Width-1.

```
__shared__ float  N_ds[TILE_SIZE];
```

Loading the tile becomes very simple with only one line of code:

```
N_ds[threadIdx.x] = N[blockIdx.x*blockDim.x+threadIdx.x];
```

We still need a barrier synchronization before using the elements in N_ds. The loop that calculates P elements, however, becomes more complex. It needs to add conditions to check for use of both halo cells and ghost cells. The handling of ghost cells are done with the same conditional statement as that in Figure 7.6. The multiply-accumulate statement becomes more complex, as shown in Figure 7.13.

```
__syncthreads();

int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);
float Pvalue = 0;
for (int j = 0; j < Mask_Width; j ++) {
  int N_index = N_start_point + j;
  if (N_index >= 0  && N_index < Width) {
    if ((N_index >= This_tile_start_point)
      && (N_index < Next_tile_start_point)) {
      Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
    } else {
      Pvalue += N[N_index] * M[j];
    }
  }
}
P[i] = Pvalue;
```

Figure 7.13 Using general caching for halo cells.

The variables This_tile_start_point and Next_tile_start_point hold the starting position index of the tile processed by the current block and that of the tile processed by the next in the next block. For example, in Figure 7.10, the value of This_tile_start_point for Block 1 is 4 and the value of Next_tile_start_point is 8.

The new if statement tests if the current access to the N element falls within tile by testing it against This_tile_start_point and Next_tile_start_point. If the element falls within the tile, that is, it is an internal element for the current block, it is accessed from the N_ds array in

the shared memory. Otherwise, it is accessed from the N array, which is hopefully in the L2 cache. The complete 1D tiled convolution kernel with caching is shown in Figure 7.14.

```
__global__ void convolution_1D_tiled_caching_kernel(float *N, float *P, int
Mask_Width,
  int Width) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE];
  N_ds[threadIdx.x] = N[i];
  __syncthreads();
  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j ++) {
     int N_index = N_start_point + j;
     if (N_index >= 0  && N_index < Width) {
       if ((N_index >= This_tile_start_point)
         && (N_index < Next_tile_start_point)) {
         Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
       } else {
         Pvalue += N[N_index] * M[j];
       }
     }
  }
  P[i] = Pvalue;
}
```

Figure 7.14 A simpler tiled 1D convolution kernel using constant memory and general caching.

7.6. Tiled 2D Convolution with Halo Cells

Now that we have learned how to tile a parallel 1D convolution computation, we can extend our knowledge to 2D quite easily. For a little more fun, we will use an example based on a class of 2D image format that is frequently encountered in image libraries and applications.

As we have seen in Chapter 3, real-world images are represented as 2D matrices and come in all sizes and shapes. Image processing libraries typically store these images in row-major layout when reading them from files into memory. If the width of the image in terms of bytes is not a multiple of the DRAM burst size, the starting point of row 1 and beyond can be misaligned from the DRAM burst boundaries. As we have seen in Chapter 5, such misalignment can result in poor utilization of DRAM bandwidth when we attempt to access data in one of the rows. As a result, image libraries often also convert images into a padded format when reading them from files into memory, as illustrated in Figure 7.15.

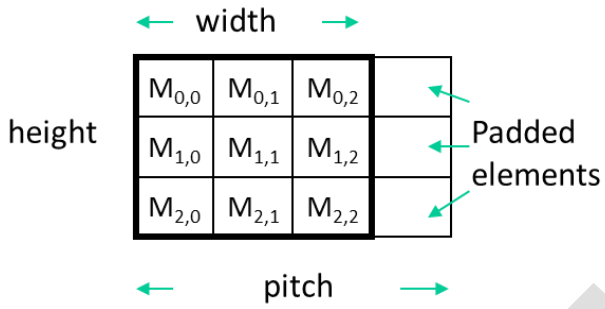


Figure 7.15 A padded image format and the concept of pitch.

In Figure 7.15, we assume that the original image is 3x3. We future assume that each DRAM burst encompass 4 pixels. Without padding, $M_{1,0}$ in row 1 would reside in one DRAM burst unit whereas $M_{1,1}$ and $M_{1,2}$ would reside in the next DRAM burst unit. Accessing row 1 would require two DRAM bursts and wasting half of the memory bandwidth. To address this inefficiency, the library pads one element at the end of each row. With the padded elements, each row occupies an entire DRAM burst size. When we access row 1 or row 2, the entire row can now be accessed in one DRAM burst. In general, the images are much larger; each row can encompass multiple DRAM bursts. The padded elements will be added such that each row ends at the DRAM burst boundaries.

With padding, the image matrix has been enlarged by the padded elements. However, during computation such as image blur (Chapter 3), one should not process the padded elements. Therefore, the library data structure will indicate the original width and height of the image as shown in Figure 7.15. However, the library also has to provide the users with the information about the padded elements so that the user code can properly find the actual starting position of all the rows. This information is conveyed as the *pitch* of the padded matrix.

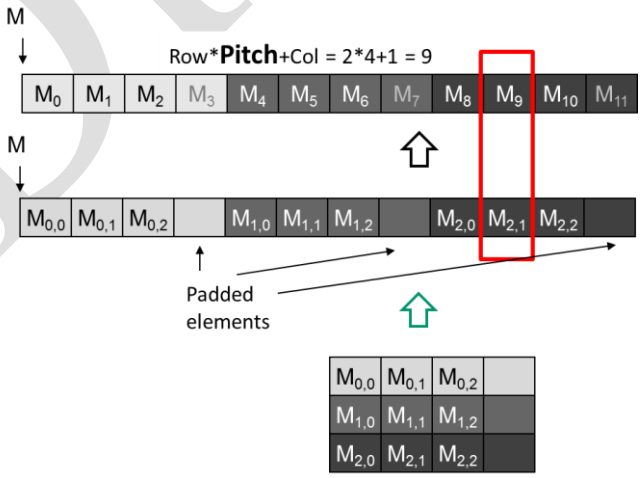


Figure 7.16 Row-major layout of a 2D image matrix with padded elements

Figure 7.16 shows how the image pixel elements can be accessed in the row-major layout of the padded image matrix. The lower layout shows the linearized order. Note that the padded elements are at the end of each row. The top layout shows the linearized 1D index of pixel elements in the padded matrix. As before, the three original elements, $M_{0,1}$, $M_{0,2}$, $M_{0,3}$ of row 0 become M_0, M_1, and M_2 in the linearized 1D array., Note that the padded elements become "dummy" linearized elements M_3, M_7, and M_{11}. The original elements of row 1, $M_{1,1}$, $M_{1,2}$, $M_{1,3}$, have their linearized 1D index as M_4, M_5, and M_6. That is, as shown in the top of Figure 7.16, to calculate the linearized 1D index of the pixel elements, we will use pitch instead of width in in the expression:

Linearized 1D index = row * pitch + column

However, when we iterate through a row, we will use width as the loop bound to ensure that we use only the original elements in a computation.

```
// Image Matrix Structure declaration
//
typedef struct {
   int width;
   int height;
   int pitch;
   int channels;
   float* data;
} * wbImage_t;
```

Figure 7.17 The C type structure definition of the image pixel element.

Figure 7.17 shows the image type that we will be using for the kernel code example. Note the channels field indicates the number of channels in the pixel: 3 for an RGB color image and 1 for a greyscale image as we have seen in Chapter 2. We assume that the value of these fields will be used as arguments when we invoke the 2D convolution kernel.

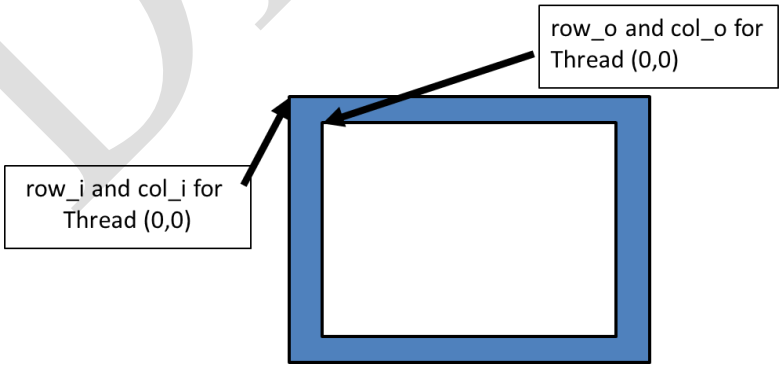


Figure 7.18 starting element indices of the input tile vs. output tile.

We are now ready to work on the design of a tiled 2D convolution kernel. In general, we will find that the design of the 2D convolution kernel is a straightforward extension of the

1D convolution kernel presented in Section 7.5. We need to first design the input and output tiles to be processed by each thread block, as shown in Figure 7.18. Note that the input tiles must include the halo cells and extend beyond their corresponding output tiles by the number of halo cells in each direction. Figure 7.19 shows the first part of the kernel:

```
__global__ void convolution_2D_kernel(float *P, float *N, int height, int width,
                        int pitch, int channels, int Mask_Width)
{
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row_o = blockIdx.y*O_TILE_WIDTH + ty;
  int col_o   = blockIdx.x*O_TILE_WIDTH + tx;

  int row_i = row_o – Mask_Width/2;
  int col_i   = col_o – Mask_Width/2;
```

Figure 7.19 Part 1 of a 2D convolution kernel.

Each thread of the kernel first calculates the y and x indices of its output element. These are the col_o and row_o variables of the kernel. The index values for thread$_{0,0}$ of the thread block (who is responsible for output the element at the upper left corner) is shown in Figure 7.18. Each thread then calculates the y and x indices of the input element it is to load into the shared memory by subtracting (Mask_Width/2) form row_o and col_o and assigning the results to row_i and col_i, also shown in Figure 7.18. Note that the input tile element to be loaded by thread$_{0,0}$ is also shown in Figure 7.18. To simply the tiling code over the kernel in Figure 7.14, we will configure each thread block to be of the same size as the input tile. In this design, we can simply have each thread to load one input N element. We will turn off some of the threads when we calculate the output since there are more threads in each block than the number of data element in each output tile.

```
__shared__ float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1][TILE_SIZE+MAX_MASK_HEIGHT-1];


If ((row_i >= 0) && (row_i < height) && (col_i >= 0) && (col_i < width)) {
   N_ds[ty][tx] = data[row_i * pitch + col_i];
} else{
   N_ds[ty][tx] = 0.0f;
}
```

Figure 7.20 Part 2 of a 2D convolution kernel.

We are now ready to load the input tiles into the shared memory. All threads participate in this activity but each of them needs to check if the y and x indices of its input tile elements are within the valid range of the input. If not, the input element it is attempting to load is

actually a ghost element and a 0.0 value should be placed into the N_ds element in shared memory. These threads belong in the thread blocks that calculate the image tiles that are close to the edge of the image. Note that we use the pitch value when we compute the linearized 1D index from the y and x index of the pixel. Also note that this code only works for the case where the number of channels is 1. In general, we should use a for-loop to load all the pixel channel values based on the number of channels present.

```
float output = 0.0f;
if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
   for(i = 0; i < MASK_WIDTH; i++) {
      for(j = 0; j < MASK_WIDTH; j++) {
         output += M[i][j] * N_ds[i+ty][j+tx];
      }
    }
  if(row_o < height && col_o < width)  {
    data[row_o*width + col_o] = output;
  }
}
```

Figure 7.21 Part 3 of a 2D convolution kernel.

The last part of the kernel, shown in Figure 7.21, computes the output value using the input elements in the shared memory. Keep in mind that we have more threads in the thread block than the number of pixels in the output tile. The if-statement ensures that only the threads whose indices fall are both smaller than the O_TILE_WIDTH should participate in the calculation of output pixels. The doubly nested for-loop iterates through the mask array and performs the multiply and accumulate operation on the mask element values and input pixel values. Since the input tile in the shared memory N_ds includes all the halo elements, the index expressions N_ds[i+ty][j+tx] gives the Ns element that should be multiplied with M[i][j]. The reader should notice that this is a straightforward extension of the index expression in corresponding for-loop in Figure 7.11. Finally, all threads whose output elements are in the valid range write their result values into their respective output elements.

To assess the benefit of the 2D tiled kernel over a basic kernel, we can also extend the analysis from 1D convolution. In a basic kernel, every thread in a thread block will perform $(\text{Mask\_Width})^2$ accesses to the image array. Thus, each thread block performs a total of $(\text{Mask\_Width})^2*(\text{O\_TILE\_WIDTH})^2$ accesses to the image array.

In the tiled kernel, all threads in a thread block collectively load one input tile. Therefore, the total number of accesses by a thread block to the image array is $(\text{O\_TILE\_WIDTH+Mask\_Width-1})^2$. That is, the ratio of image array accesses between the basic and the tiled 2D convolution kernel is:

$$(\text{Mask\_Width})^2*(\text{O\_TILE\_WIDTH})^2 / (\text{O\_TILE\_WIDTH+Mask\_Width-1})^2$$

The larger the ratio is, the more effective the tiled algorithm is in reducing the number of memory accesses as compared to the basic algorithm.

TiILE_WIDTH	8	16	32	64
Reduction Mask_Width = 5	11.1	16	19.7	22.1
Reduction Mask_Width = 9	20.3	36	51.8	64

Figure 7.22 Image array access reduction ratio for different tile sizes.

Figure 7.22 shows the trend of the image array access reduction ratio as we vary O_TILE_WIDTH, the output tile size. As O_TILE_WIDTH becomes very large, the size of the mask becomes negligible compared to tile size. Thus, each input element loaded will be used about $(\text{Mask\_Width})^2$ times. For Mask_Width value of 5, we expect that the ratio will approach 25 as the O_TILE_WIDTH becomes much larger than 5. For example, for O_TILE_WIDTH = 64, the ratio is 22.1. This is significantly higher than the ratio of 11.1 for O_TILE_WIDTH = 8. The important takeaway point is that we must have a sufficiently large O_TILE_WIDTH in order for the tiled kernel to deliver its potential benefit. The cost of a large O_TILE_WIDTH is the amount of shared memory needed to hold the input tiles.

For a larger Mask_Width, such as 9 in the bottom row of Figure 7.22, the ideal ratio should be $9^2 = 81$. However, even with a large O_TILE_WIDTH such as 64, the ratio is only 64. Note that O_TILE_WIDTH=64 and Mask_Width=9 translate into input tile size of $72^2=5184$ elements or 20,736 bytes assuming single precision data. This is more than the about the amount of available shared memory in each SM of the current generation of GPUs. Stencil computation that is derived from finite difference methods for solving differential equation often require a Mask_Width of 9 or above to achieve numerical stability. Such stencil computation can benefit from larger amount of shared memory in future generations of GPUs.

7.7. Summary

In this chapter, we have studied convolution as an important parallel computation pattern. While convolution is used in many applications such as computer vision and video processing, it is also represents a general pattern that forms the basis of many parallel algorithms. For example one can view the stencil algorithms in partial differential equation (PDE) solvers as a special case of convolution. For another example, one can also view the calculation of grid point force or potential value as a special case of convolution.

We have presented a basic parallel convolution algorithm whose implementations will be limited by DRAM bandwidth for accessing both the input N and mask M elements. We then

introduced the constant memory and a simple modification to the kernel and host code to take advantage of constant caching and eliminate practically all DRAM accesses for the mask elements. We further introduced a tiled parallel convolution algorithm that reduces DRAM bandwidth consumption by introducing more control flow divergence and programming complexity. Finally we presented a simpler tiled parallel convolution algorithm that takes advantage of the L2 caches.

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

7.8. Exercises

1. Calculate the P[0] value in Figure 7.3.

2. Consider performing a 1D convolution on array N={4,1,3,2,3} with mask M={2,1,4}. What is the resulting output array?

3. What do you think the following 1D convolution masks are doing?

   (a) [0 1 0]
   (b) [0 0 1]
   (c) [1 0 0]
   (d) [-1/2 0 1/2]
   (e) [1/3 1/3 1/3]

4. Consider performing a 1D convolution on an array of size n with a mask of size m:

   (a) How many halo cells are there in total?
   (b) How many multiplications are performed if halo cells are treated as multiplications (by 0)?
   (c) How many multiplications are performed if halo cells are not treated as multiplications?

5. Consider performing a 2D convolution on a square matrix of size nxn with a square mask of size mxm:

(a) How many halo cells are there in total?
(b) How many multiplications are performed if halo cells are treated as multiplications (by 0)?
(c) How many multiplications are performed if halo cells are not treated as multiplications?

6. Consider performing a 2D convolution on a rectangular matrix of size n1xn2 with a rectangular mask of size m1xm2:

(a) How many halo cells are there in total?
(b) How many multiplications are performed if halo cells are treated as multiplications (by 0)?
(c) How many multiplications are performed if halo cells are not treated as multiplications?

7. Consider performing a 1D tiled convolution with the kernel shown in Figure 7.11 on an array of size n with a mask of size m using a tiles of size t.

(a) How many blocks are needed?
(b) How many threads per block are needed?
(c) How much shared memory is needed in total?
(d) Repeat the same questions if you were using the kernel in Figure 7.13.

8. Revise the 1D kernel in Figure 7.6 to perform 2D convolution. Add more width parameters to the kernel declaration as needed.

9. Revise the 1D kernel in Figure 7.8 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory.

10. Revise the tiled 1D kernel in Figure 7.11 to perform 2D convolution. Keep in mind that the host code also needs to be changed to declare a 2D M array in the constant memory. Pay special attention to the increased usage of shared memory. Also, the N_ds needs to be declared as a 2D shared memory array.

Chapter 8

Parallel Patterns: Prefix Sum

An introduction to work efficiency in parallel algorithms

With special contributions by Li-Wen Chang and Juan Gómez-Luna

Keywords: scan, prefix sum, reduction, work efficiency, linear recurrence, Kogge-Stone, Brent-Kung, hierarchical algorithms, adjacent synchronization, stream-based scan

CHAPTER OUTLINE

Our next parallel pattern is prefix sum, which is also commonly known as scan. Parallel scan is frequently used to convert seemingly sequential operations, such as resource allocation, work assignment, and polynomial evaluation into parallel operations. In general, if a computation is naturally described as a mathematical recursion, it can likely be parallelized as a parallel scan operation. Parallel scan plays a key role in massively parallel computing for a simple reason: any sequential section of an application can drastically limit the overall performance of the application. Many such sequential sections can be converted into parallel computation with parallel scan. Another reason why parallel scan is an important parallel pattern is that sequential scan algorithms are linear algorithms and are extremely work-efficient, which makes it also very important to control the work efficiency of parallel scan algorithms. As we will show, a slight increase in algorithm complexity can make parallel scan run slower than sequential scan for large data sets. Therefore, work-efficient parallel

scan also represents an important class of parallel algorithms that can run effectively on parallel systems with a wide range of available computing resources.

8.1. Background

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

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

For example, If $\oplus$ is addition, then an inclusive scan operation on the input array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22, 25].

We can illustrate the applications for inclusive scan operations using an example of cutting sausage for a group of people. Assume that we have a 40-inch sausage to be served to 8 people. Each person has ordered a different amount in terms of inches: 3, 1, 7, 0, 4, 1, 6, 3. That is, person number 0 wants 3 inches of sausage, person number 1 wants 1 inch, and so on. We can cut the sausage either sequentially or in parallel. The sequential way is very straightforward. We first cut a 3-inch section for person number 0. The sausage is now 37 inches long. We then cut a 1-inch section for person number 1. The sausage becomes 36 inches long. We can continue to cut more sections until we serve the 3-inch section to person number 7. At that point, we have served a total of 25 inches of sausage, with 15 inches remaining.

With an inclusive scan operation, we can calculate the locations of all the cutting points based on the amount each person orders. That is, given an addition operation and an order input array [3 1 7 0 4 1 6 3], the inclusive scan operation returns [3 4 11 11 15 16 22 25]. The numbers in the return array are cutting locations. With this information, one can simultaneously make all the eight cuts that will generate the sections that each person ordered. The first cut point is at the 3-inch location so the first section will be 3 inches, as ordered by person number 0. The second cut point is at the 4-inch location, therefore the second section will be 1-inch long, as ordered by person number 1. The final cut point will be at the 25-inch location, which will produce a 3-inch long section since the previous cut point is at 22-inch point. This gives person number 7 what she ordered. Note that since all the cut points are known from the scan operation, all cuts can be done in parallel.

In summary, an intuitive way of thinking about inclusive scan is that the operation takes an order from a group of people and identifies all the cut points that allow the orders to be served all at once. The order could be for sausage, bread, camp ground space, or a contiguous chunk of memory in a computer. As long as we can quickly calculate all the cut points, all orders can be served in parallel.

An exclusive scan operation is similar to an inclusive operation with the exception that it returns the output array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].$$

That is, the first output element is 0 while the last output element only reflects the contribution of up to x_{n-2}.

The applications of an exclusive scan operation are pretty much the same as those for inclusive scan. The inclusive scan provides slightly different information. In the sausage example, exclusive scan would return [0 3 4 11 11 15 16 22], which are the beginning points of the cut sections. For example, the section for person number 0 starts at the 0-inch point. For another example, the section for person number 7 starts at the 22-inch point. The beginning point information is important in applications such as memory allocation, where the allocated memory is returned to the requester via a pointer to its beginning point.

Note that it is fairly easy to convert between the inclusive scan output and the exclusive scan output. One simply needs to shift all elements and fill in an element. When converting from inclusive to exclusive, one can simply shift all elements to the right and fill in value 0 for the 0^{th} element. When converting from exclusive to inclusive, we need to shift all elements to the left and fill in the last element with the previous last element plus the last input element. It is just a matter of convenience that we can directly generate an inclusive or exclusive scan, whether we care about the cut points or the beginning points for the sections.

In practice, parallel scan is often used as a primitive operation in parallel algorithms that perform radix sort, quick sort, string comparison, polynomial evaluation, solving recurrences, tree operations, stream compaction and histograms.

Before we present parallel scan algorithms and their implementations, we would like to first show a work-efficient sequential inclusive scan algorithm and its implementation. We will assume that the operation involved is addition. The algorithm assumes that the input elements are in the x array and the output elements are to be written into the y array.

```
void sequential_scan(float *x, float *y, int Max_i) {
  int accumulator = x[0];
  y[0] = accumulator;
  for (int i = 1; i < Max_i; i++) {
     accumulator += x[i];
     y[i] = accumulator;
  }
}
```

The algorithm is work-efficient, only a small amount of work is done for each input or output element. With a reasonably good compiler, only one addition, one memory load, and one memory store are used in processing each input x element. This is pretty much the minimal that we will ever be able to do. As we will see, when the sequential algorithm of a

computation is so "lean and mean," it is extremely challenging to develop a parallel algorithm that will consistently beat the sequential algorithm when the data set size becomes large.

8.2. A Simple Parallel Scan

We start with a simple parallel inclusive scan algorithm by doing a reduction operation for each output element. The main idea is to create each element quickly by calculating a reduction tree of the relevant input elements for each output element. There are multiple ways to design the reduction tree for each output element. We will first present one that is based on the Kogge-Stone algorithm that was originally invented for designing fast adder circuits in the 1970s [KS1973]. This algorithm is still being used in the design of high-speed computer arithmetic hardware today.

The algorithm, shown is Figure 8.1, is an in-place scan algorithm that operates on an array XY that originally contains input elements. It then iteratively evolves the contents of the array into output elements. Before the algorithm begins, we assume of XY[i] contains input element x_i. At the end of iteration n, XY[i] will contain the sum of the up to 2^n input elements at and before the location. That is, at the end of iteration 1, XY[i] will contain $x_{i-1}+x_i$ and at the end of iteration 2, XY[i] will contain $x_{i-3}+x_{i-2}+x_{i-1}+x_i$, and so on.

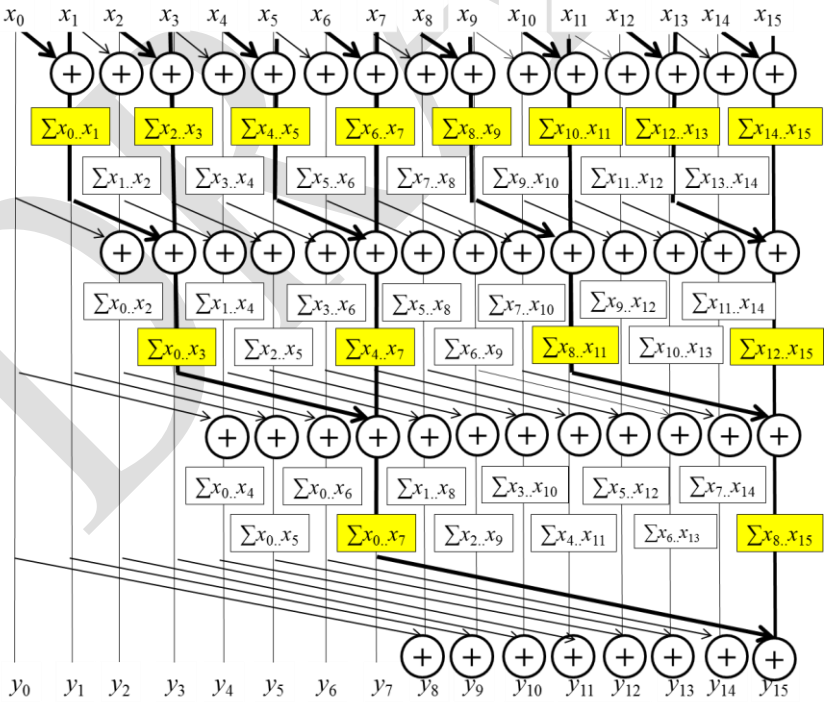


Figure 8.1: *A parallel inclusive scan algorithm based on Kogge-Stone adder design.*

Figure 8.1 illustrates the algorithm with a 16-element input example. Each vertical line represents an element of the XY array, with XY[0] in the leftmost position. The vertical direction shows the progress of iterations, starting from the top of the figure. For inclusive scan, by definition, y_0 is x_0 so XY[0] contains its final answer. In the first iteration, each position other than XY[0] receives the sum of its current content and that of its left neighbor. This is illustrated by the first row of addition operators in Figure 8.1. As a result, XY[i] contains $x_{i-1}+x_i$. This is reflected in the labeling boxes under the first row of addition operators in Figure 8.1. For example, after the first iteration, XY[3] contains x_2+x_3, shown as $\sum x_2..x_3$. Note that after the first iteration, XY[1] is equal to x_0+x_1, which is the final answer for this position. So, there should be no further changes to XY[1] in subsequent iterations.

In the second iteration, each position other than XY[0] and XY[1] receives the sum of its current content and that of the position that is two elements away. This is illustrated in the labeling boxes below the second row of addition operators. As a result, XY[i] now contains $x_{i-3}+x_{i-2}+x_{i-1}+x_i$. For example, after the first iteration, XY[3] contains $x_0+x_1+x_2+x_3$, shown as $\sum x_0..x_3$. Note that after the second iteration, XY[2] and XY[3] contain their final answers and will not need to be changed in subsequent iterations.

The reader is encouraged to work through the rest of the iterations. We now work on the parallel implementation of the algorithm illustrated in Figure 8.1. We assign each thread to evolve the contents of one XY element. We will write a kernel that performs scan on **one section** of the input that is small enough for a block to handle. The size of a section is defined as a compile-time constant SECTION_SIZE. We assume that the kernel launch will use SECTION_SIZE as the block size so there will be the same number of threads and section elements. Each thread will be responsible for calculating one output element.

All results will be calculated as if the array only has the elements in the section. **Later, we will make final adjustments to these sectional scan results for large input arrays.** We also assume that input values were originally in a global memory array X, whose address is passed into the kernel as an argument. We will have all the threads in the block to collaboratively load the X array elements into a shared memory array XY. This is done by having each thread to calculate its global data index `i = blockIdx.x*blockDim.x + threadIdx.x` for the output vector element position it is responsible for. Each thread loads the input element at that position into the shared memory at the beginning of the kernel. At the end of the kernel, each thread will write its result into the assigned output array Y.

```
__global__ void Kogge_Stone_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) {
     XY[threadIdx.x] = X[i];

  }

  // the code below performs iterative scan on XY
```

```
    ...

    Y[i] = XY[threadIdx.x];

  }
```

We now focus on the implementation of the iterative calculations for each XY element in Figure 8.1 as a for-loop:

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

The loop iterates through the reduction tree for the XY array position that is assigned to a thread. Note that we use a barrier synchronization to make sure that all threads have finished their previous iteration of additions in the reduction tree before any of them starts the next iteration. This is the same use of __syncthreads() as in the reduction discussion in Chapter 5. When the stride value becomes greater than a thread's threadIdx.x value, it means that the thread's assigned XY position has already accumulated all the required input values.

The execution behavior of the for-loop is consistent with the example shown in Figure 8.1. The actions of the smaller positions of XY end earlier than the larger positions (see the if-statement condition). This will cause some level of control divergence in the first warp when stride values are small. Note that adjacent threads will tend to execute the same number of iterations. The effect of divergence should be quite modest for large block sizes since divergence will only arise in the first warp. The detailed analysis is left as an exercise. The final kernel is shown in Figure 8.2.

```
    __global__ void Kogge_Stone_scan_kernel(float *X, float *Y,
      int InputSize) {

      __shared__ float XY[SECTION_SIZE];
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if (i < InputSize) {
        XY[threadIdx.x] = X[i];
      }

      // the code below performs iterative scan on XY
      for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (threadIdx.x >= stride) XY[threadIdx.x] += XY[threadIdx.x-stride];
      }
      Y[i] = XY[threadIdx.x];

    }
```

Figure 8.2 A Kogge-Stone kernel for inclusive scan

We can easily convert an inclusive scan kernel to an exclusive scan kernel. Recall that an exclusive scan is equivalent to an inclusive scan with all elements shifted to the right by one

position and element 0 filled with value 0. This is illustrated in Figure 8.3. Note that the only real difference is the alignment of elements on top of the picture. All labeling boxes are updated to reflect the new alignment. All iterative operations remain the same.

We can now easily convert the kernel in Figure 8.2 into an exclusive scan kernel. The only modification we need to do is to load 0 into XY[0] and X[i-1] into XY[threadIdx.x], as shown in the code below:

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

Note that the XY positions whose associated input elements are outside the range are now also filled with 0. This causes no harm and yet it simplifies the code slightly. We leave the work to finish the exclusive scan kernel as an exercise.

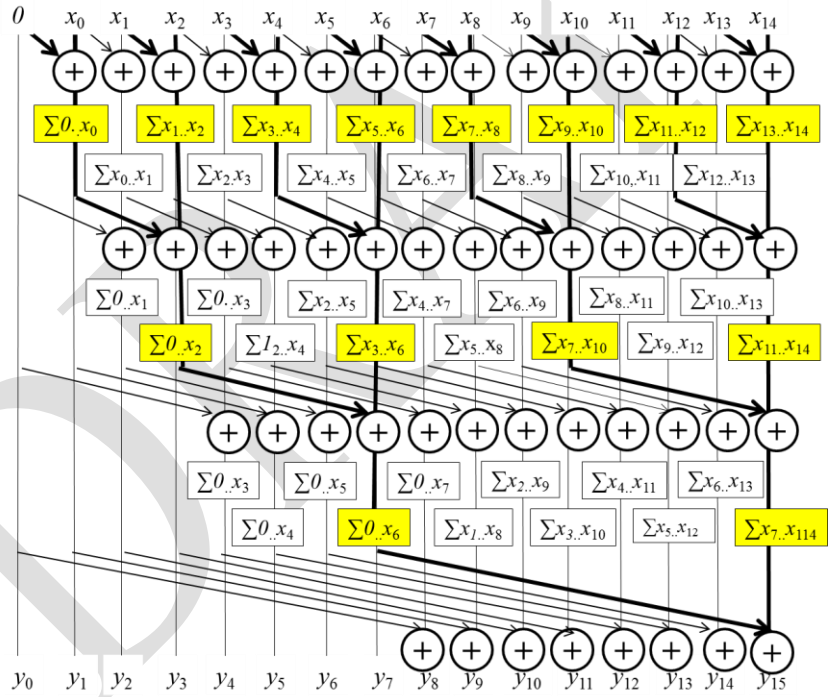


Figure 8.3 A parallel exclusive scan algorithm based on Kogge-Stone adder design.

8.3. Speed and Work Efficiency

We now analyze the speed and work efficiency of the kernel in Figure 8.2. All threads will iterate up to $\log_2 N$ steps, where N is the SECTION_SIZE. In each iteration, the number of inactive threads is equal to the stride size. Therefore, we can calculate the amount of work

done (one iteration of the for-loop, represented by the add operation in Figure 8.1) for the algorithm as

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

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

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

Recall that the number of for-loop iterations executed for a sequential scan algorithm is N-1. Note that even for modest sized sections, the kernel in Figure 8.2 does much more work than the sequential algorithm. In the case of 512 elements, the kernel does approximately 8 times more work than the sequential code. The ratio will increase as N becomes larger.

As for execution speed, the for-loop of the sequential code executes N iterations. As for the kernel code, the for-loop of each thread executes up to $\log_2 N$ iterations, which defines the minimal number of steps needed for executing the kernel. With unlimited execution resources, the speedup of the kernel code over the sequential code would be approximately $N/\log_2 N$. For N=512, the speedup would be about 512/9 = 56.9.

In a real CUDA GPU device, the amount of work done by the Kogge-Stone kernel is more than the theoretical $N*\log_2 N - (N-1)$. This is because we are using N threads. While many of the threads stop participating in the execution of the for-loop, they still consume execution resources until the entire thread block completes execution. Realistically, the amount of execution resources consumed by the Kogge-Stone is closer to $N*\log_2 N$.

We will use the concept of time units as an approximate indicator of execution time for comparing between scan algorithms. The sequential scan should take approximately N time units to process N input elements. For example, the sequential scan should take approximately 1024 time units to process 1024 input elements. With P execution units (streaming processors) in the CUDA device, we can expect the Kogge-Stone kernel to execute for $(N*\log_2 N)/P$ time units. For example, if we use 1024 threads and 32 execution units to process 1024 input elements, the kernel will likely take (1024*10)/32 = 320 time units. In this case, we expect to achieve a speedup of 1024/320 = 3.2.

The additional work done by the Kogge-Stone kernel over the sequential code is problematic in two ways. First, the use of hardware for executing the parallel kernel is much less efficient. We see that one needs to have at least 8 times more execution units in a parallel machine than the sequential machine just to break even. For example, if we execute the kernel on a parallel machine with four times the execution resources as a sequential machine, the parallel machine executing the parallel kernel can end up with only half the speed of the sequential machine executing the sequential code. Second, all the extra work consume additional

energy. This makes the kernel less appropriate for power-constrained environments such as mobile applications.

The strength of the Kogge-Stone kernel is that it can achieve very good execution speed when there is enough hardware resource. It is typically used to calculate the scan result for a section with modest number of elements, such as 32 or 64. As we have seen, its execution has very limited amount of control divergence. In newer GPU architecture generations, its computation can be efficiently performed with shuffle instructions within warps. We will see later in this chapter that it is an important component of the modern high-speed parallel scan algorithms.

8.4. A More Work-Efficient Parallel Scan

While the Kogge-Stone kernel in Figure 8.2 is conceptually simple, its work efficiency is quite low for some practical applications. Just by inspecting Figures 8.1 and 8.3, we can see that there are potential opportunities for sharing some intermediate results to streamline the operations performed. However, in order to allow more sharing across multiple threads, we need to strategically calculate the intermediate results to be shared and then quickly distribute them to different threads.

As we know, the fastest parallel way to produce sum values for a set of values is reduction tree. With sufficient execution units, a reduction tree can generate the sum for N values in $\log_2 N$ time units. Furthermore, the tree can also generate a number of sub-sums that can be used in the calculation of some of the scan output values. This observation forms the basis of the Brent-Kung adder design [BK1979], which can also be used in a parallel scan algorithm.

In Figure 8.4, we produce the sum of all 16 elements in 4 steps. We use the minimal number of operations needed to generate the sum. During the first step, only the odd element of XY[i] will be updated to XY[i-1]+XY[i]. During the second step, only the XY elements whose indices are of the form of 4*n-1, which are 3, 7, 11, 15 in Figure 8.4, will be updated. During the third step, only the XY elements whose indices are of the form 8*n-1, which are 7 and 15, will be updated. Finally, during the fourth step, only XY[15] is updated. The total number of operations performed is 8+4+2+1 = 15. In general, for a scan section of N elements, we would do (N/2)+(N/4)+…+2+1= N-1 operations for this reduction phase.

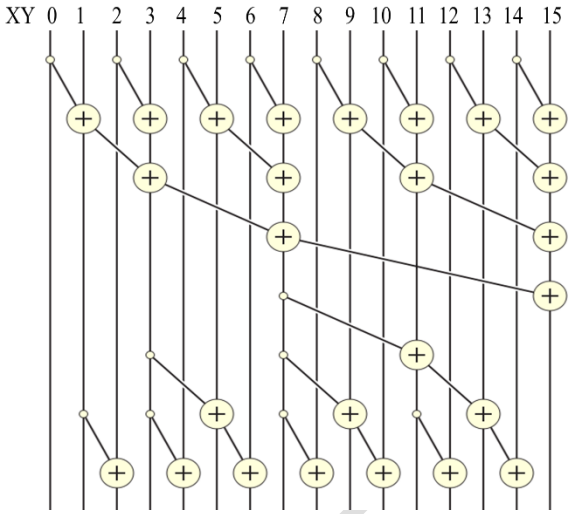


Figure 8.4: *A parallel inclusive scan algorithm based on Brent-Kung adder design.*

The second part of the algorithm is to use a reverse tree to distribute the partial sums to the positions that can use these values as quickly as possible. This is illustrated in the bottom half of Figure 8.4. At the end of the reduction phase, we have quite a few usable partial sums. For our example, the first row of Figure 8.5 shows all the partial sums in XY right after the top reduction tree. An important observation is that XY[0], XY[7] and X[15] contain their final answers. Therefore, all remaining XY elements can obtain the partial sums they need from no farther than 4 positions away.

For example, XY[14] can obtain all the partial sums it needs from XY[7], XY[11], and XY[13]. To organize our second half of the addition operations, we will first show all the operations that need partial sums from 4 positions away, then 2 positions away, then 1 position way. By inspection, XY[7] contains a critical value needed by many positions in the right half. A good way is to add XY[7] to XY[11], which brings XY[11] to the final answer. More importantly, XY[7] also becomes a good partial sum for XY[12], XY[13], and XY[14]. No other partial sums have so many uses. Therefore, there is only one addition XY[11] = XY[7]+XY[11] that needs to occur in the 4-position level in Figure 8.4. We show the updated partial sum in the second row of Figure 8.5.

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

Figure 8.5 *partial sums available in each XY element after the reduction tree phase*

We now identify all additions using partial sums that are 2 positions away. We see that XY[2] only needs the partial sum that is next to it in XY[1]. Same with XY[4]; it needs the partial sum next to it to be complete. The first XY element that can need a partial sum two positions away is XY[5]. Once we calculate XY[5] = XY[3]+XY[5], XY[5] contains the final answer. Same analysis show that XY[6] and XY[8] can become complete with the partial sums next to them in XY[5] and XY[7].

The next 2-position addition is XY[9] = XY[7]+XY[9], which makes XY[9] complete. XY[10] can wait for the next round to catch XY[9]. XY[12] only needs the XY[11], which contains its final answer after the 4-position addition. The final 2-position addition is XY[13] = XY[11]+XY[13]. The third row shows all the updated partial sums in XY[5], XY[9], and XY[13]. It is clear that now every position is either complete or can be completed when added by their left neighbor. This leads to the final row of additions in Figure 8.4, which completes the contents for all the incomplete positions XY[2], XY[4], XY[6], XY[8], XY[10], and XY[12].

We could implement the reduction tree phase of the parallel scan using the following loop:

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

Note that this loop is very similar to the reduction in Figure 5.2. The only difference is that we want the threads that have thread index in the form of 2^n-1, rather than 2^n to perform addition in each iteration. This is why we added 1 to the threadIdx.x when we select the threads for performing addition in each iteration. However, this style of reduction is known to have control divergence problems. As we have seen in Chapter 5, a better way is to use a decreasing number of contiguous threads to perform the additions as the loop advances:

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

By using a more complex index calculation in each iteration of the for-loop, the execution of the kernel now has much fewer control divergence within warps. In our example in Figure 8.4, there are 16 threads in the block. In the first iteration, stride is equal to 1. The first 8 consecutive threads in the block will satisfy the if-condition. The index values calculated for these threads will be 1, 3, 5, 7, 9, 11, 13, 15. These threads will perform the first row of additions in Figure 8.4. In the second iteration, stride is equal to 2. Only the first 4 threads

in the block will satisfy the if-condition. The index values calculated for these threads will be 3, 7, 11, 15. These threads will perform the second row of additions in Figure 8.4. Note that since each iteration will always be using consecutive threads in each iteration, the control divergence problem does not arise until the number of active threads drops below the warp size.

The distribution tree is a little more complex to implement. We make an observation that the stride value decreases from SECTION_SIZE/4 to 1. In each iteration, we need to "push" the value of XY element from a position that is a multiple of stride value minus 1 to a position that is stride away. For example, in Figure 8.4, the stride value decreases from 4 down to 1. In the first iteration in Figure 8.4, we would like to push the value of XY[7] to XY[11], where 7 is 2*4-1. Note that only thread 0 will be used for this iteration. In the second iteration, we would like to push the values of XY[3], XY[7], and XY[11] to XY[5], XY[9], and XY[13]. This can be implemented with the following loop:

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

The calculation of index is similar to that in the reduction tree phase. The final kernel code for a Brent-Kung parallel scan is shown in Figure 8.6. The reader should notice that we never need to have more than SECTION_SIZE/2 threads for either the reduction phase or the distribution phase. So, we could simply launch a kernel with SECTION_SIZE/2 threads in a block. Since we can have up to 1024 threads in a block, each scan section can have up to 2048 elements. However, we will need to have each thread to load two X elements at the beginning and store two Y elements at the end.

As was in the case of the Kogge-Stone scan kernel, one can easily adapt the Brent-Kung inclusive parallel scan kernel into an exclusive scan kernel with a minor adjustment to the statement that loads X elements into XY. Interested readers should also read [Harris2007] for an interesting natively exclusive scan kernel that is based on a different way of designing the distribution tree phase of the scan kernel.

```
__global__ void Brent_Kung_scan_kernel(float *X, float *Y,
  int InputSize) {

  __shared__ float XY[SECTION_SIZE];
  int i = 2*blockIdx.x*blockDim.x + threadIdx.x;
  if (i < InputSize) XY[threadIdx.x] = X[i];
  if (i+blockDim.x < InputSize) XY[threadIdx.x+blockDim.x] = X[i+blockDim.x];

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

```
        XY[index] += XY[index - stride];
      }
    }

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

    __syncthreads();
    if (i < InputSize) Y[i] = XY[threadIdx.x];
    if (i+blockDim.x < InputSize) Y[i+blockDim.x] = XY[threadIdx.x+blockDim.x];
  }
```

Figure 8.6 *A Brent-Kung kernel for inclusive scan*

We now turn our attention to the analysis of the number of operations in the distribution tree stage. The number of operations is (16/8)-1+(16/4)-1+(16/2)-1. In general, for N input elements, the total number of operations would be (2-1)+(4-1)+…+(N/4-1)+(N/2-1) which is N-1-$\log_2$(N). The total number of operations in the parallel scan, including both the reduction tree (N-1 operations) and the inverse reduction tree phases (N-1-$\log_2$(N) operations), is 2*N-2-$\log_2$(N). Note that the upper bound of the number of operation is now proportional to N, rather than N*$\log_2$(N).

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

While the Brent-Kung algorithm has a much higher level of theoretical work-efficiency than Kogge-Stone, its advantage in a CUDA kernel implementation is more limited. Recall that the Brent-Kung algorithm is using N/2 threads. The major difference is that the number of active threads drops much faster through the reduction tree than the Kogge-Stone algorithm. However, the inactive threads still consume execution resources in a CUDA device. As a result, the amount of resources consumed by Brent-Kung kernel is actually closer to (N/2)*(2*$\log_2$(N) -1). This makes the work-efficiency of Brent-Kung about the same of that of Kogge-Stone in a CUDA device. Using the example in Section 8.4, if we process 1024 input elements with 32 execution units, the Brent-Kung kernel is expected to take approximately 512*(2*10-1)/32 = 304 time units. This results in a speedup of 1024/304 = 3.4.

8.5. An Even More Work-Efficient Parallel Scan

We can design a parallel scan algorithm that achieves even better work efficiency than Brent-Kung by adding a phase of fully independent scans on sub-sections of the input. At the

beginning of the algorithm, we partition the section of input into sub-sections. The number of sub-sections is the same as the number of threads in a thread block, one for each thread. During the first phase, we have each thread to perform a scan on its sub-section. For example, in Figure 8.7, we assume that there are four threads in a block. We partition the input section into four sub-sections. During the first phase, thread 0 will perform scan on its section (2, 1, 3, 1) and generate (2, 3, 6, 7). Thread 1 will perform scan on its section (0, 4, 1, 2) and generate (0, 4, 5, 7), etc.

Note that if each thread directly performs scan by accessing the input from global memory, their accesses would not be coalesced. For example, during the first iteration, thread 0 would be accessing input element 0, thread 1 input element 4, etc. Therefore, we use the **corner turning** technique presented in Chapter 4, to improve memory coalescing. At the beginning of the phase, all threads collaborate to load the input into the shared memory in an iterative manner. In each iteration, adjacent threads load adjacent elements to enable memory coalescing. For example, in Figure 8.7, we will have all threads to collaborate and load four elements in a coalesced manner: thread 0 to load element 0, thread 1 element 1, etc. All threads then move to load the next four elements: thread 0 to load element 4, thread 1 element 5, etc.

Once all input elements are in the shared memory, the threads access their own sub-section from the shared memory. This is shown as Step 1 in Figure 8.7. Note that at the end of Step 1, the last element of each section (highlighted as black in the second row) contains the sum of all input elements in the section. For example the last element of section 0 contains value 7, the sum of the input elements (2, 1, 3, 1) in the section.

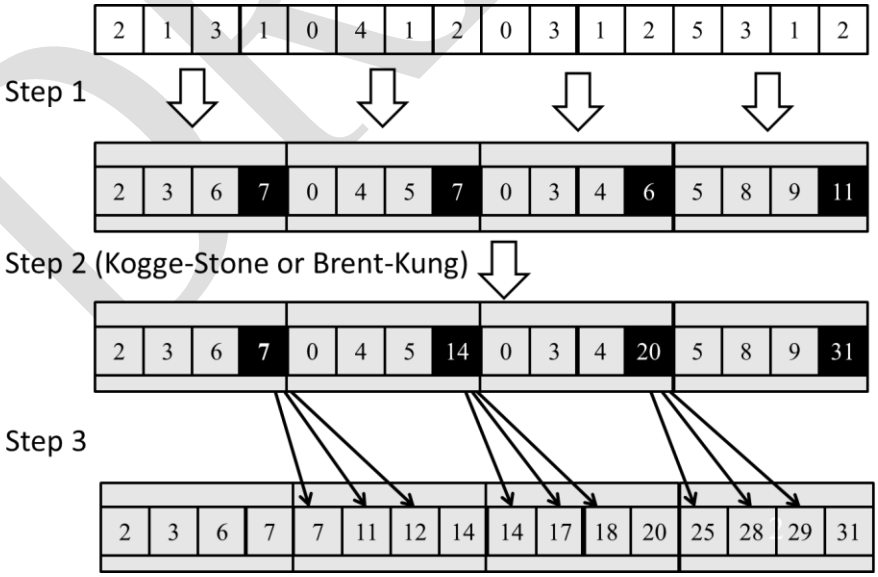


Figure 8.7 A two-phase parallel scan for higher work efficiency and speed

During the second phase, all threads in each block collaborate and perform a scan operation on a logical array that consists of last elements of all sections. This can be done with a Kogge-Stone or Brent-Kung algorithm since there are only a modest number (number of threads in a block) of elements. In Step 3, each thread adds the new value of the last element of its predecessor's section to its elements. The last elements of each sub-section do not need to be updated during this phase. For example, in Figure 8.7, thread 1 adds the value 7 to elements (0, 4, 5) in its section to produce (7, 11, 12). Note that the last element of the section is already the correct value 14 and does not need to be updated.

Note that with this three-phase approach, we can use a much smaller number of threads than the number of elements in a section. The maximal size of the section is no longer limited by the number of threads one can have in the block but rather the size of the shared memory; all elements of the section need to fit into the shared memory. This limitation will be removed in the hierarchical methods to be discussed in the remainder of this chapter.

The major advantage of the three-phase approach is the efficiency of its use of execution resources. Assume that we use Kogge-Stone for phase 2. For an input list of N elements, if we use T threads, the amount of work done by each phase is N-1 for phase 1, $T*\log_2 T$ for phase 2, and N-T for phase 3. If we use P execution units, we can expect that the execution will take $(N-1 + T*\log_2 T + N-T) / P$ time units.

For example, if we use 64 threads and 32 execution units to process 1024 elements, the algorithm should take approximately (1024-1 + 64*6 + 1024-64) / 32 = 74 time units. This results in a speedup of 1024/74 = 13.8.

8.6. Hierarchical Parallel Scan for Arbitrary-Length Inputs

For many applications, the number of elements to be processed by a scan operation can be in the millions or even billions. The three kernels that we presented so far assume that the entire input can be loaded in the shared memory. Obviously, we cannot expect that all input elements of these large scan applications can fit into the shared memory. This is the reason we say that they process a section of the input. Furthermore, it would be a loss of parallelism opportunity if we used only one thread block to process these large data sets. Fortunately, there is a hierarchical approach to extending the scan kernels that we have generated so far to handle inputs of arbitrary size. The approach is illustrated in Figure 8.8.

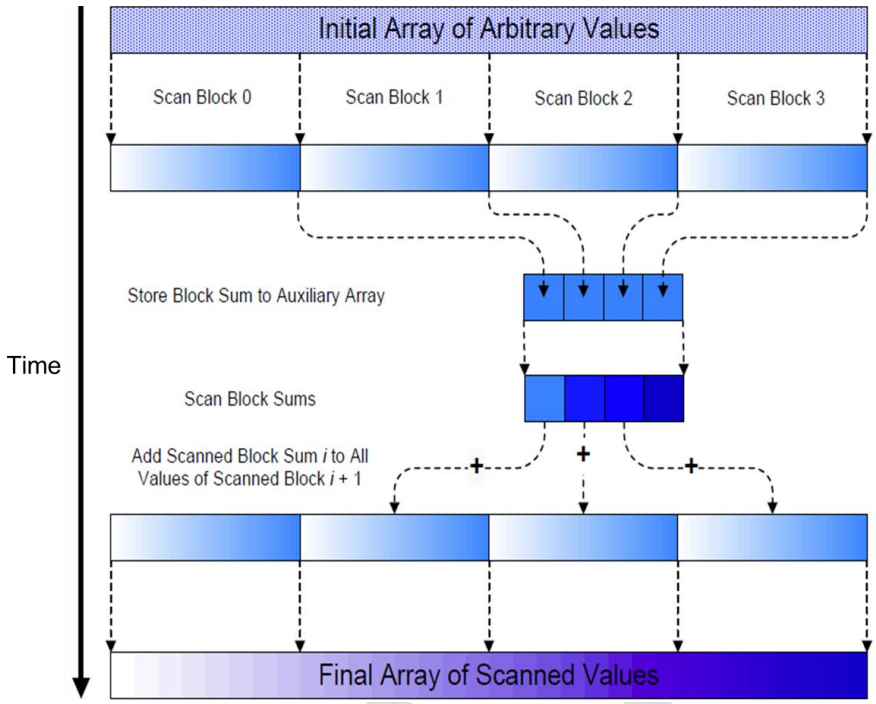


Figure 8.8 *A hierarchical scan for arbitrary length inputs*

For a large data set, we first partition the input into sections so that each of them can fit into the shared memory and processed by a single block. For the current generation of CUDA devices, the Brent-Kung kernel in Figure 8.8 can process up to 2048 elements in each section using 1024 threads in each block. For example, if the input data consist of 2,000,000 elements, we can use ceil(2,000,000/2048.0) = 977 thread blocks. With up to 65,536 thread blocks in the x-dimension of a grid, this approach can process up to 134,217,728 elements in the input set. If the input is even bigger than this, we can use additional levels of hierarchy to handle truly arbitrary number of input elements. However, for this chapter, we will restrict our discussion to a two-level hierarchy that can process up to 134,217,728 elements.

Assume that we launch one of the three kernels in Sections 8.2, 8.4 and 8.5 on a large input data set. At the end of the grid execution, the `Y` array will contain the scan results for individual sections, called *scan blocks* in Figure 8.8. Each result value in a scan block only contains the accumulated values of all preceding elements in the same scan block. These scan blocks need to be combined into the final result. That is, we need to write and launch another kernel that adds the sum of all elements in preceding scan blocks to each element of a scan block.

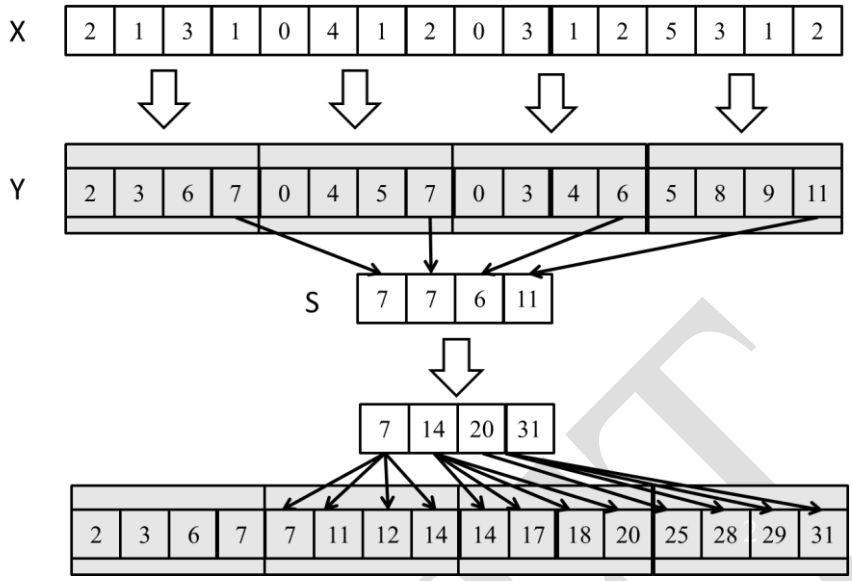


Figure 8.9: *An example of hierarchical scan.*

Figure 8.9 shows a small example of the hierarchical scan approach of Figure 8.8. In this example, there are 16 input elements that are divided into 4 scan blocks. We can use the Kogge-Stone kernel, the Brent-Kung kernel, or the three-phase kernel to process the individual scan blocks. The kernel treats the four scan blocks as independent input data sets. After the scan kernel terminates, each `Y` element contains the scan result within its scan block. For example, scan block 1 has inputs 0, 4, 1, 2. The scan kernel produces the scan result for this section, 0, 4, 5, 7. Note that these results do not contain the contributions from any of the elements in scan block 0. In order to produce the final result for this scan block, the sum of all elements in scan block 0, 2+1+3+1=7, should be added to every result element of scan block 1.

For another example, the inputs in scan block 2 are 0, 3, 1, 2. The kernel produces the scan result for this scan block, 0, 3, 4, 6. In order to produce the final results for this scan block, the sum of all elements in both scan block 0 and scan block 1, 2+1+3+1+0+4+1+2=14, should be added to every result element of scan block 2.

It is important to note that the last output element of each scan block gives the sum of all input elements of the scan block. These values are 7, 7, 6, and 11 in Figure 8.9. This brings us to the second step of the hierarchical scan algorithm in Figure 8.8, which gathers the last result elements from each scan block into an array and performs a scan on these output elements. This step is also illustrated in Figure 8.9, where the last scan output elements of all collected into a new array `S`.

This can be done by changing the code at the end of the scan kernel so that the last thread of each block writes its result into an `S` array using its `blockIdx.x` as index. A scan operation is

then performed on S to produce output values 7, 14, 20, 31. Note that each of these second-level scan output values are accumulated sum from the beginning location X[0] to the end of each scan block. That is, Output value in S[0]=7 is the accumulated sum from X[0] to the end of scan block 0, which is X[3]. Output value in S[1]=14 is the accumulated sum from X[0] to the end of scan block 1, which is X[7]. [1]

Therefore, the output values in the S array give the scan results at "strategic" locations of the original scan problem. That is, in Figure 8.9, the output values in S[0], S[1], S[2], and S[3] give the final scan results for the original problem at positions X[3], X[7], X[11], and X[15]. These results can be used to bring the partial results in each scan block to their final values. This brings us to the last step of the hierarchical scan algorithm in Figure 8.8. The second-level scan output values are added to the values of their corresponding scan blocks.

For example, in Figure 8.9, the value of S[0] (value 7) will be added to Y[0], Y[1], Y[2], Y[3] of thread block 1, which completes the results in these positions. The final results in these positions are 7, 11, 12, 14. This is because S[0] contains the sum of the values of the original input X[0] through X[3]. These final results are 14, 17, 18, and 20. The value of S[1] (14) will be added to Y[8], Y[9], Y[10], Y[11], which completes the results in these positions. The value of S[2] (20) will be added to Y[12], Y[13], Y[14], Y[15]. Finally, the value of S[3] is the sum of all elements of the original input, which is also the final result in Y[15].

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

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

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

The second kernel is simply the one of the three parallel scan kernels, which takes S as input and writes S as output.

[1] While the second step of Figure 8.9 is logically the same as the second step of Figure 8.7. The main difference is that Figure 8.9 involves threads from different thread blocks. As a result, the last element of each section needs to be collected into a global memory array so that they can be visible across thread blocks.

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

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

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

8.7. Single-Pass Scan for Memory Access Efficiency

In the hierarchical scan mentioned in Section 8.6, the partial scanned results are stored into global memory before launching the global scan kernel, and then re-loaded back from the global memory by the third kernel. The latencies of these extra memory stores and loads are not overlapped with the computation in the subsequent kernels and can significantly impact the speed of the hierarchical scan algorithms. In order to avoid such negative impact, multiple techniques [DGS2008][YLZ2013][MG2016] have been proposed. In this chapter, a stream-based scan algorithm is discussed. The reader is encouraged to read the references to understand the other techniques.

In the context of CUDA C programming, a stream-based scan algorithm (not to be confused with CUDA Streams to be introduced in Chapter 18) refers to a hierarchical scan algorithm where partial sum data is passed in one direction through the global memory between neighboring thread blocks. Stream-based scan builds on a key observation that the global scan step (middle part of Figure 8.8) can be performed in a domino fashion. For example, in Figure 8.9, Scan Block 0 can pass its partial sum value 7 to Scan Block 1, and then complete its job. Scan Block 1 receives the partial sum value 7 from Scan Block 0, sums up with its local partial sum value 7 to get 14, passes its partial sum value 14 to Scan Block 2, and then completes its final step.

In a stream-based scan, one can write a single kernel to perform all the three steps of the hierarchical scan algorithm in Figure 8.8. Thread block i first performs a scan on its scan block, using one of the three parallel algorithms we presented in Sections 8.2 through 8.5. It then waits for its left neighbor block i-1 to pass the sum value. Once it receives the sum from block i-1, it generates and passes its sum value to its right neighbor block i+1. It then moves on to add the sum value received from block i-1 to finish all the output values of the scan block.

During the first phase of the kernel, all block can execute in parallel. They will be serialized during the data streaming phase. However, as soon as each block receives the sum value from its predecessor, it can perform its final phase in parallel with all other blocks that have

received the sum values from their predecessors. As long as the sum values can be passed through the blocks quickly, there can be ample of parallelism among blocks.

In order to make this stream-based scan work adjacent (block) synchronization has been proposed in [YLZ2013]. Adjacent synchronization is a customized synchronization to allow the adjacent thread blocks to synchronize and/or exchange data. Particularly, in scan, the data are passed from Scan Block i-1 to Scan Block i, like a produce-consumer chain. On the producer side (Scan Block i-1), the flag is set to a particular value after the partial sum is stored to memory, while on the consumer side (Scan Block i), the flag is checked to see if it is that particular value before the passed partial sum is loaded. As mentioned, the loaded value is further added with the local sum and then is passed to the next block (Scan Block i+1). Adjacent synchronization can be implemented using atomic operations. The following code segment illustrates the use of atomic operations to implement adjacent synchronization.

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

This code section is only executed by one leader thread in each block (e.g., thread with index 0). The rest of threads will wait in __syncthreads() in the last line. In block bid, the leader thread checks flags[bid], a global memory array, repeatedly, until it is set. Then, it loads the partial sum from its predecessor by accessing the global memory array scan_value[bid] and stores the value into its local register variable previous_sum. It sums up with its local partial sum local sum, and stores the result into the global memory array scan_value[bid+1]. The memory fence function __threadfence() is required to ensure that the partial sum is completely stored to memory before the flag is set with atomicAdd(). The array scan_value must be declared as volatile, in order to prevent the compiler from optimizing, reordering, or register allocating the accesses to the scan_value elements.

Although it may appear that the atomic operations on the flags array and the accesses to the scan_value array could incur global memory traffic, these operations are mostly performed in the second-level caches of recent GPU architectures (more details in Chapter 9). Any stores and loads to the global memory will likely be overlapped with the phase 1 and phase 3 computational activities of other blocks. On the other hand, when executing the three-kernel scan algorithm in Section 8.5, the stores to and loads from the S array elements in the

global memory are in a separate kernel and cannot be overlapped with the phase 1 and phase 3.

There is one subtle issue with stream-based algorithms. In GPUs, thread blocks may not *always* be scheduled linearly according to their blockIdx values, which means Scan Block i may be scheduled and performed after Scan Block i+1. In this situation, the execution order arranged by the scheduler might contradict the execution assumed by the adjacent synchronization code, and cause performance loss or even a dead lock. For example, the scheduler may schedule Scan Block i through Scan Block i+N before it schedules Scan Block i-1. If Scan Block i through Scan Block i+N occupy all the streaming multiprocessors, Scan Block i-1 would not be able to start execution until at least one of them finishes execution. However, all of them are waiting for the sum value from Scan Block i-1. This causes the system to deadlock.

To resolve this issue, multiple techniques [YLZ2013][GSO2012] have been proposed. Here we only discuss one particular method, dynamic block index assignment, and leave the rest as reference for readers. Dynamic block index assignment basically decouples the usage of thread block index from the built-in blockIdx.x. In scan, the particular i of the Scan Block i is not tied to the value of blockIdx.x anymore. Instead, it is calculated using the following code after the thread block is scheduled:

```
__shared__ int sbid;
if (threadIdx.x == 0)
  sbid = atomicAdd(DCounter, 1);
__syncthreads();
const int bid = sbid;
```

The leader thread increments atomically a global counter variable pointed by DCounter. The global counter stores the dynamic block index of the next block that is scheduled. The leader thread then stores the acquired dynamic block index value in a shared memory variable sbid, so that it is accessible by all threads of the block after __syncthreads(). This guarantees all Scan Blocks are scheduled linearly, and prevents a potential deadlock.

8.8. Summary

In this chapter, we studied scan as an important parallel computation pattern. Scan is used to enable parallel allocation of resources to parties whose needs are not uniform. It converts seemingly sequential recursive computation into parallel computation, which helps to reduce sequential bottlenecks in many applications. We show that a simple sequential scan algorithm performs only N additions for an input of N elements.

We first introduced a parallel Kogge-Stone scan algorithm that is fast, conceptually simple but not work-efficient. As the data set size increases, the number of execution units needed for a parallel algorithm to break even with the simple sequential algorithm also increases. For an input of 1024 elements, the parallel algorithm performs over 9 times more additions

than the sequential algorithm and requires at least 9 times more execution resources to break even with the sequential algorithm. This is why Kogge-Stone scan algorithms are typically used within modest-sized scan blocks.

We then presented a parallel Brent-Kung scan algorithm that is conceptually more complicated. Using a reduction tree phase and a distribution tree phase, the algorithm performs only 2*N-3 additions no matter how large the input data sets are. Such a work-efficient algorithm whose number of operations grows linearly with the size of the input set is often also referred to as data-scalable algorithms. Unfortunately, due to the nature of threads in a CUDA device, the resource consumption of a Brent-Kung kernel ends up very similar to that of a Kogge-Stone kernel. A three-phase scan algorithm that employees corner turning and barrier synchronization proves to be effective in addressing the work-efficiency problem.

We also presented a hierarchical approach to extending the parallel scan algorithms to handle the input sets of arbitrary sizes. Unfortunately, a straightforward, three-kernel implementation of the hierarchical scan algorithm incurs redundant global memory accesses whose latencies are not overlapped with computation. We show that one can use a stream-based hierarchical scan algorithm to enable a single-pass, single kernel implementation and improve the global memory access efficiency of the hierarchical scan algorithm. This, however, requires a carefully designed adjacent block synchronization using atomic operations, thread memory fence, and barrier synchronization. Special care also has to be given to prevent deadlocks using dynamic block index assignment.

References

[HSO2007] Mark Harris, Shubhabrata Sengupta, John D Owens, *Parallel Prefix Sum with CUDA*, GPU Gems 3, 2007. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

[DGS2008] Yuri Dotsenko, Naga K. Govindaraju, , Peter-Pike Sloan, Charles Boyd, and John Manferdelli, Fast Scan Algorithms on Graphics Processors. In Proceedings of the 22nd annual International Conference on Supercomputing (pp. 205-213), 2008.

[YLZ2013] Shengen Yan, Guoping Long, and Yunquan Zhang, StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization, PPoPP. In ACM SIGPLAN Notices (Vol. 48, No. 8, pp. 229-238), 2013.

[GSO2012] Kshitij Gupta, Jeff A. Stuart and John D. Owens. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In Innovative Parallel Computing (InPar), 2012 (pp. 1-14). IEEE.

[KS1973] Kogge, P. & Stone, H. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". IEEE Transactions on Computers, 1973, C-22, 783-791

[BK1979] Brent, R.P. and Kung, H.T., “A regular layout for parallel adders,” Technical Report, Computer Science Department, Carnegie-Mellon University, 1979

[MG2016] Merrill, D. and Garland, M. Single-pass Parallel Prefix Scan with Decoupled Look-back. Technical Report NVR2016-001, NVIDIA Research. Mar. 2016.

8.7. Exercises

1. Analyze the parallel scan kernel in Figure 8.2. Show that control divergence only occurs in the first warp of each block for stride values up to half of the warp size. That is, for warp size 32, control divergence will occur to iterations for stride values 1, 2, 4, 8, and 16.

2. For the Brent-Kung scan kernel, assume that we have 2048 elements, how many add operations will be performed in both the reduction tree phase and the inverse reduction tree phase?
   (A) (2048-1)*2
   (B) (1024-1)*2
   (C) 1024*1024
   (D) 10*1024

3. For the Kogge-Stone scan kernel based on reduction trees, assume that we have 2048 elements, which of the following gives the closest approximation on how many add operations will be performed?
   (A) (2048-1)*2
   (B) (1024-1)*2
   (C) 1024*1024
   (D) 10*1024

4. Use the algorithm in Figure 8.3 to complete an exclusive scan kernel.

5. Complete the host code and all the three kernels for the hierarchical parallel scan algorithm in Figure 8.9.

6. Analyze the hierarchical parallel scan algorithm and show that it is work efficient and the total number of additions is no more than 4*N-3.

7. Consider the following array: [4 6 7 1 2 8 5 2]. Perform a parallel inclusive prefix scan on the array using the Kogge-Stone algorithm. Report the intermediate states of the array after each step.

8. Repeat the previous problem using the work-efficient algorithm.

9. Using the two-level hierarchical scan discussed in section 8.5, what is the largest possible dataset that can be handled if computing on a:

   a) GeForce GTX 280?
   b) Tesla C2050?
   c) GeForce GTX 690?

Chapter 9

Parallel Patterns – Parallel Histogram Computation

An Introduction to atomic operations and privatization

Keywords: histogram, feature extraction, output interference, race condition, atomic operation, read-modify-write, memory bound, memory latency, memory throughput, memory coalescing, block partition, interleaved partition

CHAPTER OUTLINE

The parallel computation patterns that we have presented so far all allow the task of computing each output element to be assigned to a thread. Therefore, these patterns are amenable to the owner-computes rule, where every thread can write into their designated output element(s) without concern about interference from other threads. This chapter introduces the parallel histogram computation pattern, a frequently encountered application computing pattern where each output element can potentially be updated by all threads. As such, one must take care to coordinate among threads as

they update output elements and avoid any interference that corrupts the final results. In practice, there are many other important parallel computation patterns where output interference cannot be easily avoided. Therefore, the parallel histogram computation pattern provides an example with output interference in these patterns. We will first examine a baseline approach that uses *atomic operations* to serialize the updates to each element. This baseline approach is simple but inefficient, often resulting in disappointing execution speed. We will then present some widely-used optimization techniques, most notably privatization, to significantly enhance execution speed while preserving correctness. The cost and benefit of these techniques depend on the underlying hardware as well as the characteristics of the input data. It is therefore important for a developer to understand the key ideas of these techniques and reason about their applicability under different circumstances.

9.1. Background

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

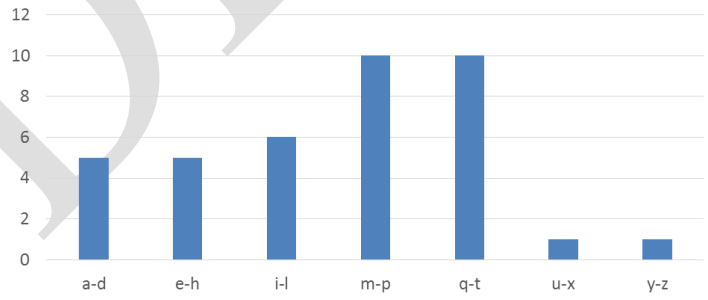


Figure 9.1 A histogram representation of "programming massively parallel processors."

Histograms provide useful summaries of data sets. In our example, we can see that the phrase being represented consists of letters that are heavily concentrated in the middle intervals of the alphabet and very light in the later intervals. Such shape of the histogram is sometimes referred to as a *feature* of the data set, and provides a quick way to determine if there are significant phenomena in the data set. For example, the shape of a histogram of the purchase categories and locations of a credit card account can be used to detect fraudulent usage. When the shape of the histogram deviates significantly from the norm, the system raises a flag of potential concern.

Many other application domains rely on histograms to summarize data sets for data analysis. One such area is computer vision. Histograms of different types of object images, such as faces vs. cars, tend to exhibit different shapes. By dividing an image into subareas and analyzing the histograms for these subareas, one can quickly identify the interesting subareas of an image that potentially contain the objects of interest. The process of computing histograms of image subareas is the basis of *feature extraction* in computer vision, where feature refers to patterns of interest in images. In practice, whenever there is a large volume of data that needs to be analyzed to distill interesting events (i.e., "Big Data"), histograms are likely used as a foundational computation. Credit card fraudulence detection and computer vision obviously meet this description. Other application domains with such needs include speech recognition, website purchase recommendations, and scientific data analysis such as correlating heavenly object movements in astrophysics.

```
1.  sequential_Histogram(char *data, int length, int *histo) {
2.      for (int i = 0; i < length; i++) {
3.          int alphabet_position = data[i] – 'a';
4.          if (alphabet_position >= 0 && alphabet_position < 26)
5.              histo[alphabet_position/4]++
6.          }
7.      }
8.  }
```

Figure 9.2 A simple C function for calculating histogram for an input text string.

Histograms can be easily computed in a sequential manner, as shown in Figure 9.2. For simplicity, the function is only required to recognize lower-case letters. The C code assumes that the input data set comes in a char array *data[]* and the histogram will be generated into the int array *histo[]* (Line 1). The number of data items is specified in function parameter *length*. The for loop (Line 2 through Line 4) sequentially traverses

the array, identifies the particular alphabet index into the *index* variable, and increments the *histo[index/4]* element associated with that interval. The calculation of the alphabet index relies on the fact that the input string is based on the standard ASCII code representation where the alphabet characters "a" through "z" are encoded in consecutive values according to the alphabet order.

Although one may not know the exact encoded value of each letter, one can assume that the encoded value of a letter is the encoded value of "a" plus the alphabet position difference between that letter and "a". In the input, each character is stored in its encoded value. Thus, the expression *data[i] – "a"* (Line 3) derives the alphabet position of the letter with the position of "a" being 0. If the position value is greater than or equal to 0 and less than 26, the data character is indeed a lower-case alphabet letter (Line 4). Keep in mind that we defined the intervals such that each interval contains four alphabet letters. Therefore, the interval index for the letter is its alphabet position value divided by 4. We use the interval index to increment the appropriate *histo[]* array element (Line 4).

The C code in Figure 9.2 is quite simple and efficient. The data array elements are accessed sequentially in the *for* loop so the CPU cache lines are well used whenever they are fetched from the system DRAM. The *histo[]* array is so small that it fits well in the level-one (L1) data cache of the CPU, which ensures very fast updates to the *histo[]* elements. For most modern CPUs, one can expect that execution speed of this code to be memory bound, i.e., limited by the rate at which the *data[]* elements can be brought from DRAM into the CPU cache.

9.2. Use of Atomic Operations

A straightforward approach to parallel histogram computation is dividing the input array into sections and have each thread to process one of the sections. If we use P threads, each thread would be doing approximately 1/P of the original work. We will refer to this approach as "Strategy I" in our discussions. Using this strategy, we should be able to expect a speedup close to P. Figure 9.3 illustrates this approach using our text example. To make the example fit in the picture, we reduce the input to the first 24 characters in the phrase. We assume that P = 4 and each thread processes a section of 6 characters. We show the workload of the four threads in Figure 9.3.

Each thread iterates through its assigned section and increment the appropriate interval counter for each character. Figure 9.3 shows the actions taken by the four threads in the

first iteration. Note that threads 0, 1, and 2 all need to update the same counter (m-p), which is a conflict referred to as output interference. One must understand the concepts of race conditions and atomic operations in order to confidently handle such output interferences in his/her parallel code.

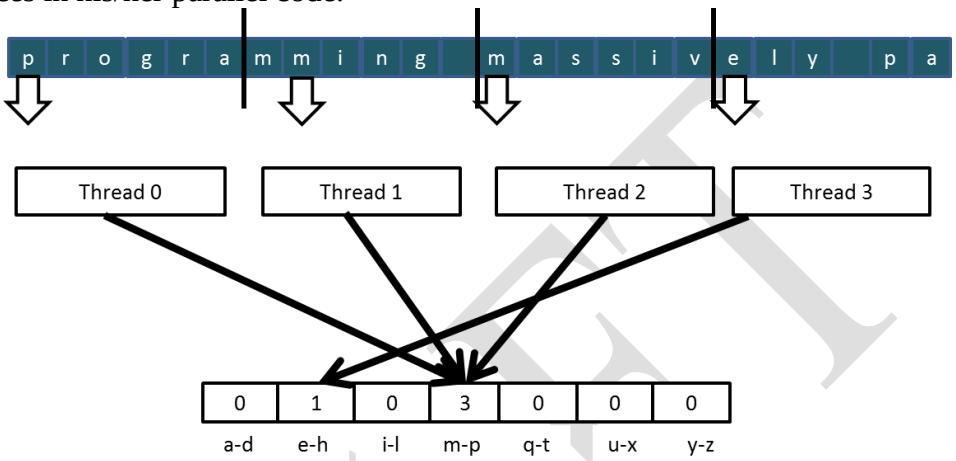


Figure 9.3 Strategy I for parallelizing histogram computation

An increment to an interval counter in the *histo[]* array is an update, or read-modify-write, operation on a memory location. The operation involves reading the memory location (read), adding one to the read content (modify), and writing the new value back to the memory location (write). Read-modify-write is a very common operation for coordinating collaborative activities.

For example, when we make a flight reservation with an airline, we bring up the seat map and look for available seats (read), we pick a seat to reserve (modify), and change the seat status to unavailable in the seat map(write). A bad potential scenario can happen as follows:

- Two customers simultaneously bring up seat map of the same flight.
- Both customers pick the same seat, say 9C.
- Both customers change the status of seat 9C to unavailable.

After the sequence, both customers thought that they have seat 9C. We can imagine that they will have an unpleasant situation when they board the flight and find out that one of them cannot take the reserved seat! Believe it or not, such unpleasant situation indeed happen in real life due to flaws in airline reservation software.

For another example, some stores allow customers to wait for service without standing in line. They ask each customer to take a number from one of the kiosks. There is a display that shows the number that will be served next. When a service agent becomes available, he/she asks the customer to present the ticket that matches the number, verify the ticket, and update the display number to the next higher number. Ideally, all customers will be served in the order they enter the store. An undesirable outcome would be that two customers simultaneously sign in at two kiosks and both receive tickets with the same number. When a service agent call for that number, both customers will feel that they are the one who should receive service.

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

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

(a)

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

(b)

Figure 9.4 Race condition in updating a *histo[]* array element.

Figure 9.4(a) depicts a scenario where Thread 1 completes all three parts of its read-modify-write sequence during time periods 1 through 3 before Thread 2 starts its sequence at time period 4. The value in the parenthesis in front of each operation shows the value being written into the destination, assuming the value of *histo[x]* was initially 0. Under this scenario, the value of *histo[x]* afterwards is 2, exactly what one would expect. That is, both threads successfully incremented the histo[x] element. The element value starts with 0 and becomes 2 after the operations complete.

In Figure 9.4(b), the read-modify-write sequences of the two threads overlap. Note that Thread 1 writes the new value into *histo[x]* at time period 4. When Thread 2 reads

histo[x] at time period 3, it still has the value 0. As a result, the new value it calculates and eventually writes to *histo[x]* is 1 rather than 2. The problem is that Thread 2 read *histo[x]* too early, before Thread 1 completes its update. The net outcome is that the value of *histo[x]* afterwards is 1, which is incorrect. The update by Thread 1 is lost.

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

(a)

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

(b)

Figure 9.5 Race condition scenarios where Thread 2 runs ahead of Thread 1.

During parallel execution, threads can run in any order relative to each other. In our example, Thread 2 can easily start its update sequence ahead of Thread 1. Figure 9.5 shows two such scenarios. In Figure 9.5(a), Thread 2 completes its update before Thread 1 starts its. In Figure 9.5(b), Thread 1 starts its update before Thread 2 completes its. It should be obvious that the sequences in 9.5(a) result in correct outcome for histo[x] but those in 9.5(b) produce incorrect outcome.

The fact that the final value of *histo[x]* varies depending on the relative timing of the operations involved indicates that there is a race condition. We can eliminate such variation by eliminating the possible interleaving of operation sequences of Thread 1 and Thread 2. That is, we would like to allow the timings shown in Figures 9.4(a) and 9.5(a) while eliminating the possibilities shown in Figures 9.4(b) and 9.5(b). This can be accomplished through the use of *atomic operations*.

An atomic operation on a memory location is an operation that performs a read-modify-write sequence on the memory location in such a way that no other read-modify-write sequence to the location can overlap with it. That is, the read, modify, and write parts of the operation form an undividable unit, thus the name atomic operation. In practice, atomic operations are realized with hardware support to lock out other operations to the same location until the current operation is complete. In our example, such support eliminates the possibility depicted in Figures 9.4(b) and 9.5(b) since the trailing thread cannot start its update sequence until the leading thread completes its.

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

Atomic operations are usually named according to the modification performed on the memory location. In our text histogram example, we are adding a value to the memory location so the atomic operation is called atomic add. Other types of atomic operations include subtraction, increment, decrement, minimum, maximum, logical and, logical or, etc.

A CUDA program can perform an atomic add operation on a memory location through a function call:

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

The function is an intrinsic function that will be compiled into a hardware atomic operation instruction which reads the 32-bit word pointed to by the address argument in global or shared memory, adds val to the old content, and stores the result back to memory at the same address. The function returns the old value of the address.

Intrinsic Functions

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

Figure 9.6 shows a CUDA kernel that performs parallel histogram computation based on Strategy I. Line 1 calculates a global thread index for each thread. Line 2 divides the total amount of data in the buffer by the total number of threads to determine the number of characters to be processed by each thread. The ceiling formula, introduced in Chapter 2, is used to ensure that all contents of the input buffer are processed. Note that the last few threads will likely process a section that is only partially filled. For example, if we have 1000 characters in the input buffer and 256 threads, we would assign sections of (1000-1)/256 +1 = 4 elements to each of the first 250 threads. The last 6 threads will process empty sections.

Line 3 calculates the starting point of the section to be processed by each thread using the global thread index calculated in Line 1. In the example above, the starting point of the section to be processed by thread i would be i * 4 since each section consists of 4 elements. That is, the starting point of thread 0 is 0, thread 8 is 32, and so on.

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
1.   int i = threadIdx.x + blockIdx.x * blockDim.x;
2.   int section_size = (size-1)  / (blockDim.x * gridDim.x) +1;
3.   int start = i*section_size;
     // All threads handle blockDim.x * gridDim.x  consecutive elements
4.   for (k = 0; k < section_size; k++) {
5.    if (start+k < size) {
6.      int alphabet_position = buffer[start+k] – ‘a’;
7.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1);
       }
     }
}
```

Figure 9.6 A CUDA kernel for calculation histogram based on Strategy I

The for-loop starting in line 4 is very similar to the one we have in Figure 9.2. This is because each thread essentially executes the sequential histogram computation on its assigned section. There are two noteworthy differences. First, the calculation of the alphabet position is guarded by an if-condition. This test ensures that only the threads whose index into the buffer is within bounds will access the buffer. It is to prevent the threads that receive partially filled or empty sections from making out-of-bound memory accesses.

Finally, the increment expression (histo[alphabet_position/4]++) in Figure 9.2 becomes an atomicAdd() function call in Line 6 of Figure 9.6. The address of the location to be updated, &(histo[alphabet_position/4]), is the first argument. The value to be added to the location, 1, is the second argument. This ensures that any simultaneous updates to any histo[] array element by different threads are properly serialized.

9.3. Block vs. Interleaved Partitioning

In Strategy I, we partition the elements of buffer[] into sections of continuous elements, or blocks, and assign each block to a thread. This partitioning strategy is often referred to as *block partitioning*. Partitioning data into continuous blocks is conceptually simple and intuitive. On a CPU, where parallel execution typically involve a small number of threads, block partitioning is often the best performing strategy since the sequential access pattern by each thread makes good use of cache lines. Since each CPU cache typically supports only a small number of threads, there is little interference in cache usage by different threads. The data in cache lines, once brought in for a thread, can be expected to remain for the subsequent accesses.

As we learned in Chapter 5, the large number of simultaneously active threads in an SM typically cause too much interference in the caches that one cannot expect a data in a cache line to remain available for all the sequential accesses by a thread under Strategy I. Rather, we need to make sure that threads in a warp access consecutive locations to enable memory coalescing. This means that we need to adjust our strategy for partitioning buffer[].

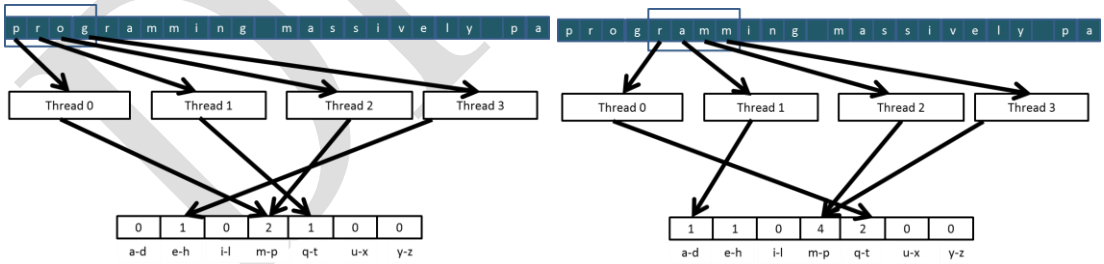


Figure 9.7 Desirable access pattern to the input buffer for memory coalescing – Strategy II.

Figure 9.7 shows the desirable access pattern for memory coalescing for our text histogram example. During the first iteration, the four threads access characters 0 through 3 ("prog"), as shown in Figure 11.7(a). With memory coalescing, all the

elements will be fetched with only one DRAM access. During the second iteration, the four threads access characters "ramm" in one coalesced memory access. Obviously, this is a toy example. In reality, there will be many more threads. There are also subtle issues such as each thread should process four characters in each iteration to fully utilize the bandwidth between the caches and the SMs.

Now that we understand the desired access pattern, we can derive the partitioning strategy to solve this problem. Instead of the block partitioning strategy, we will use an interleaved partitioning strategy where each thread will process elements that are separated by the elements processed by all threads during one iteration. In Figure 9.7, the partition to be processed by thread 0 would be elements 0 ("p"), 4 ("r"), 8 ("i"), 12 ("m"), 16 ("i"), and 20 ("y"). Thread 1 would process elements 1 ("r"), 5 ("a"), 9 ("n"), and 13 ("a"), 17 ("v"), and 21 ("_"). It should be clear why this is called interleaved partitioning: the partition to be processed by different threads are interleaved with each other.

Figure 9.8 shows a revised kernel based on Strategy II. It implements interleaved portioning in Line 2 by calculating a stride value, which is the total number threads launched during kernel invocation (blockDim.x*gridDim.x). In the first iteration of the while loop, each thread index the input buffer using its global thread index: Thread 0 accesses element 0, Thread 1 accesses element 1, Thread 2 accesses element 2, etc. Thus, all threads jointly process the first blockDim.x*gridDim.x elements of the input buffer. In the second iteration, all threads add blockDim.x*gridDim.x to their indices and jointly process the next section of blockDim.x*gridDim.x elements.

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
1.  unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
// All threads handle blockDim.x * gridDim.x consecutive elements in each iteration
2.   for (unsigned int i = tid;  i < size; i += blockDim.x*gridDim.x ) {
3.      int alphabet_position = buffer[i] – 'a';
4.      if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo[alphabet_position/4]), 1)
     }
}
```

Figure 9.8 A CUDA kernel for calculating histogram based on Strategy II.

The for-loop controls the iterations for each thread. When the index of a thread exceeds the valid range of the input buffer (i is greater than or equal to size), the thread has

completed processing its partition and will exit the loop. Since the size of the buffer may not be a multiple of the total number of threads, some of the threads may not participate in the processing of the last section. So some threads will execute one fewer for-loop iteration than others.

Thanks to the coalesced memory accesses, the version in Figure 9.8 will likely execute several times faster than that in Figure 9.6. However, there is still plenty of room for improvement, as we will show in the rest of this chapter. It is interesting that the code in Figure 9.8 is actually simpler even though interleaved partitioning is conceptually more complicated than block partitioning. This is often true in performance optimization. While an optimization may be conceptually complicated, its implementation can be quite simple.

9.4. Latency vs. Throughput of Atomic Operations

The atomic operation used in the kernels of Figures 9.6 and 9.8 ensures the correctness of updates by serializing any simultaneous updates to a location. As we all know, serializing any portion of a massively parallel program can drastically increase the execution time and reduce the execution speed of the program. Therefore, it is important that such serialized operations account for as little execution time as possible.

As we learned in Chapter 5, the access latency to data in DRAMs can take hundreds of clock cycles. In Chapter 3, we learned that GPUs use zero-cycle context switching to tolerate such latency. As long as we have many threads whose memory access latencies can overlap with each other, the execution speed is limited by the throughput of the memory system. Thus it is important that GPUs make full use DRAM bursts, banks, and channels to achieve very high memory access throughput.

It should be clear to the reader at this point that the key to high memory access throughput is the assumption that many DRAM accesses can be simultaneously in progress. Unfortunately, this assumption breaks down when many atomic operations update the same memory location. In this case, the read-modify-write sequence of a trailing thread cannot start until the read-modify-write sequence of a leading thread is complete. As shown in Figure 9.9, the execution of atomic operations to the same memory location is such that each one is the only one in progress. The duration of each atomic operation is approximately the latency of a memory read (left section of the atomic operation time) plus the latency of a memory write (right section of the atomic operation time). The length of these time sections of each read-modify-write operation,

usually hundreds of clock cycles, defines the minimal amount of time that must be dedicated to servicing each atomic operation and limits the throughput, or the rate at which atomic operations can be performed.

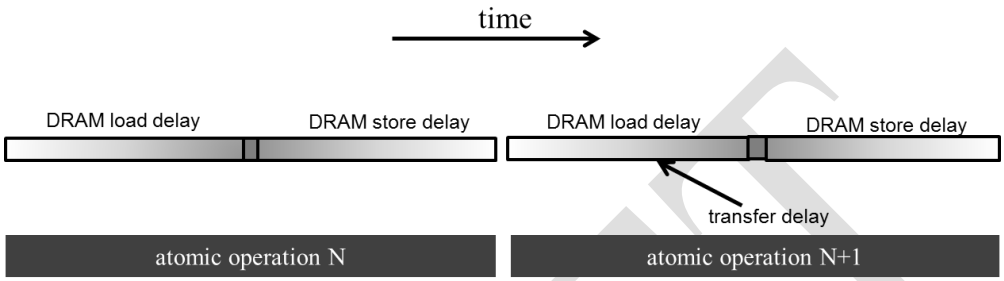


Figure 9.9 Throughput of atomic operation is determined by the memory access latency.

For example, assume a memory system with 64-bit Double Data Rate DRAM interface, 8 channels, 1 GHz clock frequency, and typical access latency of 200 cycles. The peak access throughput of the memory system is 8 (bytes/transfer)*2 (transfers per clock per channel)* 1G (clocks per second) * 8 (Channels) = 128 GB/sec. Assuming each data accessed is 4 bytes, the system has a peak access throughput of 32G data elements per second.

However, when performing atomic operations on a particular memory location, the highest throughput one can achieve is one atomic operation every 400 cycles (200 cycles for the read and 200 cycles for the write). This translates into a time-based throughput of 1/400 atomics/clock * 1G (clocks/second) = 2.5M atomics/second. This is dramatically lower than most users expect from a GPU memory system.

In practice, not all atomic operations will be performed on a single memory location. In our text histogram example, the histogram has 7 intervals. If the input characters are uniformly distributed in the alphabet, the atomic operations evenly distributed among the histo[] elements. This would boost the throughput to 7*2.5M = 17.5M atomic operations per second. In reality, the boost factor tends to be much lower than the number of intervals in the histogram because the characters tend to have biased distribution in the alphabet. For example, in Figure 9.1, we see that characters in the example phrase are heavily biased towards the m-p and q-t intervals. The heavy contention traffic to update these intervals will likely reduce the achievable throughput to much less than 17.5M atomic operations per second.

For the kernels of Figure 9.6 and Figure 9.8, low throughput of atomic operations will have significant negative impact on the execution speed. To put things into perspective, assume for simplicity that the achieved throughput of the atomic operations is 17.5M atomic operations per second. We see that the kernel in Figure 9.8 performs approximately six arithmetic operations (-, >=, <. /, +, +) with each atomic operation. Thus the maximal arithmetic execution throughput of the kernel will be 6*17.5M = 105M arithmetic operations per second. This is only a tiny fraction of the typical peak throughput of 1,000,000M or more arithmetic operations per second on modern GPUs! This type of insight has motivated several categories of optimizations to improve the speed of parallel histogram computation as well as other types of computation using atomic operations.

9.5. Atomic Operation in Cache Memory

A key insight from the previous section is that long latency of memory access translates into low throughput in executing atomic operations on heavily contended locations. With this insight, an obvious approach to improving the throughput of atomic operations is to reduce the access latency to the heavily contended locations. Cache memories are the primary tool for reducing memory access latency.

Recent GPUs allow atomic operation to be performed in the last level cache, which is shared among all SMs. During an atomic operation, if the updated variable is found in the last level cache, it is updated in the cache. If it cannot be found in the last level cache, it triggers a cache miss and is brought into the cache where it is updated. Since the variables updated by atomic operations tend to be heavily accessed by many threads, these variables tend to remain in the cache once they are brought in from DRAM. Since the access time to the last level cache is in tens of cycles rather than hundreds of cycles, the throughput of atomic operations are improved by at least an order of magnitude by just allowing them to be performed in the last level cache. This was evident in the big throughput improvement of atomic operations from the Tesla generation to the Fermi generation, where the atomic operations are first supported in the last level (L2) cache. However, the improved throughput is still insufficient for many applications.

9.6. Privatization

The latency for accessing memory can be dramatically reduced by placing data in the shared memory. Shared memory is private to each SM and has very short access latency (a few cycles). Recall that this reduced latency directly translates into increase throughput of atomic operations. The problem is that due to the private nature of shared

memory, the updates by threads in one thread block are no longer visible to threads in other blocks. The programmer must explicitly deal with this lack of visibility of histogram updates across thread blocks.

In general, a technique referred to as *privatization*, is commonly used to address the output interference problem in parallel computing. The idea is to replicate highly contended output data structures into private copies so that each thread (or each subset of threads) can update its private copy. The benefit is that the private copies can be accessed with much less contention and often at much lower latency. These private copies can dramatically increase the throughput for updating the data structures. The down side is that the private copies need to be merged into the original data structure after the computation completes. One must carefully balance between the level of contention and the merging cost. Therefore, in massively parallel systems, privatization is typically done for subsets of threads rather than individual threads.

In our text histogram example, we can create a private histogram for each thread block. Under this scheme, a few hundred threads would work on a copy of the histogram stored in short-latency shared memory, as opposed to tens of thousands of threads pounding on a histogram stored in medium latency second level cache or long latency DRAM. The combined effect of fewer contending threads and shorter access latency can result in orders of magnitude of increase in update throughput.

Figure 9.10 shows a privatized histogram kernel. Line 2 allocates a shared memory array bins_s[] whose dimension is set during kernel launch. In the for loop at Line 3, all threads in the thread block cooperatively initialize all the bins of their private copy of the histogram. The barrier synchronization in Line 5 ensures that all bins of the private histogram have been properly initialized before any thread starts to update them.

The for loop at Lines 6-7 is identical to that in Figure 9.8, except that the atomic operation is performed on the shared memory histo_s[]. The barrier synchronization in Line 8 ensures that all threads in the thread block complete their updates before merging the private copy into the original histogram.

Finally, the for loop at Lines 9-10 cooperatively merge the private histogram values into the original version. Note that atomic add operation is used to update the original histogram elements. This is because multiple thread blocks can simultaneously update the same histogram elements and must be properly serialized with atomic operations. Note that both for loops in Figure 9.10 are written so that the kernel can handle histograms of arbitrary number of bins.

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
                    unsigned int num_elements, unsigned int num_bins) {
1.  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
        // Privatized bins
2.  extern __shared__ unsigned int histo_s[];
3.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.      histo_s[binIdx] = 0u;
    }
5.  __syncthreads();
        // Histogram
6.  for(unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
      int alphabet_position = buffer[i] – “a”;
7.    if (alphabet_position >= 0 && alpha_position < 26) atomicAdd(&(histo_s[alphabet_position/4]), 1);
    }
8.  __syncthreads();
  // Commit to global memory
9.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
10.     atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```

Figure 9.10 A privatized text histogram kernel.

9.7. Aggregation

Some data sets have a large concentration of identical data values in localized areas. For example, in pictures of the sky, there can be large patches of pixels of identical value. Such high concentration of identical values causes heavy contention and reduced throughput of parallel histogram computation.

For such data sets, a simple and yet effective optimization is for each thread to aggregate consecutive updates into a single update if they are updating the same element of the histogram [Merrill2015]. Such aggregation reduces the number of atomic operations to the highly contended histogram elements, thus improving the effective throughput of the computation.

```
__global__ void histogram_privatized_kernel(unsigned char* input, unsigned int* bins,
   unsigned int num_elements, unsigned int num_bins) {
1.  unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
          // Privatized bins
2.  extern __shared__ unsigned int histo_s[];
3.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx +=blockDim.x) {
4.      histo_s[binIdx] = 0u;
    }
5.  __syncthreads();
6.  unsigned int prev_index = -1;
7.  unsigned int accumulator = 0;

8.  for(unsigned int i = tid; i < num_elements; i += blockDim.x*gridDim.x) {
9.      int alphabet_position = buffer[i] – “a”;
10.     if (alphabet_position >= 0 && alpha_position < 26) {
11.        unsigned int curr_index = alphabet_position/4;
12.        if (curr_index != prev_index) {
13.           if (accumulator >= 0) atomicAdd(&(histo_s[alphabet_position/4]), accumulator);
14.           accumulator = 1;
15.           prev_index = curr_index;
           }
16.        else {
17.           accumulator++;
           }
        }
     }
18.  __syncthreads();
          // Commit to global memory
19.  for(unsigned int binIdx = threadIdx.x; binIdx < num_bins; binIdx += blockDim.x) {
20.     atomicAdd(&(histo[binIdx]), histo_s[binIdx]);
    }
}
```

Figure 9.11 An aggregated text histogram kernel.

Figure 9.11 shows an aggregated text histogram kernel. Each thread declares three additional register variables curr_index, prev_index and accumulator. The accumulator keeps track of the number of updates aggregated thus far and prev_index tracks the index of the histogram element whose updates has been aggregated. Each thread initializes the prev_index to -1 (Line 6) so that no alphabet input will match it. The accumulator is initialized to zero (Line 7), indicating that no updates has been aggregated.

When an alphabet data is found, the thread compares the index of the histogram element to be updated with that being aggregated. If the index is different, the streak of aggregated updates to the histogram element has ended (Line 12). The thread uses atomic operation to add the accumulator value to the histogram element whose index is tracked by prev_index. This effectively flushes out the total contribution of the previous streak of aggregated updates. If the curr_index matches the prev_index, the thread simply adds one to the accumulator (Line 17), extending the streak of aggregated updates by one.

An observation is that the aggregated kernel requires more statements and variables. Thus, if the contention rate is low, an aggregated kernel may execute at lower speed than the simple kernel. However, if the data distribution leads to heavy contention in atomic operation execution, aggregation result in significant higher speed.

9.8. Summary

Histogramming is a very important computation for analyzing large data sets. It also represents an important class of parallel computation patterns where the output location of each thread is data-dependent, which makes it infeasible to apply owner-computes rule. It is therefore a natural vehicle for introducing the practical use of atomic operations that ensure the integrity of read-modify-write operations to the same memory location by multiple threads. Unfortunately, as we explained in this chapter, atomic operations have much lower throughput than simpler memory read or write operations because their throughput is approximately the inverse of two times the memory latency. Thus, in the presence of heavy contention, histogram computation can have surprisingly low computation throughput. Privatization is introduced as an important optimization technique that systematically reduces contention and enabled the use of local memory such as the shared memory which supports low latency and thus high throughput. In fact, supporting very fast atomic operations among threads in a block is a very important use case of the shared memory. For data sets that cause heavy contention, aggregation can also lead to significantly higher execution speed.

9.9. Exercises

1. Assume that each atomic operation in a DRAM system has a total latency of 100ns. What is the maximal throughput we can get for atomic operations on the same global memory variable?
   (A) 100G atomic operations per second

(B) 1G atomic operations per second
(C) 0.01G atomic operations per second
(D) 0.0001G atomic operations per second

2. For a processor that supports atomic operations in L2 cache, assume that each atomic operation takes 4ns to complete in L2 cache and 100ns to complete in DRAM. Assume that 90% of the atomic operations hit in L2 cache. What is the approximate throughput for atomic operations on the same global memory variable?
   (A) 0.225G atomic operations per second
   (B) 2.75G atomic operations per second
   (C) 0.0735 atomic operations per second
   (D) 100G atomic operations per second

3. In question 1, assume that a kernel performs 5 floating-point operations per atomic operation. What is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
   (A) 500 GFLOPS
   (B) 5 GFLOPS
   (C) 0.05 GFLOPS
   (D) 0.0005 GFLOPS

4. In Question 1, assume that we privatize the global memory variable into shared memory variables in the kernel and the shared memory access latency is 1ns. All original global memory atomic operations are converted into shared memory atomic operation. For simplicity, assume that the additional global memory atomic operations for accumulating privatized variable into the global variable adds 10% to the total execution time. Assume that a kernel performs 5 floating-point operations per atomic operation. What is the maximal floating-point throughput of the kernel execution as limited by the throughput of the atomic operations?
   (A) 4500 GFLOPS
   (B) 45 GFLOPS
   (C) 4.5 GFLOPS
   (D) 0.45 GFLOPS

5. To perform an atomic add operation to add the value of an integer variable Partial to a global memory integer variable Total. Which one of the following statement should be used?
   (A) atomicAdd(Total, 1);
   (B) atomicAdd(&Total, &Partial);
   (C) atomicAdd(Total, &Partial);
   (D) atomicAdd(&Total, Partial);

References

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

Chapter 10

Parallel Patterns: Sparse Matrix Computation

An introduction to data compression and regularization

CHAPTER OUTLINE

Our next parallel pattern is sparse matrix computation. In a sparse matrix, the majority of the elements are zeros. Storing and processing these zero elements are wasteful in terms of memory capacity, memory bandwidth, time and energy. Many important real-world problems involve sparse matrix computation. Due to the importance of these problems, several sparse matrix storage formats and their corresponding processing methods have been proposed and widely used in the field. All of these methods employ some type of compaction techniques to avoid storing or processing zero elements at the cost of introducing some level of irregularity into the data representation. Unfortunately, such irregularity can lead to underutilization of memory bandwidth, control flow divergence, and load imbalance in parallel computing. It is therefore important to strike a good balance between compaction and regularization. Some storage formats achieve a higher level of compaction at a high-level of irregularity. Others achieve a more modest level of compaction while keeping the representation more regular. The parallel computation performance of their corresponding methods is known to be heavily dependent on the distribution of non-zero elements in the sparse matrices. Understanding the wealth of work in sparse matrix storage formats and their corresponding parallel algorithms gives a parallel programmer important background for addressing compaction and regularization challenges in solving related problems.

10.1. Background

A sparse matrix is a matrix where the majority of the elements are zeros. Sparse matrices arise in many science, engineering and financial modeling problems. For example, as we saw in Chapter 6, matrices can be used to represent the coefficients in a linear system of equations. Each row of the matrix represents one equation of the linear system. In many science and engineering problems, the large number of variables and equations involved are sparsely coupled. That is, each equation only involves a small number of variables. This is illustrated in Figure 10.1, where each column of the matrix corresponds to the coefficients for a variable: column 0 for x_0, column 1 for x_1, etc. For example, the fact that row 0 has non-zero elements in columns 0 and 2 indicates that variables x_0 and x_2 are involved in equation 0. It should be clear that none of the variables are present in equation 1, variables x_1, x_2 and x_3 are present in equation 2, and finally variables x_0 and x_3 are present in equation 3.

Row 0	3	0	1	0
Row 1	0	0	0	0
Row 2	0	2	4	1
Row 3	1	0	0	1

Figure 10.1: A simple sparse matrix example

Sparse matrices are typically stored in a format, or representation, that avoids storing zero elements. We will start with the Compressed Sparse Row (CSR) storage format, which is illustrated in Figure 10.2. CSR stores only non-zero values in a one dimensional data storage, shown as data[] in Figure 10.2. Array data[] stores all the non-zero values in the sparse matrix in Figure 10.1. This is done by storing the non-zero elements of Row 0 (3 and 1) first, followed by the non-zero elements of Row 1 (none), followed by the non-zero elements of Row 2 (2, 4, 1), and finally the non-zero elements of Row 3 (1, 1). The format compresses away all the zero elements.

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers	row_ptr[5]	{ 0, 2, 2, 5, 7 }		

Figure 10.2: Example of Compressed Sparse Row (CSR) format.

With the compressed format, we need to put in two sets of markers to preserve the structure of the original sparse matrix in the compressed representation. The first set of markers form a column index array, col_index[] in Figure 10.2, that gives the column index of every non-zero value in the original sparse matrix. Since we have squeezed away non-zero elements of each row, we need to use these markers to remember where the remaining elements were in

the original rows of the sparse matrix. For example, value 3 and 1 came from columns 0 and 2 of row 0 in the original sparse matrix. The col_index[0] and col_index[1] elements are assigned to store the column indices for these two elements. For another example, values 2, 4, and 1 came from columns 1, 2, and 3 of row 2 in the original sparse matrix. Therefore, col_index[2], col_index[3], and col_index[4] store indices 1, 2, and 3.

The second set of markers give the starting location of every row in the compressed format. This is because the size of each row becomes variable after zero elements are removed. It is no longer possible to use indexing based on the fixed row size to find the starting location of each row in the compressed storage. In Figure 10.2, we show a row_ptr[] array whose elements are the indices for the beginning locations of each row. That is, row_ptr[0] indicates that Row 0 starts at location 0 of the data[] array, row_ptr[1] indicates that Row 1 starts at location 2, etc. Note that row_ptr[1] and row_ptr[2] are both 2. This means that none of the elements of the Row 1 is stored in the compressed format. This makes sense since Row 1 in Figure 10.1 consists entirely of zero values. Note also that row_ptr[4] stores the starting location of a non-existing "Row 4". This is for convenience, as some algorithms need to use the starting location of the next row to delineate the end of the current row. This extra marker gives a convenient way to locate the ending location of Row 3.

As we discussed in Chapter 6, matrices are often used in solving linear system of N equations of N variables in the form of $A*X + Y = 0$, where A is an N×N matrix, X is a vector of N variables, and Y is a vector of N constant values. The objective is to solve for the X variable values that will satisfy all the equations. An intuitive approach is to inverse the matrix so that $X = A^{-1} * (-Y)$. This can be done for matrices of moderate size through methods such as Guassian elimination, as we illustrated in Chapter 6. While it is theoretically possible to use these methods to solve equations represented in sparse matrices, the sheer size of many sparse matrices can simply overwhelm this intuitive approach. Furthermore, an inversed sparse matrix is often much larger than the original due the fact that the inversion process tends to generate a large number of addition non-zero elements called fill-in. As a result, it is often impractical to compute and store the inversed matrix in solving real-world problems.

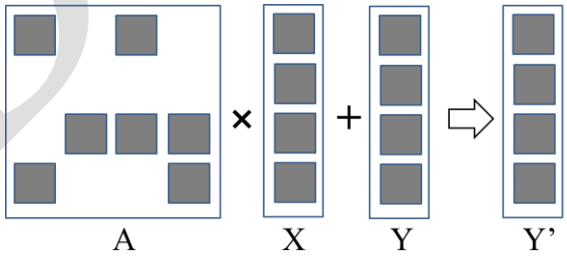


Figure 10.3: A small example of matrix-vector multiplication and accumulation

Instead, linear systems of equations represented in sparse matrices can be better solved with an iterative approach. When the sparse matrix A is positive-definite (i.e., $x^T Ax > 0$ for all non-zero vectors x in R^n), one can use Conjugate Gradient methods to iteratively solve the

corresponding linear system with guaranteed convergence to a solution [Hest1952]. Conjugate Gradient methods guess a solution for X and perform A*X+Y and see if the result is close to a 0 vector. If not, we can use a gradient vector formula to refine the guessed X and perform another iteration of A*X+Y.

The most time consuming part of such iterative approaches is in the evaluation of A*X+Y, which is a sparse matrix-vector multiplication and accumulation. Figure 10.3 shows a small example of matrix-vector multiplication and accumulation, where A is a sparse matrix. The dark squares in A represent non-zero elements. In contrast, both X and Y are typically dense vectors. That is, most of the elements of X and Y hold non-zero values. Due to its importance, standardized library function interfaces have been created to perform this operation under the name SpMV (Sparse Matrix Vector multiplication and accumulate). We will use SpMV to illustrate the important tradeoffs between different storage formats in parallel sparse matrix computation.

```
1.   for (int row = 0; row < num_rows; row++) {
2.     float dot = 0;
3.     int row_start = row_ptr[row];
4.     int row_end =   row_ptr[row+1];
5.     for (int elem = row_start; elem < row_end; elem++) {
6.         dot += data[elem] * x[col_index[elem]];
       }
7.     y[row] += dot;
     }
```

Figure 10.4: a sequential loop that implements SpMV based on the CSR format

A sequential implementation of SpMV based on the CSR format is quite straightforward, which is shown in Figure 10.4. We assume that the code has access to (1) num_rows, a function argument that specifies the number of rows in the sparse matrix, (2) a floating point data array of A elements (via the data[] input parameter), two floating-point x[] and y[] arrays of X and Y elements, and two integer row_ptr and col_index arrays as described in Figure 10.2. There are only seven lines of code. Line 1 is a loop that iterates through all rows of the matrix, with each iteration calculating a dot product of the current row and the vector x.

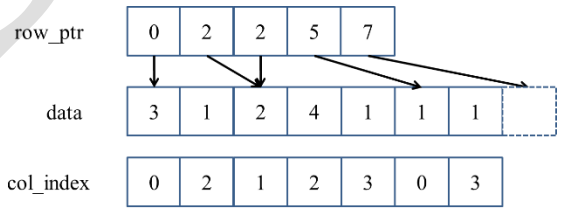


Figure 10.5: Sequential loop SpMV ~~loop~~ operating on the sparse matrix example in Figure 10.1

In each row, Line 2 first initializes the dot product to zero. It then sets up the range of data[] array elements that belong to the current row. The starting and ending locations can be loaded from the row_ptr[] array. Figure 10.5 illustrates this for the small sparse matrix in Figure 10.1. For row=0, row_ptr[row] is 0 and row_ptr[row+1] is 2. Note that the two elements from Row 0 reside in data[0] and data[1]. That is, row_ptr[row] gives the starting position of the current row and row_ptr[row+1] gives the starting position of the next row, which is one after the ending position of the current row. This is reflected in the loop in Line 5, where the loop index iterates from position given by row_ptr[row] to row_ptr[row+1]-1.

The loop body in Line 6 calculates the dot product for the current row. For each element, it uses the loop index elem to access the matrix A element in data[elem]. It also uses elem to retrieve the column index for the element from col_index[elem]. This column index is then used to access the appropriate x element for multiplication. For example, The element in data[0] and data[1] are form column 0 (col_index[0]=0) and column 2 (col_index[1]=2). So the inner loop will perform the dot product for row 0 as data[0]*x[0]+data[1]*x[2]. The reader is encouraged to work out the dot product for other rows as an exercise.

CSR completely removes all zero elements from the storage. It does incur storage overhead by introducing the col_index and row_ptr arrays. In our small example where the number of zero elements is not much larger than the number of non-zero elements, the storage overhead is actually more than the space saved by not storing the zero elements. However, it should be clear that for sparse matrices where the vast majority of elements are zeros, the overhead introduced is far less than the space saved by not storing zeros. For example, in a sparse matrix where only 1% of the elements are non-zero values, the total storage for the CSR representation including all the overhead would be around 2% of the space required to store both zero and non-zero elements.

Removing all zero elements from the storage also eliminates the need to fetch these zero elements from memory or perform useless multiplication operations on these zero elements. This can significantly reduce the consumption of memory bandwidth and computation resources.

It should be obvious that any SpMV computation code will reflect the storage format assumed. Therefore, we will add the storage format to the name of a code to clarify the combination used. We will refer to the SpMV code in Figure 10.4 as sequential SpMV/CSR. With a good understanding of sequential SpMV/CSR, we are now ready to discuss parallel sparse computation.

10.2. Parallel SpMV Using CSR

Note that the dot product calculation for each row of the sparse matrix is independent of that for other rows. That is, all iterations of the outer loop (Line 1) in Figure 10.4 are logically independent of each other. We can easily convert this sequential SpMV/CSR into a parallel

CUDA kernel by assigning each iteration of the outer loop to a thread. That is, each thread calculates the inner product for a row of the matrix, which is illustrated in Figure 10.6, where Thread 0 calculates the dot product for row 0, Thread 1 for row 1, and so on.

Thread 0	3	0	1	0
Thread 1	0	0	0	0
Thread 2	0	2	4	1
Thread 3	1	0	0	1

Figure 10.6: Example of mapping threads to rows in parallel SpMV/CSR

In a real-world sparse matrix application, there are usually thousands to millions of rows, each of which contains tens to hundreds of non-zero elements. This makes the mapping shown in Figure 10.6 seem very appropriate: there are many threads and each thread has a substantial amount of work. We show a parallel SpMV/CSR in Figure 10.7.

```
1.  __global__ void SpMV_CSR(int num_rows, float *data, int *col_index,
      int *row_ptr, float *x, float *y) {

2.    int row = blockIdx.x * blockDim.x + threadIdx.x;

3.    if (row < num_rows) {
4.      float dot = 0;
5.      int row_start = row_ptr[row];
6.      int row_end =   row_ptr[row+1];
7.      for (int elem = row_start; elem < row_end; elem++) {
8.        dot += data[elem] * x[col_index[elem]];
        }
9.      y[row] = dot;
      }

    }
```

Figure 10.7: A parallel SpMV/CSR kernel

It should be clear that the kernel looks almost identical to the sequential SpMV/CSR loop. The outermost loop construct has been removed since it is replaced by the thread grid. In Line 2, the row index assigned to a thread is calculated as the familiar expression blockIdx.x*blockDim.x + threadIdx.x. Also, due to the need to handle any arbitrary number of rows, Line 3 checks if the row index of a thread exceeds the number of rows. This handles the situation where the number of rows is not a multiple of the thread block size.

While the parallel SpMV/CSR kernel is quite simple, it has two major shortcomings. First the kernel does not make coalesced memory accesses. If the reader examines Figure 10.5, it should be cl ear that adjacent threads will be making simultaneous non-adjacent memory accesses. In our small example, threads 0, 1, 2, and 3 will access data[0], none, data[2], and

data[5] in the first iteration of their dot product loop. They will then access data[1], none, data[3], and data[6] in the second iteration, and so on. As a result, the parallel SpMV/CSR kernel in Figure 10.7 does not make efficient use of memory bandwidth.

The second shortcoming of the SpMV/CSR kernel is that it can potentially have significant control flow divergence in all warps. The number of iterations taken by a thread in the dot product loop depends on the number of non-zero elements in the row assigned to the thread. Since the distribution of non-zero elements among rows can be random, adjacent rows can have very different number of non-zero elements. As a result, there can be wide spread control flow divergence in most or even all warps.

It should be clear that both the execution efficiency and memory bandwidth efficiency of the parallel SpMV kernel depends on the distribution of the input data matrix. This is quite different from most of the kernels we have studied so far. However, such data-dependent performance behavior is quite common in real-world applications. This is one of the reasons why parallel SpMV is such an important parallel pattern, it is simple and yet it illustrates an important behavior in many complex parallel applications. We will discuss three important techniques in the next sections to address the non-coalesced memory accesses and control flow divergence in the parallel SpMV/CSR kernel.

10.3. Padding and Transposition

The problems of non-coalesced memory accesses and control divergence can be addressed by applying data padding and transposition on the sparse matrix data. These ideas were used in the ELL storage format, whose name came from the sparse matrix package in ELLPACK a package for solving elliptic boundary value problems [Rice1984] A simple way to understand the ELL format is to start with CSR format, as is illustrated in Figure 10.8.

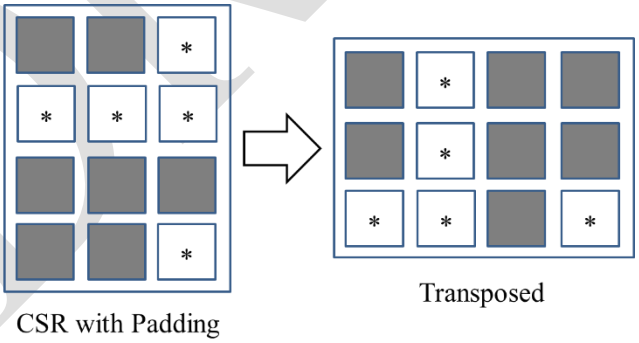


Figure 10.8. ELL Storage Format

From a CSR representation, we first determine the rows with the maximal number of non-zero elements. We then add **dummy** (zero) elements to all other rows after the non-zero elements to make them the same length as the maximal rows. This makes the matrix a rectangular matrix. For our small sparse matrix example, we determine that row 2 has the

maximal number of elements. We then add one zero element to row 0, three zero elements to row 1, and one zero element to row 3 to make all them the same length. These additional zero elements are shown as squares with an * in Figure 10.8. Now the matrix has become a rectangular matrix. Note that the col_index array also needs to be padded the same way to preserve their correspondence to the data values.

We can now lay the padded matrix out in column major order. That is, we will place all elements of column 0 in consecutive memory locations, followed by all elements of column 1, and so on. This is equivalent to **transposing** the rectangular matrix in the row major order used by the C language. In terms of our small example, after the transposition, data[0] through data[3] now contain 3, *, 2, 1, the 0^{th} elements of all rows. This is illustrated in the bottom portion of Figure 10.9. Similarly, col_index[0] through col_index[3] contain the column positions of 0^{th} elements of all rows. Note that we no longer need the row_ptr since the beginning of row i is now simply data[i]. With the padded elements, it is also very easy to move from the current element of row i to the next element by simply adding the number of rows in the original matrix to the index. For example, the 0^{th} element of row 2 is in data[2] and the next element is in data[2+4]=data[6], where 4 is the number of rows in the original matrix in our small example.

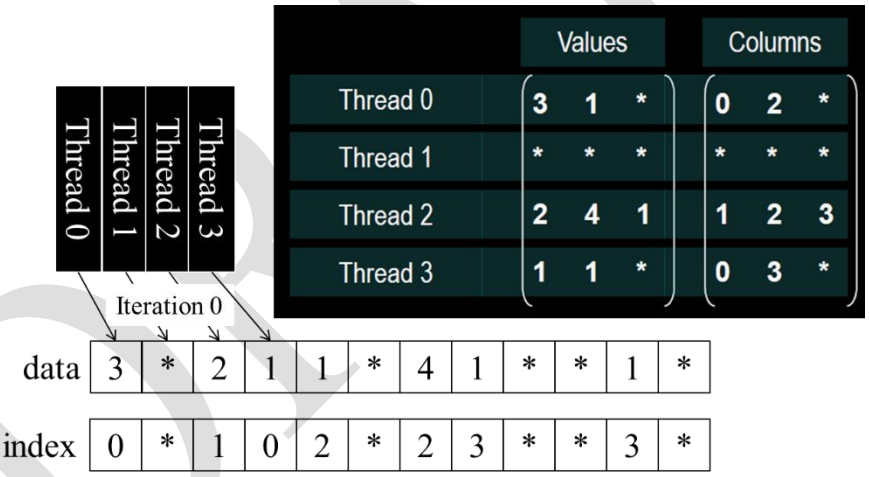


Figure 10.9: More details of our small example in ELL

Using the ELL format, we show a parallel SpMV_ELL kernel in Figure 10.10. The kernel receives slightly different arguments. It no longer needs the row_ptr. Instead, it needs an argument num_elem to know the number of elements in each row after padding. Recall that num_elem is the maximal number of non-zero elements among all rows in the original sparse matrix.


```
1.  __global__ void SpMV_ELL(int num_rows, float *data, int *col_index,
      int num_elem, float *x, float *y) {

2.    int row = blockIdx.x * blockDim.x + threadIdx.x;
3.    if (row < num_rows) {
4.      float dot = 0;
5.      for (int i = 0; i < num_elem; i++) {
6.        dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];
        }
7.      y[row] = dot;
      }
    }
```

Figure 10.10: A parallel SpMV/ELL kernel.

A first observation is that the SpMV/ELL kernel code is simpler than that of SpMV/CSR. With padding, all rows are now of the same length. In the dot-product loop in Line 5, all threads can simply loop through the number of elements given by num_elem. As a result, there is no longer control flow divergence in warps: all threads now iterate exactly the same number of times in the dot-product loop. In the case where a dummy element is used in a multiplication and accumulation step, it will not affect the final result because its value is 0.

A second observation is that in the dot-product loop body, each thread accesses its 0^{th} element in data[row] and in general, its i^{th} element in data[row+i*num_rows]. As we have seen in Figure 10.9, by arranging the elements in column major order all adjacent threads are now accessing adjacent memory locations, enabling memory coalescing and thus making more efficient use of memory bandwidth.

By eliminating control flow divergence and enabling memory coalescing, SpMV/ELL should run faster than SPMV/CSR. Furthermore, SpMV/ELL is simpler. This seems to make SpMV/ELL an all-around winning approach. Unfortunately, it does have a potential downside. In situations where one or a small number of rows have an exceedingly large number of non-zero elements, the ELL format will result in excessive number of padded elements. These padded elements will take up storage, need to be fetched, and take part in calculations even though they do not contribute to the final result. They consume memory storage, memory bandwidth, and execution cycles. Consider our sample matrix, in the ELL format we have replaced a 4x4 matrix with a 4x3 matrix, and with the overhead from the column indices we are storing more data than contained in the original 4x4 matrix.

For example, if a 1000x1000 sparse matrix has 1% of its elements of non-zero value. This means that on average, each row has 10 non-zero elements. With the overhead, the size of a CSR representation would about 2% of the uncompressed total size. Assume that one of the rows has 200 non-zero values while all other rows have less than 10. Using the ELL format, we would pad all other rows to 200 elements. This makes the ELL representation about 40% of the uncompressed total size and 20 times larger than the CSR representation. While the excessively long row will make only one of the warps of the SpMV/CSR kernel run for a

long time, the padding will make all warps of the SpMV/ELL kernel run for a long time. With so many padded dummy elements, an SpMV/ELL kernel can actually run more slowly than an SpMV/CSR kernel. This calls for a method to control the number of padded elements when we convert from the CSR format to the ELL format.

10.4. Using a Hybrid Approach to Regulate Padding

The root of the problem with excessive padding in the ELL representation is that one or a small number of rows have exceedingly large number of non-zero elements. If we have a mechanism to "take away" some elements from these rows, we can reduce the number of padded elements in ELL. The Coordinate (COO) format provides such a mechanism.

			Row 0		Row 2			Row 3		
Nonzero values	data[7]	{	3,	1,	2,	4,	1,	1,	1	}
Column indices	col_index[7]	{	0,	2,	1,	2,	3,	0,	3	}
Row indices	row_index[7]	{	0,	0,	2,	2,	2,	3,	3	}

Figure 10.11: Example of Coordinate (COO) format

The COO format is illustrated in Figure 10.11, where each non-zero element is stored with both its column index and row index. We have both col_index and row_index arrays to accompany the data array. For example A[0,0] of our small example is now stored with both its column index (0 in col_index[0]) and its row index (0 in row_index[0]). With the COO format, one can look at any element in the storage and know where the element came from in the original sparse matrix. Like the ELL format, there is no need for row_ptr since each element self-identifies its column and row position.

While the COO format does come with the cost of additional storage for the row_index array, it also comes with the additional benefit of flexibility. We can arbitrarily re-order the elements in a COO format without losing any information as long as we re-order the data, col_index, and row_index arrays the same way. This is illustrated using our small example in Figure 10.12.

Nonzero values	data[7]	{	1	1,	2,	4,	3,	1	1	}
Column indices	col_index[7]	{	0	2,	1,	2,	0,	3,	3	}
Row indices	row_index[7]	{	3	0,	2,	2,	0,	2,	3	}

Figure 10.12 Re-ordering Coordinate (COO) format.

In Figure 10.12, we have reordered the elements of data, col_index, and row_index. Now data[0] actually contains an element from row 3 and column 0 of the original sparse matrix. Because we have also shifted the row index and column index values along with the data value, we can correctly identify this element's location in the original sparse matrix. The

reader may ask why we would want to reorder these elements. Such reordering would disturb the locality and sequential patterns that are important for efficient use of memory bandwidth.

The answer lies in an important use case for the COO format. It can be used to curb the length of rows in the CSR or ELL format. First, we make an important observation. In the COO format, we can process the elements in any order we want. For each element in data[i], we can simply perform a y[row_index[i]] += data[i] * x[col_index[i]] operation. The correct y element identified by row_index[i] will receive the correct contribution from the product of data[i] and x[col_index]. If we make sure somehow we perform this operation for all elements of data, we will calculate the correct final answer regardless of the order in which we process these elements.

Before we convert from a sparse matrix from CSR to ELL, we can take away some of the elements from the rows with exceedingly large number of non-zero elements and place them into a separate COO storage. We can use SpMV/ELL on the remaining elements. With excess elements removed from the extra-long rows, the number of padded elements for other rows can be significantly reduced. We can then use a SpMV/COO to finish the job. This approach of employing two formats to collaboratively complete a computation is often referred to as a hybrid method.

Let's illustrate a hybrid ELL and COO method for SpMV using our small sparse matrix, as shown in Figure 10.13. We see that row 2 has the most number of non-zero elements. We remove the last non-zero element of row 2 from the ELL representation and move it into a separate COO representation. By removing the last element of row 2, we reduce the maximal number of non-zero elements among all rows in the small sparse matrix from 3 to 2. As shown in Figure 10.13, we reduced the number of padded elements from 5 to 2. More importantly, all threads now only need to take 2 iterations rather than 3. This can give a 50% acceleration to the parallel execution of the SpMV/ELL kernel.

A typical way of using an ELL-COO hybrid method is for the host to convert the format from something like a CSR format into ELL. During the conversion, the host removes some non-zero elements form the rows with exceedingly large number of non-zero elements. The host places these elements into a COO representation. The host then transfers the ELL representation of the data to a device. When the device completes the SpMV/ELL kernel, it transfers the resulting y values back to the host. These values are missing the contributions from the elements in the COO representation. The host performs a sequential SpMV/COO kernel on the COO elements and finishes their contributions to the y element values.

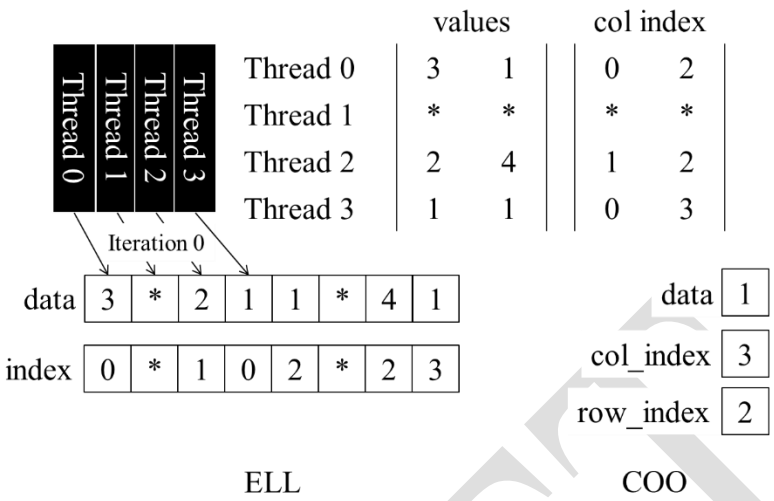


Figure 10.13: Our Small example in ELL and COO Hybrid

The user may question whether the additional work done by the host to separate COO elements from an ELL format could incur too much overhead. The answer is it depends. In situations where a sparse matrix is only used in one SpMV calculation, this extra work can indeed incur significant overhead. However, in many real-work applications, the SpMV is performed on the same sparse kernel repeatedly in an iterative solver. In each iteration of the solver, the x and y vectors vary but the sparse matrix remains the same since its elements correspond to the coefficients of the linear system of equations being solved and these coefficients do not change from iteration to iteration. So, the work done to produce both the hybrid ELL and COO representation can be amortized across many iterations. We will come back to this point in the next section.

In our small example, the device finishes the SpMV/ELL kernel on the ELL portion of the data. The y values are then transferred back to the host. The host then add the contribution of the COO element with the operation y[2] += data[0] * x[col_index[0]] = 1*x[3]. Note that there are in general multiple non-zero elements in the COO format. So, we expect that the host code to be a loop as shown in Figure 10.14.

```
1.    for (int i = 0; i < num_elem; row++)
2.      y[row_index[i]] += data[i] * x[col_index[i]];
```

Figure 10.14: a sequential loop that implements SpMV/COO

The loop is extremely simple. It iterates through all the data elements and perform the multiply and accumulate operation on the appropriate x and y elements using the accompanying col_index and row_index elements. We will not present a parallel SpMV/COO kernel. It can be easily constructed using each thread to process a portion of the data elements and use an atomic operation to accumulate the results into y elements. This is because the threads are no longer mapped to a particular row. In fact, many rows will likely

be missing from the COO representation; only the rows that have exceedingly large number of non-zero elements will have elements in the COO representation. Therefore, it is better just to have each thread to take a portion of the data element and use atomic operation to make sure that none of the threads will trample the contribution of other threads.

The hybrid SpMV/ELL-COO method is a good illustration of productive use of both CPUs and GPUs in a heterogeneous computing system. The CPU can perform SpMV/COO fast using its large cache memory. The GPU can perform SpMV/ELL fast using its coalesced memory accesses and large number of hardware execution units. The removal of some elements from the ELL format is a form of regularization technique: it reduces the disparity between long and short rows and makes the workload of all threads more uniform. Such improved uniformity results in benefits such as less control divergence in a SpMV/CSR kernel or less padding in a SpMV/ELL kernel.

10.5. Sorting and Partitioning for Regularization

While COO helps to regulate the amount of padding in an ELL representation, we can further reduce the padding overhead by sorting and partitioning the rows of a sparse matrix. The idea is to sort the rows according to their length, say from the longest to the shortest. This is illustrated with our small sparse matrix in Figure 10.15. Since the sorted matrix looks largely like a triangular matrix, the format is often referred to as Jagged Diagonal Storage (JDS) format. As we sort the rows, we typically maintain an additional jds_row_index array that preserves the original index of the row. For CSR, this is similar to the row_ptr array in that ~~there is~~both arrays have one element ~~per~~ for each row of the matrix. Whenever we exchange two rows in the sorting process, we also exchange the corresponding elements of the jds_row_index array. This way, we can always keep track the original position of all rows.

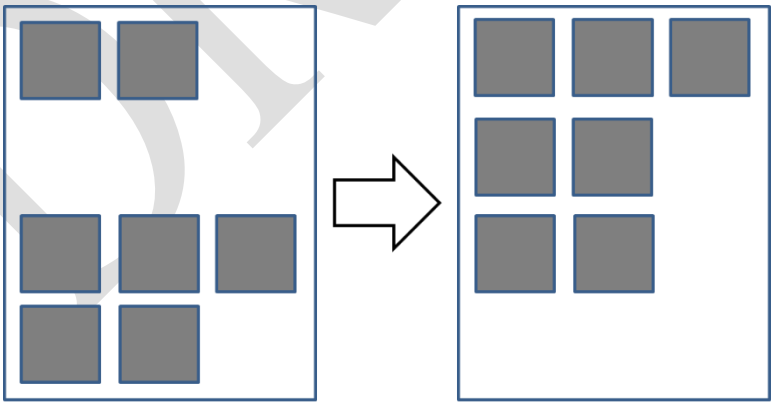


Figure 10.15: Sorting rows according to their length

Once a sparse matrix is in JDS format, we can partition the matrix into sections of rows. Since the rows have been sorted, all rows in a section will likely have more or less uniform number of non-zero elements. In Figure 10.15, we can divide the small matrix into three sections: the first section consists of the one row that has three elements. The second section consists of the two rows with two elements each. The third section consists of one row without any element. We can then generate ELL representation for each section. Within each section, we only need to pad the rows to match the row with the maximal number of elements in that section. This would reduce the number of padded elements. In our example, we do not even need to pad within any of the three sections. We can then transpose each section independently and launch a separate kernel on each section. In fact, we do not even need to launch a kernel for the section of rows with no non-zero elements.

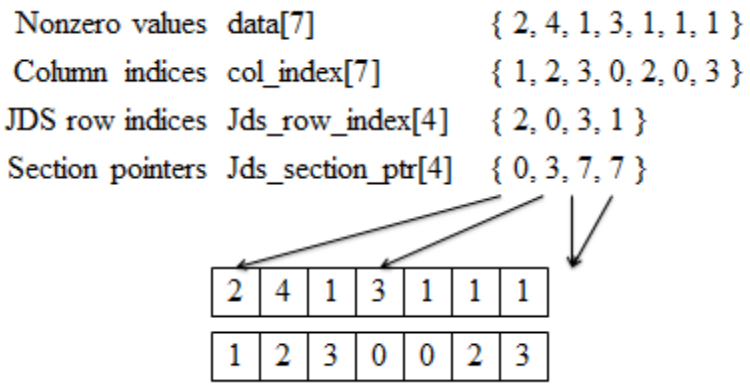


Figure 10.16 JDS format and sectioned ELL

Figure 10.16 shows a JDS-ELL representation of our small sparse matrix. It assumed the same sorting and partitioning results shown in Figure 10.15. Out of the three sections, the first section has only one row so the transposed layout is the same as the original. The second section is a 2×2 matrix and has been transposed. The third section consists of Row 1, which does not have any non-zero element. This is reflected in the fact that its starting location and the next section's starting position are identical.

We will not show a SpMV/JDS kernel. The reason is that we would be just using either an SpMV/CSR kernel on each section of the CSR, or a SpMV/ELL kernel on each section of the ELL after padding. The host code required to create a JDS representation and to launch SpMV kernels on each section of the JDS representation is left as an exercise.

Note that we want each section to have a large number of rows so that its kernel launch will be worthwhile. In the extreme cases where a very small number of rows have extremely large number of non-zero elements, we can still use the COO hybrid with JDS to allow us to have more rows in each section.

Once again, the reader should ask whether sorting rows will result into incorrect solutions to the linear system of equations. Recall that we can freely re-order equations of a linear

system without changing the solution. As long as we re-order the y elements along with the rows, we are effectively re-ordering the equations. Therefore, we will end up with the correct solution. The only extra step is to reorder the final solution back to the original order using the jds_row_index array.

The other question is whether sorting will incur significant overhead. The answer is similar to what we saw in the hybrid method. As long as the SpMV/JDS kernel is used in an iterative solver, one can afford to perform such sorting as well as the re-ordering of the final solution x elements and amortize the cost among many iterations of the solver.

In more recent devices, the memory coalescing hardware has relaxed the address alignment requirement. This allows one to simply transpose a JDS-CSR representation. Note that we do need to adjust the jds_section_ptr array after transposition. This further eliminates the need to pad rows in each section. As memory bandwidth becomes increasingly the limiting factor of performance, eliminating the need to store and fetch padded elements can be a significant advantage. Indeed, we have observed that while sectioned JDS-ELL tend to give the best performance on older CUDA devices, transposed JDS-CSR tend to give the best performance on Fermi and Kepler.

We would like to make an additional remark on the performance of sparse matrix computation as compared to dense matrix computation. In general, the FLOPS rating achieved by either CPUs or GPUs are much lower for sparse matrix computation than for dense matrix computation. This is especially true for SpMV, where there is no data re-use in the sparse matrix. The CGMA value (Chapter 4) is essentially 1, limiting the achievable FLOPS rate to a small fraction of the peak performance. The various formats are important for both CPUs and GPUs since both are limited by memory bandwidth when performing SpMV. Many folks have been surprised by the low FLOPS rating of this type of computation on both CPUs and GPUs in the past. After reading this chapter, one should be no longer be surprised.

10.6. Summary

In this chapter, we presented sparse matrix computation as an important parallel pattern. Sparse matrices are important in many real-world applications that involve modeling complex phenomenon. Furthermore, sparse matrix computation is a simple example of data-dependent performance behavior of many large real-world applications. Due to the large amount of zero elements, compaction techniques are used to reduce the amount of storage, memory accesses and computation performed on these zero elements. Unlike most other kernels presented in this book so far, the SpMV kernels are sensitive to the distribution of data, specifically the non-zero elements in sparse matrices. Not only can the performance of each kernel vary significantly across matrices, their relative merit can also change significantly. Using this pattern, we introduce the concept of regularization using hybrid methods and sorting/partitioning. These regularization methods are used in many real-world

applications. Interestingly, some of the regularization techniques re-introduce zero elements into the compacted representations. We use hybrid methods to mitigate the pathological cases where we could introduce too many zero elements. Readers are referred to [Bell2009] and encouraged to experiment with different sparse data sets to gain more insight into the data dependent performance behavior of the various SpMV kernels presented in this chapter.

References

[Rice1984] John R. Rice and Ronald F. Boisvert, *Solving Elliptic Problems Using, ELLPACK, Springer Verlag*, 1984, 497 pages

[Hest1952] Hestenes, Magnus R.; Stiefel, Eduard (December 1952). "Methods of Conjugate Gradients for Solving Linear Systems" (PDF). *Journal of Research of the National Bureau of Standards* **49** (6)

[Bell2009] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," Proceedings of the ACM Conference on High-Performance Computing Networking Storage and Analysis (SC'09), 2009.

10.7. Exercises

1. Complete the host code to produce the hybrid ELL-COO format, launch the ELL kernel on the device, and complete the contributions of the COO elements.

2. Complete the host code for producing JDS-ELL and launch one kernel for each section of the representation.

3. Consider the following sparse matrix:

```
1 0 7 0
0 0 8 0
0 4 3 0
2 0 0 1
```

Represent it in each of the following formats: (a) COO, (b) CSR, and (c) ELL.

4. Given a sparse matrix of integers with m rows, n columns, and z non-zeros. How many integers are needed to represent the matrix in (a) COO, (b) CSR, and (c) ELL. If the information provided is not enough, indicate what information is missing.

Chapter 16
Application Case Study – Machine Learning

Boris Ginsburg

Keywords: Convolutional Neural Network, Machine Learning, Deep Learning, Matrix-matrix Multiplication, Forward Propagation, Gradient Back-Propagation, Training, cuDNN

CHAPTER OUTLINE

In this chapter we will describe a case study of accelerating machine learning algorithms with GPUs. Machine learning has been used in many applications to train or adapt the application logic according to the experience gleaned from data sets. To be effective, one often needs to conduct such training with a massive amount of data. While machine learning has existed as a subject of computer science for a long time, it has recently gained a great deal of practical industry acceptance due to the availability of inexpensive, massively parallel GPU computing systems that can effectively train application logic with massive data sets. We will start with brief introduction to deep learning, and then consider one of the most popular algorithms – convolutional neural networks in more details. Convolutional neural networks have high compute-to-bandwidth ratio, and high levels of parallelism, which makes them a perfect candidate for GPU acceleration. We will first implement a convolutional neural network with a very basic algorithm. Next we show how we can

improve this basic implementation with shared memory. We will then show how one can formulate the convolutional layers as matrix multiplication problems.

16.1. Background

Machine learning is a field of computer science which explores algorithms whose logic can be learned directly from data rather than ~~be~~ explicitly programmed. Machine learning is most successful in computing tasks where designing explicit algorithms is infeasible, mostly because there is not enough knowledge in the design of such explicit algorithms. For example, machine learning has ~~provided the~~contributed to the recent improvement in the core program logic in application areas such as automatic speech recognition, computer vision, natural language processing, and search engines.

Conventional machine-learning systems required humans with considerable domain expertise to define meaningful features for transforming the raw data (e.g. the pixels of an image or digital samples of a speech excerpt) into a curated representation, from which the machine-learning algorithms could detect important patterns that can be used for training the application logic. By contrast, deep-learning is a set of methods that allow~~s~~ a machine-learning system to automatically discover the complex features needed for detection directly from raw data [1]. This area of machine learning is called ‘deep’ because it is based on the idea of hierarchical, multi-level feature representation. The hierarchical features are obtained by composing simple non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. For example in computer vision, the first layer of representation typically detect edges at particular orientations and locations in the image. The second layer typically detects so called ‘motifs’ by spotting particular patterns of edges, regardless of small variations in the edge positions. The third layer assemble these motifs into larger parts. Such layered structures, as illustrated in Figure 16.1, are often referred to as ‘feed forward networks’ since the information flows from one layer to the next layer in one direction in these systems.

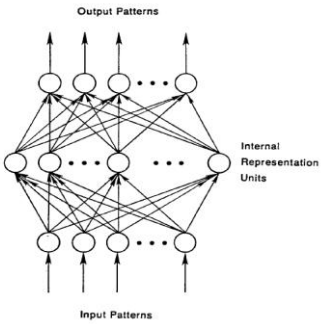


Figure 16.1: A multi-layer feed forward network

Deep learning procedures based on feed forward networks can learn very complex features which can achieve more accurate pattern recognition results ~~than~~ compared to features that are manually engineered by humans; however, this method requires that~~,~~ ~~as long as~~ there is enough data to allow the system to automatically discover ~~sufficient~~ an adequate number of relevant patterns. There is one ~~particular type~~category of deep learning procedures that are easier to train and that can be generalized much better than others. ~~This type of~~These deep-learning procedures ~~is~~ are based on a particular type of feed forward network called the convolutional neural network (CNN).

The ~~convolutional neural network (~~CNN~~s)~~ were invented in late 1980s [3]. By the early 1990s, ~~convolutional neural network~~CNNs had been successfully applied to automated speech recognition, optical character recognition (OCR), hand-writing recognition, and face recognition. However, until the late 1990s, the mainstream of computer vision and that of automated speech recognition had been based on carefully engineered features. ~~There was insufficient~~The amount of labeled data was insufficient for a deep-learning system to compete with recognition/classification functions crafted by human experts. It was a common belief that it was computationally infeasible to automatically build hierarchical feature extractors that have enough layers to perform better than human-defined application-specific feature extractors.

Interest in deep, feed ~~-~~forward networks was revived around 2006 by a group of researchers who introduced unsupervised learning methods that could create multi-layer, hierarchical feature detectors without requiring labelled data [5]. The first major application of this approach was in speech recognition. The breakthrough was made possible by GPUs that allowed researchers to train networks ten times faster than traditional CPUs. This advancement coupled with the massive amount of media data available online drastically elevated the position of deep-learning approaches. Despite their success in speech, convolutional neural networks were largely ignored in the field of computer vision until 2012.

In 2012, a group of researchers from University of Toronto trained a large, deep convolutional neural network to classify 1,000 different classes in the ILSVRC contest [7]. The network was huge by the norms of the time: it had approximately 60 million parameters and 650,000 neurons. It was trained on 1.2 million high-resolution images from the Imagenet database. The network was trained in only one week on two GPUs using ~~the~~ a very efficient ~~cuda-convnet~~CUDA-based convolutional neural network library ~~[8]~~ written by Alex Krizhevsky [8]. The network achieved break-through results with a winning test error rate of 15.3%. In comparison, the second place team that used the traditional computer vision algorithms had an error rate of 26.2%. This success triggered a revolution in computer vision, and convolutional neural networks (abbreviated as *ConvNet* for the rest of this chapter) became a mainstream tool in computer vision, natural language processing, reinforcement learning, and many other traditional machine learning areas.

16.2. Convolutional neural networks

To explain how ConvNets work, we will use LeNet-5, the network designed in the late 1980s for hand-writing digit recognition [3]. As shown in Figure 16.2, LeNet-5 is composed from 3 types of layers: convolutional layers, subsampling layers, and full connection layers. We will consider each type of layer in next section. The input to the network is shown as a gray image with a hand-written digit represented as 2D 32x32 pixel array. The last layer computes the output, which is the probability for the original image to belong to one of the 10 classes (digits) that the network is set up to recognize.

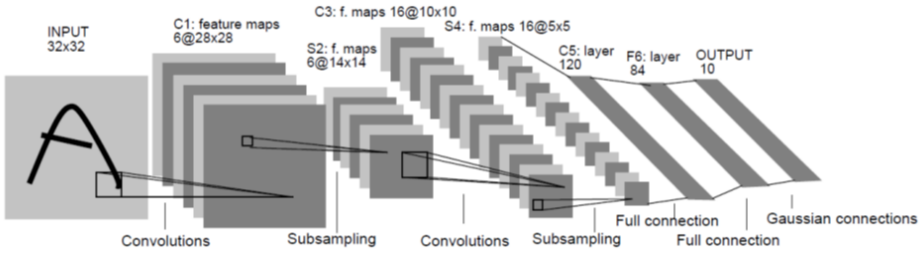


Figure 16.2 LeNet-5, a convolutional neural network for hand-written digit recognition.

ConvNets: basic layers

The computation in a convolutional network~~s~~ is organized as a sequence of layers. We will call inputs to and outputs ~~to~~ from layers 'feature maps'. For example in Figure 16.2, the computation of the C1 layer is organized to generate six output feature maps from the INPUT pixel array. The first type of layers are the convolution layers such as C1. The ~~computation result or~~ output to be produced for input feature maps consists of pixels, each of which is produced by performing a convolution between a small local patch of the feature map pixels ~~of~~ produced by the previous layer (INPUT in the case of C1) and a set of weights (i.e., a convolution mask as defined in Chapter 7) called a filter bank. Furthermore, each output feature mpa pixel is the sum of convolution results from all input feature maps.

All pixels in an input feature map are processed with the same filter bank when generating a particular output feature map. Different pairs of input and output feature map pairs in a layer use different filter banks. For example, there are 6 input feature maps and 16 output feature maps for layer C3. A total of 6*16=96 filter banks will be used in C3. Although not shown in Figure 16.2, all filter banks used in LeNet-5 are 5x5 convolutions. They differ in the 25 weights that are present in them. In general, if~~f~~ a convolution layer~~s~~ has *n* input feature maps and *m* output feature maps, *n***m* different filter banks will be used.

Formatted: Font: Italic

Formatted: Font: Italic

Formatted: Font: Italic

Recall from Chapter 7 that generating a 32x32 convolution image from a 32x32 input image and a 5x5 convolution mask requires one to make assumptions about the "ghost cells". Instead of making such assumptions, the LeNet-5 design simply uses two elements at the edge of each dimension as 'ghost cells.' This reduces the size of each dimension by four: two at the top, two at the bottom, two at the left, and two at the bottom. As a result, we see that ~~with convolution with each filter bank~~for C1, the 32x32 INPUT image results in an output feature map that is a 28x28 image. Figure 16.2 illustrates this computation by showing that a pixel in the C1 layer is generated from a square (5x5 although not explicitly shown) patch of INPUT pixels.

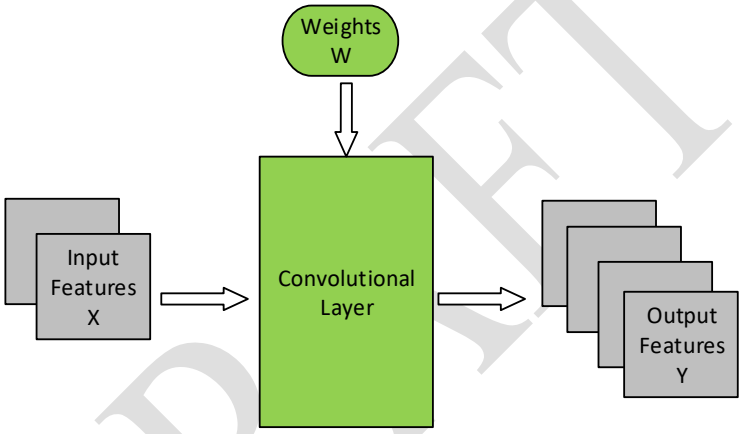


Figure 16.3 Overview of the forward propagation path of a convolution layer.

Figure 16.3 shows an overview of the forward propagation path of a convolution layer. We assume that the input feature maps are stored in a 3D array X[C, H, W], where 'C' is the number of input feature maps, 'H' is the height of each input map image, and 'W' is the width of each input map image. That is, the highest dimension index selects one of the feature maps and the lower two dimension indices selects one of the pixels in a feature map. For example, the input feature maps for the C1 layer is stored in X[1, 32, 32] since there is only one input ~~image~~ feature map (INPUT in Figure 16.2) that consists of 32 pixels in each of the x and y dimensions.

The output feature maps of a convolutional layer ~~is~~ are also stored in a 3D array Y[M, H-K+1, W-K+1], where "M" is the number of output feature maps, "H" is the height of each input map image, "W" is the width of each input map image, and "K" is the height (and width) of

each filter bank W[~~C~~M, ~~M~~C, K, K].[1] For example, the output feature maps for the C1 layer is stored in Y[6, 28, 28] since C1 generates six output feature maps and a 5x5 filter bank. Two elements are used at each edge of the image as halo cells when generating the convolved image. There are MxC filter banks. Filter bank W[m, c,_,_] is used when using input feature map X[c,_,_] to calculate output feature map Y[m,_,_]. Note that each output feature map is the sum of convolutions of all input feature maps. Therefore, we can consider the forward propagation path of a convolutional layer as set of M 3D convolutions, where each 3D convolution is specified by a 3D filter bank that is a C x K x K submatrix of W.

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
  int m, c, h, w, p, q;
  int H_out = H – K + 1;
  int W_out = W – K + 1;

  for(int m = 0;  m < M;  m++)                  // for each output feature maps
    for(int h = 0; h < H_out; h++)              // for each output element
      for(int w = 0; w < W_out; w++) {
        Y[m, h, w] = 0;
        for(int c = 0;  c < C; c++)             // sum over all input feature maps
          for(int p = 0; p < K; p++)            // KxK  filter
            for(int q = 0; q < K; q++)
              Y[m, h, w] += Y[m, h, w] + X[c, h + p, w + q] * W[m, c, p, q];
      }
}
```

Figure 16.4 A sequential implementation of the forward propagation path of a convolution layer.

Figure 16.4 shows a sequential implementation of the forward propagation path of a convolution layer. Each iteration of the outermost (*m*) for-loop generates an output feature map. Each of the next two levels (h and w) of for-loops generates one pixel of the current output feature map. The innermost three levels performs the 3D convolution between the input feature maps and the 3D filter banks.

Formatted: Font: Italic

The output feature maps of a convolution layer typically go through a subsampling (also known as pooling) layer. A subsampling layer reduces the size of image maps by combining pixels. For example, in Figure 16.2, subsampling layer S2 takes 6 input feature maps of size

[1] Note that W is used for both the width of images and the name of the filter bank matrix. In each case, the usage should be clear from the context.

28x28 and generates 6 feature maps of size 14x14. Each pixel in a subsampling feature map is generated from a 2x2 neighborhood in the corresponding input feature map. The values of these four pixels are averaged to form one pixel in the output feature map. The output of a subsampling layer has the same number of output feature maps as the previous layer, but each map has half the number of rows and columns. For example, the number of output feature maps (6) of the subsampling layer S2 is the same as the number of its input feature maps, or the output feature maps of the convolutional layer C1.

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
   int m, h, w, p, q;
   for(m = 0;  m < M;  m++)              // for each output feature maps
     for(h = 0; x h < H/K; h++)          // for each output element, this code assumes
       for(w = 0; y w < W/K; yw++) {     // that H and W are multiple sof K
         S[m, x, y] = 0.;
         for(p = 0; p < K; p++) {        // loop over KxK input samples
           for(q = 0; q < K; q++)
             S[m, h, w] += S[m, h, w] + Y[m, K*x h + p, K*y w+ q] /(K*K);
         }
         // add bias and apply non-linear activation
         S[m, h, w] = sigmoid(S[m, h, w] + b[m])
       }
}
```

Figure 16.5 A sequential C implementation of the forward propagation path of a subsampling layer.

Figure 16.5 shows a sequential C implementation of the forward propagation path of a subsampling layer. Each iteration of the outermost (m) for-loop generates an output feature map. The next two levels (h, w) of for-loops generates individual pixels of the current output map. The two innermost for-loops sum up the pixels in the neighborhood. K is equal to 2 in our LeNet-5 example in Figure 16.2. A bias value b[m] that is specific to each output feature map is then added to each output feature map and the sum goes through a non-linear function such as tanh, sigmoid or ReLU to give the output pixel values a more desirable distribution. ReLU is a very simple non-linear filter which passes only non-negative values:

$$Y = X, \text{ if } X \geq 0, \text{ and } 0 \text{ otherwise.}$$

To complete our example, convolutional layer C3 has 16 output feature maps, each of which is a 10x10 image. This layer has 6x16 filter banks and each filter bank has 5x5 weights. The output of C3 is passed through subsampling layer S4 which generates 16 5x5 output feature

maps. Finally, the last convolutional layer C5 which uses 16x120=1920 5x5 filter banks to generate 120 one-pixel output features from its 16 input feature maps.

These feature maps are passed through fully connected layer F6 which has 84 output units, where each output is fully connected to all inputs. The output is computed as a product of a weight matrix W with input vector X. For the F6 example, W is a 120x84 matrix then bias is added, and output is passed through sigmoid. In summary the output is an 84-element vector Y6 = sigmoid (W * X + b) assuming the implementation shown in Figure 16.2.

The final stage is an output layer that uses Gaussian filters to generate a vector of 10 elements, which correspond to the probability that input image contains one of 10 digits. It also computes *loss* functions which estimate the difference between true label and the prediction.

ConvNets: Back-propagation

Training of ConvNets is based on a procedure called gradient back-propagation. The training data set is labeled with the "correct answer." In the hand writing recognition example, the labels give the correct letter in the image. The label information can be used to generate the "correct" output of the last stage: the correct probability values of the 10-element vector.

For each training image, the final stage of the network calculates the loss function or the error as the difference between the generated output vector element values and the "correct" output vector element values. Given a sequence of training images, we can numerically calculate the gradient of the loss function with respect to the elements of the output vector. Intuitively, it gives the rate at which the loss function value changes when the values of the output vector elements change.

The back-propagation process starts by calculating the gradient of loss function dE/dY for the last layer. It then propagates the gradient from the last layer towards the first layer through all layers of network. Each layer receives as its input dE/dY – gradient with respect to its output feature maps and computes dE/dX – gradient with respect to its input feature maps.

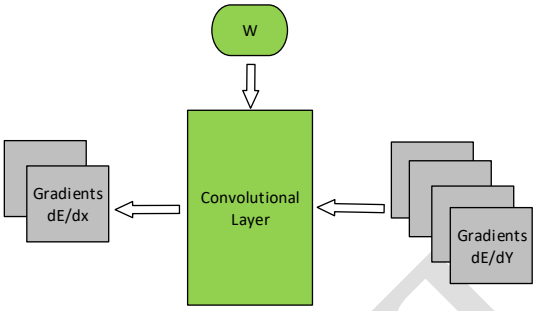


Figure 16.6: Convolutional layer: Back-propagation of dE/dX

If a layer has learned parameters ('weights') W, then the layer also computes dE/dW - gradient of loss with respect to weights:

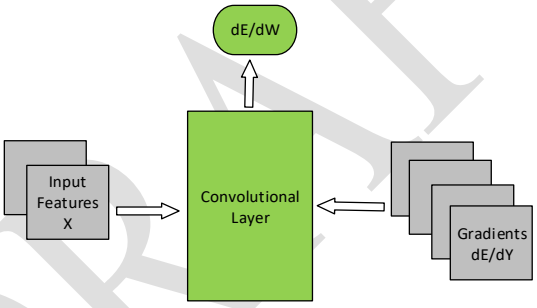


Figure 16.7 Convolutional layer: Back-propagation of dE/dw

For example the fully connected layer is given as: Y = W * X. The back-propagation of gradient dE/dY is given by two equations:

$$\frac{\partial E}{\partial X} = W^T * \frac{\partial E}{\partial Y} \text{ and } \frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y} * X^T$$

Let's describe now the back-propagation for a convolutional layer. We will start from the calculation of dE/dX.

Note that the calculation of dE/dX is important for propagation the gradient to the previous layer. The gradient dE/dX with respect to the channel *c* of input X is given as sum of "backward convolution" with corrsponfing $W^T(c,m)$ over all layer outputs *m:*

$$\frac{\partial E}{\partial X}(c,h,w) = \sum_{m=1}^{M}\sum_{p=1}^{K}\sum_{q=1}^{K}\left(W(p,q) * \frac{\partial E}{\partial Y}(h-p, w-q)\right)$$

Figure 16.8 shows the calculation of the dE/dX function in the form of one matrix for each input feature map. Note that the code assumes that dE/dY has been calculated for all the output feature maps of the layer and passed in with a pointer argument dE_dY. It also assumes that the space of dE/dX has been allocated in the device memory whose handle is passed in as a pointer argument. The kernel will be generating the elements of dE/dX.

```
void convLayer_backward_xgrad(int M, int C, int H_in, int W_in, int K,
        float* dE_dY, float* W, float* dE_dX)
{
  int m, c, h, w, p, q;
  int H_out = H_in – K + 1;
  int W_out = W_in – K + 1;
  for(c = 0;  c < C; c++)
   for(h = 0; h < H_in; h++)
    for(w = 0; w < W_in; w++)
      dE_dX[c, h, w] = 0.;

  for(m = 0;  m < M;  m++)
   for(h = 0; h < H_out; h++)
    for(w = 0; w < W_out; w++)
      for(c = 0;  c < C; c++)
        for(p = 0; p < K; p++)
          for(q = 0; q < K; q++)
            dE_dX[c, h + p, w + q] += dE_dY[m, h, w] * W[m, c, p, q];
}
```

Figure 16.8 dE/dX calculation of the backward path of a convolution layer.

The algorithm for calculating dE/dW for a convolution layer computation is very similar to that of dE/dX and is shown in Figure 16.9. Since each W(~~e~~m,~~m~~c) affects all elements of output Y(m), we should accumulate gradients over all pixels in the corresponding output feature map:

$$\frac{\partial E}{\partial W}(c,m;p,q) = \sum_{h=1}^{H_{out}} \sum_{w=1}^{W_{out}} (X(h+p, w+q) * \frac{\partial E}{\partial Y}(h,w))$$

Note that while the calculation of dE/dX is important for propagating the gradient to the previous layer, the calculation of the dE/dW is key to the adjustments to the weight values of the current layer.

```
void convLayer_backward_wgrad(int M, int C, int H, int W, int K,
        float* dE_dY, float* X, float* dE_dW)
{
```

```
  int m, c, h, w, p, q;
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  for(m = 0; m < M; m++)
   for(c = 0; c < C; c++)
     for(p = 0; p < K; p++)
       for(q = 0; q < K; q++)
         dE_dW[m, c, p, q] = 0.;

  for(m = 0;  m < M;  m++)
   for(h = 0; h < H_out; h++)
     for(w = 0; w < W_out; w++)
       for(c = 0;  c < C;  c++)
         for(p = 0; p < K; p++)
           for(q = 0; q < K; q++)
             dE_dW[m, c, p, q] += X[c, h + p, w + q] * dE_dY[m, c, h, w];
}
```

Figure 16.9 dE/dW calculation of the backward path of a convolutional layer.

After the dE/dW values at all feature map element positions are computed, weights are updated iteratively to minimize the expected error: $W(t+1) = W(t) - \lambda * dE/dW$, where λ is a constant called the learning rate. The initial value of λ is set empirically and reduced through the iterations according to the rule defined by user The value of λ is reduced through the iterations to ensure that the convergence to a minimal error. The negative sign of the adjustment term makes the change opposite to the direction of the gradient so that the change will likely reduce the error. Recall that the weight values of the layers determine the behavior the network: they determine how the input is transformed through the network. This adjustment of these weight values of all the layers adapts the behavior of the network. That is, the network "learns" from a sequence of labeled training data and adapts its behavior by adjusting all weight values at all its layers.

The training data sets are usually large, so the training of ConvNets is typically done using Stochastic Gradient Descent: instead of doing forward-backward step to compute dE/dW for the whole training data set, one randomly selects a small subset ('mini-batch') of N images from training data set, and computes the gradient only for this subset. Next, one will select another subset etc[2]. This adds one additional dimension to all data arrays with n - the index of sample in the mini-batch. It also adds one additional loop over samples:

[2] If we would work by "optimization book" we should return samples back to the training set, and then build new mini-batch by randomly picking next samples. In practice we go sequentially over whole training set. In machine learning it's called epoch. Then we shuffle the whole training set, and start the next epoch.


```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
   int n, m, c, h, w, p, q;
   int H_out = H – K + 1;
   int W_out = W – K + 1;
   for(n = 0;  n < N;  n++)                  // for each sample in the mini-batch
     for(m = 0;  m < M;  m++)                // for each output feature maps
       for(h = 0; h < H_out; h++)            // for each output element
         for(w = 0; w < W_out; w++) {
           Y[n, m, h, w] = 0;
           for (c = 0;  c < C; c++)          // sum over all input feature maps
             for (p = 0; p < K; p++)         // KxK  filter
               for (q = 0; q < K; q++)
                 Y[n, m, h, w] = Y[n, m, h, w] + X[n, c, h + p, w + q] * W[m, c, p, q];
         }
}
```

Figure 16.10 Forward path of a convolutional layer with mini-batch training.

Figure 16.10 shows the revised forward path implementation of a convolutional layer. It generates the output feature maps for all the samples of a mini-batch. During back-propagation, one first computes the average gradient of the error with respect to the weights of last layer over all samples in mini-batch. The gradient is then propagated backward through the layers and used to adjust all the weights. Each iteration of the weight adjustment processes one mini-batch. The training typically measured in epochs, where one epoch is a sequential pass over all the samples in the training data set. The training data set is typically reshuffled between epochs.

16.3 Convolutional layer – a basic CUDA implementation of forward propagation

The computation pattern in training a convolutional network is very similar to matrix multiplication: it is both compute intensive and highly parallel. We can compute in parallel different samples in a mini-batch, different output feature maps for the same sample, and different elements for each output feature map. Figure 16.11 shows a conceptual parallel code for the forward path of a convolutional layer. Each parallel_for loop indicates that all its iterations can be executed in parallel.

```
void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
   int n, m, c, h, w, p, q;
   int H_out = H – K + 1;
   int W_out = W – K + 1;
   parallel_for(n = 0;  n < N;  n++)
```

```
    parallel_for (m = 0;  m < M;  m++)
     parallel_for(h = 0; h < H_out; h++)
      parallel_for(w = 0; w < W_out; w++) {
       Y[n, m, h, w] = 0;
       for (c = 0;  c < C; c++)
        for (p = 0; p < K; p++)
         for (q = 0; q < K; q++)
          Y[n, m, h, w] = Y[n, m, h, w] + X[n, c, h + p, w + q] * W[m, c, p, q];
     }
}
```

Figure 16.11 Parallelization of the forward path of a convolutional layer with min-batch training.

As shown in Figure 16.11, there are four levels of parallelism. The total number of parallel iterations is the product N*M*H_out*W_out. This very high degree of available parallelism makes ConvNets an excellent candidate for GPU acceleration. As an example, let's implement the forward path for convolutional layer.

Let's refine the high-level parallel code into a kernel by making some high-level design decisions. Assume that we will have each thread to compute one element of one output feature map. We will use 2D thread blocks, where each thread block computes a tile of TILE_WIDTH x TILE_WIDTH elements in one output feature map. For example, if we set TILE_WIDTH = 16, we would have a total 256 threads per block. Blocks will be organized into a 3D grid:

1) the first dimension (X) in the grid corresponds to samples (N) in the batch
2) the second dimension (Y) corresponds to the (M) output features maps
3) the last dimension (Z) will define the location of the output tile inside the output feature map.

The last dim Z depends on the number of tiles in the horizontal and vertical dimensions of the output image. Assume for simplicity that H_out (height of the ouput image) and W_out (width of the output image) are multiples of 16, the tile width:

```
# define TILE_WIDTH 16
W_grid = W_out/TILE_WIDTH;    // number of horizontal tiles per output map
H_grid = H_out/TILE_WIDTH;    // number of vertical tiles per output map
Z = H_grid * W_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(N, M, Z);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(…);
```

As we discussed before, each thread block is responsible for computing one 16x16 tile in the output Y(n, c, ., .)., and each thread will compute one element Y[n, m, h, w] where

```
n = blockIdx.x;
m = blockIdx.y;
h = blockIdx.z / W_grid + threadIdx.y;
w = blockIdx.z % W_grid + threadIdx.x;
```

This result is the kernel shown in Figure 16.12. Note that in the code above we use multi-dimensional index in arrays. We leave it to reader to translate this pseudo-code into regular C assuming that X, Y, and W must be accessed via linearized indexing based on row-major layout (Chapter 3).

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
  int n, m, h, w, c, p, q;
  n = blockIdx.x;
  m = blockIdx.y;
  h = blockIdx.z / W_grid + threadIdx.y;
  w = blockIdx.xz % W_grid + threadIdx.x;
  float acc = 0.;
  for (c = 0;  c < C; c++) {          // sum over all input channels
   for (p = 0; p < K; p++)            // loop over KxK  filter
     for (q = 0; q < K; q++)
       acc += acc + X[n, c, h + p, w + q] * W[m, c, p, q];
  }
  Y[n, m, h, w] = acc;
}
```

Figure 16.12 Kernel for the forward path of a convolution layer.

The kernel in Figure 16.12 has a very high degree of parallelism but consumes too much global memory bandwidth. Like the based convolution pattern, the execution speed of the kernel will be limited by the global memory bandwidth. As we have seen in Chapter 7, we can use shared memory tiling to dramatically improve the execution speed of the kernel. Let's now modify the basic kernel to reduce traffic to global memory. The reader should review the tiling sections of Chapter 7 before proceeding. The kernel is shown in Figrure 16.13. The basic design is stated in the comments and outlined below:

1. Load the filter W[m, c] into the shared memory

2. All threads collaborate to copy the portion of the input X[n,c,.,.] that is required to compute the output tile into the shared memory array X_shared
3. Compute partial sum of output Y_shared[n, m,.,.]
4. Move to the next input channel c

We need to allocate shared memory for the input block X_tile_width * X_tile_width, where X_tile_width = TILE_WIDTH + K-1. In addition we also need to allocate shared memory for K*K filter coefficients. So the total amount of shared memory will be (TILE_WIDTH + K-1)* (TILE_WIDTH + K-1)+ K*K . Since we do not know K at compile time we need to add it to the kernel definition as the third parameter.

```
…
size_t shmem_size = sizeof(float) * ( (TILE_WIDTH + K-1)*(TILE_WIDTH + K-1) + K*K );
ConvLayerForward_Kernel<<< gridDim, blockDim, shmem_size>>>(…);
…
```

We will divide the shared memory between input buffer and filter inside the kernel. The first X_tile_width* X_tile_width entries are allocated to the input tiles and the rest of the entries are allocated to the weight values.

```
__global__ void
ConvLayerForward_Kernel(int C, int W_grid, int K, float* X, float* W, float* Y)
{
   int n, m, h0, w0, h_base, w_base, h, w;
   int X_tile_width = TILE_WIDTH + K-1;
   extern __shared__ float shmem[];
   float* X_shared = &shmem[0];
   float* W_shared = &shmem[X_tile_width * X_tile_width];
   n = blockIdx.x;
   m = blockIdx.y;
   h0 = threadIdx.x;
   w0 = threadIdx.y;
   h_base = (blockIdx.z / W_grid) * TILE_SIZE;   // vertical base out data index for the block
   w_base = (blockIdx.z % W_grid) * TILE_SIZE; // horizontal base out data index for the block
   h = h_base + h0;
   w = w_base + w0;

   float acc = 0.;
   int c, j, k, p, q;
   for (c = 0; c < C; c++) {                    // sum over all input channels
                                                // load weights for W [m, c,..],
                                                // h0 and w0 used as shorthand for threadIdx.x
                                                // and threadIdx.y
     if (( h0 < K) && ( w0 < K))
```

```
      W_shared[h0, w0]= W [m, c, h0, w0];
     __syncthreads();
                                        // load tile from X[n, c,…] into shared memory

     for (int i = h; i < h_base + X_tile_width; i += TILE_WIDTH) {
       for (int j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
         X_shared[i - h_base, j - w_base] = X[n, c, h, w]
     }
     __syncthreads();
     for (p = 0; p < K; p++) {
       for (q = 0; q < K; q++)
         acc = acc + X_shared[h + p, w + q] * W_shared[p, q];
     }
     __syncthreads();
   }
   Y[n, m, h, w] = acc;
}
```

Figure 16.13 A kernel that uses shared memory tiling to reduce the global memory traffic of the forward path of the convolutional layer

The use of shared memory tiling results in a very high level of acceleration in the execution of the kernel. The analysis is similar to that discussed in Chapter 7 and is left as an exercise.

16.4 Reduction of convolutional layer to matrix multiplication

We can build an even faster convolutional layer by reducing it to matrix multiplication and then using highly efficient matrix multiplication GEMM from the CUDA linear algebra library (cuBLAS). This method was proposed by Chellapilla K., Puri S., Simard P. [9]. The central idea is unfolding and duplicating of the inputs to the convolutional kernel in such way that all elements needed to compute one output element will be stored as one sequential block. This will reduce the forward operation of the convolutional layer to one large GEMM matrix-matrix multiplication [3].

Consider for small example a convolutional layer which takes as input C=3 feature maps of 3x3 and produces M=2 output features 2x2. It uses M*C = 6 filter banks, where each filter bank is 2x2. The matrix version of this layer will be constructed in the following way:

First we will rearrange all input elements. Since the results of the convolutions are summed across input features, the input features can be concatenated into one large matrix. Each row

[3] See also https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/ for very detailed explanation

of this matrix contains all the input values necessary to compute one element of an output feature. This means that each input element will be replicated multiple times. For example, the center of each 3x3 input feature is used four times to compute each element of an output feature, so it will be duplicated 4 times. The middle element on each edge is used two times so it will be duplicated two times. The four elements at corners of each input feature is used only one time and will not need to be duplicated. Therefore, the total number of elements in the expanded input feature matrix is 4*1 + 2*4 + 1*4 = 16.

In general, the size of the expanded (unrolled) input feature map matrix can be derived by thinking about the number of input feature map elements required to generate each output feature map element. In general, the height, or the number of rows, of the expanded matrix is the number of input feature elements contributing to each output feature map element. The number is C*K*K: each output element is the convolution of K*K elements from each input feature map and there are C input feature maps. In our example, the K is 2 since the filter bank is 2x2 and there are three input feature maps. Thus the height of the expanded matrix should be 3*2*2 = 12, which is exactly the height of the matrix shown in Figure 16.14.

The width, or the number columns, of the expanded matrix should be the number of elements in each output feature map. Assuming that the output feature maps are H_out x W_out matrices, the number of columns of the expanded matrix is H_out*W_out. In our example, note that the each output feature map is a 2x2 matrix so there are four columns in the expanded matrix. Note that the number of output feature maps M does not play into the duplication. This is because all output feature maps share the same expanded matrix.

The ratio of expansion for the input feature maps is the size of the expanded matrix over the total size of the original input feature maps. The reader should verify that the expansion ratio is (K*K*H_out*W_out)/(H_in*W_in), where H_in and W_in are the height and width of each input feature map. In our example, the ratio is (2*2*2*2)/(3*3)=16/9. In general, if the input feature maps and output feature maps are much larger than the filter banks, the ratio will approach K*K.

The filter banks are represented as a filter-bank matrix in a fully linearized layout, where each row contains all weight values that are needed to produce one output feature map. The height of the filter-bank matrix is the number of output feature maps (M). The height of the filter-bank matrix allows the output feature maps to share a signle expanded input matrix. The width of the filter-bank matrix is the number of weight values needed for generating each output feature map element, which is `C*K*K`. Note that there is no duplication when placing the weight values into the filter-banks matrix. In our example, the filter-bank matrix is simply a linearized arrangement of the six filter-banks.

When we multiply the filter-bank matrix W by the expanded input matrix X_unrolled the output features Y are computed as one large matrix of height M and width `H_out*W_out`.

Let's discuss now how we can implement this algorithm in CUDA. Let's first discuss the data layout. We can start from the layout of the input and output matrices.

- We assume that the input feature map samples in a mini-batch will be supplied in the same way as that for the basic CUDA kernel. It is organized as an N x C x H x W array, where N is the number of samples in a mini-batch, C is the number of input feature maps, H is the height of each input feature map, and W is the width of each input feature map.

- As we showed in Figure 16.14, the matrix multiplication will naturally produce an output Y stored as an M x H_out*W_out array. This is what the original basic CUDA kernel would produce.

- Since the filter-bank matrix does not involve duplication of weight values, we assume that it will be prepared as ahead of time and organized as an M x C x (K x K) array as illustrated in Figure 16.14.

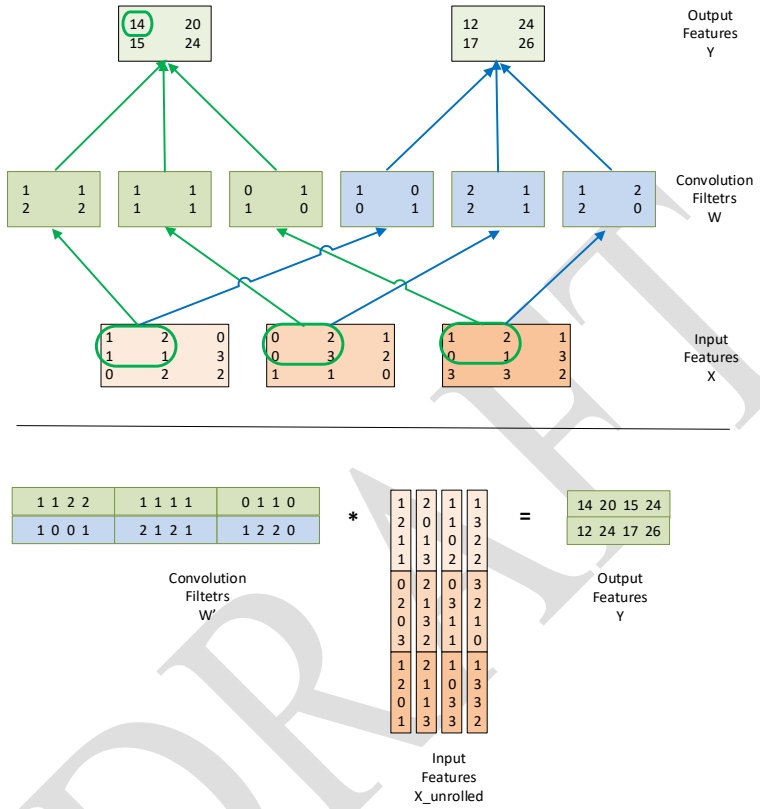


Figure 16.14: Reduction of convolutional layer to GEMM

The preparation of the expanded input feature map matrix X_unroll is more complex. Since each expansion increases the size of input by approximately up to K*K times, the expansion ratio can be very large for typical K values of 5 or larger. The memory footprint for keeping all sample input feature maps for a mini-batch can be prohibitively large. To reduce the memory footprint, we will allocate only one buffer for X_unrolled [C * K * K* H_out * W_out]. We will reuse this buffer by adding loop over samples in the batch. During each iteration, we convert the simple input feature map from its original form into the expanded matrix.

void convLayer_forward(int N, int M, int C, int H, int W, int K, float* X, float* W_unroll, float* Y)

```
{
  int W_out = W – K + 1;
  int H_out = H – K + 1;
  int W_unroll = C * K * K;
  int H_unroll = H_out * W_out;
  float* X_unrolled = malloc(W_unroll * H_unroll * sizeof(float));
  for (int n=0; n < N; n++) {
   unroll C, H, W, K, n, X, X_unrolled);
   gemm(H_unroll, M, W_unroll, X_unrolled, W, Y[n]);
  }
}
```

Figure 16.15 Implementing the forward path of a convolutional layer with matrix multiplication.

Figure 16.15 shows the sequential implementation of the forward path of a convolutional layer with matrix multiplication. The code loops through all samples in the batch.

Figure 16.16 shows a sequential function that produces the X_unroll array by gathering and duplicating the elements of an input feature map X. The function uses five levels of loops. The innermost two levels of for-loop (w and h) place one input feature map element for each of the output feature map elements. The next two levels repeat the process for each of the K*K input feature map elements for the filtering operations. The outermost loop repeats the process of all input feature maps. This implementation is conceptually straightforward and can be quite easily parallelized since the loops do not impose dependencies among their iterations. Also, successive iterations of the innermost loop reads from a localized tile of one of the input feature maps in X and write into sequential locations in the expanded matrix X_unrolled. This should result in efficient memory bandwidth usage on a CPU.

```
void unroll(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int c, h, w, p, q, w_base, w_unroll, h_unroll;
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  for(c = 0; c < C; c++) {
   w_base = c * (K*K);
   for(p = 0; p < K; p++)
    for(q = 0; q < K; q++) {
     for(h = 0; h <  H_out; h++)
      for(w = 0; w < W_out; w ++) {
        w_unroll = w_base + p * K + q;
        h_unroll = h * W_out + w;
        X_unroll(h_unroll, w_unroll) = X(c, h + p, w + q);
```

```
        }
    }
}
```

Figure 16.16 The function that generates the unrolled X matrix.

We are now ready to design a CUDA kernel which implements the input feature map unrolling. Each CUDA thread will be responsible for gathering (K*K) input elements from one input feature map for one element of output feature map. The total number of threads will be (C * H_out * W_out). We will use one-dimensional blocks. If we assume that a maximum number of threads per block is CUDA_MAX_NUM_THREADS (e.g. 1,024), the total number of blocks in the grid will be num_blocks = ceil((C*H_out*W_out) / CUDA_MAX_NUM_THREADS).

```
void unroll_gpu(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  int num_threads = C * H_out * W_out;
  int num_blocks = ceil((C * H_out * W_out) / CUDA MAX_NUM_THREADS);
  unroll_Kernel<<<num_blocks, CUDA MAX_NUM_THREADS>>>();
}
```

Figure 16.17 Host code for invoking the unroll kernel.

Figure 16.18 shows an implementation of the unroll kernel. Note that each thread will build a K*K section of a column, shown as a shaded box in the Input Features X_Unrolled array in Figure 16.14. Each such section contains all elements of input feature map X from channel c, required for convolution with corresponding filter to produce one element of output Y.

```
__global__ void unroll_Kernel(int C, int H, int W, int K, float* X, float* X_unroll)
{
  int c, s, h_out, w-out, h_unroll, w_base, p, q;
  int t = blockId.x * CUDA MAX_NUM_THREADS + threadId.x;
  int H_out = H – K + 1;
  int W_out = W – K + 1;
  int W_unroll = H_out * W_out;

  if (t < C * W_unroll) {
   c = t / W_unroll;
   s = t % W_unroll;
   h_out = s / W_out;
```

```
    w_out = s % W_out;
    h_unroll = h_out * W_out + w_out;
    w_base = c * K * K;
    for(p = 0; p < K; p++)
     for(q = 0; q < K; q++) {
       w_unroll = w_base + p * K + q;
       X_unroll(h_unroll, w_unroll) = X(c, h_out + p, w_out + q);
      }
   }
}
```

Figure 16.18 A high-performance implementation of the unroll kernel.

Comparing the loop structure of Figure 16.18 and Figure 16.16 shows that the innermost two loop levels in Figure 16.16 have been exchanged into outer level loops. Having each thread to collect all input feature map elements from an input feature map needed for generating an output generates a coalesced memory write pattern. As shown in Figure 16.16, adjacent threads will be writing adjacent X_unroll elements in a row as they all move vertically to complete their sections. The read access patterns to X are similar. We leave the analysis of the read access pattern as an exercise.

An important high-level assumption is that we keep the input feature maps, filter bank weights, and output feature maps in the device memory. The filter-bank matrix is prepared once and stored in the device global memory for use by all input feature maps. For each sample in the mini-batch, we launch the unroll_Kernel to prepare an expanded matrix and launch a matrix multiplication kernel, as outlined in Figure 16.15.

Implementing convolutions with matrix multiplication can be very efficient, since matrix multiplication is highly optimized on all hardware platforms. Matrix multiplication is especially fast on GPUs because it has a high ratio of floating-point operations per byte of global memory data access. This ratio increases as the matrices get larger, meaning that matrix multiplication is less efficient on small matrices. Accordingly, this approach to convolution is most effective when it creates large matrices for multiplication.

As we mentioned earlier, the filter-bank matrix is an M x C*K*K matrix and the expanded input feature map matrix is a C*K*K x H_out*W_out matrix. Note that except for the height of the filter-bank matrix, the sizes of all dimensions depend on products of the parameters to the convolution, not the parameters themselves. While individual parameters can be small, their products tend to be large. This means that size of the matrices tends to be consistently large and thus the performance using this approach can be very consistent. For example, it is often true that in early layers of a convolutional network, C is small, but H_out and W_out are large. On the other hand, at the end of the network, C is large, but H_out and W-out

attend to be small. However, the product C*H_out*W_out is usually fairly large for all layers, so performance can be consistently good.

The disadvantage of forming the expanded input feature map matrix is that it involves duplicating the input data up to K*K times, which can require a prohibitively large temporary allocation. To work around this, implementations such as the one shown in Figure 16.15 materialize X_unroll matrix piece by piece, for example, by forming the expanded input feature map matrix and calling matrix multiplication iteratively for each sample of the mini-batch. However, this limits the parallelism in the implementation, and can sometimes lead to cases where the matrix multiplications are too small to effectively utilize the GPU. This approach also lowers the computational intensity of the convolutions, because X_unroll must be written and read, in addition to reading X itself, requiring significantly more memory traffic as a more direct approach. Accordingly, the highest performance implementation has even more complex arrangements in realizing the unrolling algorithm to both maximize GPU utilization while keeping the reading from DRAM minimal. We will come back to this point when we present the CUDNN approach in next section.

16.5 CUDNN Library

CUDNN is a library of optimized routines for implementing deep learning primitives. It was designed to make it much easier for deep learning frameworks to take advantage of GPUs. It provides a flexible, easy-to-use C-language deep learning API that integrates neatly into existing frameworks (Caffe, Tensorflow, Theano, Torch,…). The library requires that input and output data be resident in the GPU device memory as we discussed in the previous section. This requirement is analogous to that of cuBLAS.

The library is thread-safe and its routines can be called from different host threads. Convolutional routines for the forward and backward paths use a common descriptor that encapsulates the attributes of the layer. Tensors and filters are accessed through opaque descriptors, with the flexibility to specify the tensor layout using arbitrary strides along each dimension. The most important computational primitive in convolutional neural networks is a special form of batched convolution. In this section, we describe the forward form of this convolution. The CUDNN parameters governing this convolution are listed Table 16.1.

Parameter	Meaning
N	Number of images in mini-batch
C	Number of input feature maps
H	Height of input image
W	Width of input image
K	Number of output feature maps
R	Height of filter
S	Width of filter
u	Vertical stride

v	Horizontal stride
pad_h	Height of zero-padding
Pad_w	Width of zero-padding

Table 16.1: Convolution parameters for CUDNN. Note that the CUDNN naming convention is slightly different than what we have been using in previous sections.

There are two inputs to the convolution:

- D is a four-dimensional N x C x H x W tensor, which forms the input data, [4]
- F is a four-dimensional K x C x R x S tensor, which forms the convolutional filters.

The input data array (tensor) D ranges over N samples in a mini-batch, C input feature maps per sample, H rows per input feature map, and W columns per input feature map. The filters range over K output feature maps, C input feature maps, R rows per filter bank, and S columns per filter bank. The output is also a four-dimensional tensor O that ranges over N samples in the mini-batch, K output feature maps, P rows per output feature map, and Q columns per output feature map, where P = f(H;R; u; pad_h), Q = f(W; S; v; pad_w), meaning that the height and width of the output feature maps depend on the input feature map and filter bank height and width, along with padding and striding choices. The striding parameters u and v allow the user to reduce the computational load by computing only a subset of the output pixels. The padding parameters allow users to specify how many rows or columns of 0 entries are appended to each feature map for improved memory alignment and/or vectorized execution.

CUDNN supports multiple algorithms for implementing a convolutional layer: matrix-multiplication-based (GEMM and Winograd [13]), FFT-based [10], etc. The GEMM-based algorithm to implement the convolutions with a matrix multiplication, is similar to the approach presented in Section 16.4. As we discussed at the end of Section 16.4 materializing the expanded input feature matrix in global memory can be costly in both the global memory space and bandwidth consumption. CUDNN avoids this problem by lazily generating and loading the expanded input feature map matrix X_unroll into on-chip memory only, rather than by gathering it in off-chip memory before calling a matrix multiplication routine. NVIDIA provides a matrix multiplication based routine that achieves a high utilization of maximal theoretical floating point throughput on GPUs. The algorithm for this routine is similar to the algorithm described in [12] Fixed sized sub-matrices of the input matrices A and B are successively read into on-chip memory and are then used to compute a sub-matrix of the output matrix C. All indexing complexities imposed by the convolution are handled in the management of tiles in this routine. We compute on tiles of A and B while fetching the next tiles of A and B from off-chip memory into on-chip caches and other memories. This technique hides the memory latency associated with the data transfer, allowing the

[4] Tensor is a mathematical term for arrays that have more than two dimensions. In mathematics, matrices have only two dimensions. Arrays with three or more dimensions are called tensors. For the purpose of this book, one can simply treat a T-dimensional tensor as a T-dimensional array.

matrix multiplication computation to be limited only by the time it takes to perform the arithmetic

Since the tiling required for the matrix multiplication routine is independent of any parameters from the convolution, the mapping between the tile boundaries of X_unroll and the convolution problem is non-trivial. Accordingly, the CUDNN approach entails computing this mapping and using it to load the correct elements of A and B into on-chip memories. This happens dynamically as the computation proceeds, which allows the CUDNN convolution implementation to exploit optimized infrastructure for matrix multiplication. It requires additional indexing arithmetic compared to a matrix multiplication, but fully leverage the computational engine of matrix multiplication to perform the work. After the computation is complete, CUDNN performs the required tensor transposition to store the result in the user's desired data layout.

.

16.6 Exercises

1. Implement the forward path for the pooling layer described in the section 16.2.
2. We used an [N x C x H x W] layout for input and output features. Can we reduce the memory bandwidth by changing it to an [N x H x W x C]. What are potential benefits of [C x H x W x N] layout?
3. It is possible to implement the convolutional layer using FFT using the schema described in [10].
4. Implement the backward path for the convolutional layer described in the section 16.2.
5. Analyze the read access pattern to X in the unroll_kernel in Figure 16.18 and show whether the memory reads done by adjacent threads can be coalesced.

References

1. LeCun, Y., Bengio, Y., & Hinton, G. E., (2015) *Deep learning*, Nature 521, 436–444 (28 May 2015) http://www.nature.com/nature/journal/v521/n7553/full/nature14539.html
2. Rumelhart, D. E., Hinton, G. E., & Wiliams R. J. (1986). *"Chapter 8 : Learning Internal Representations by Error Propagation"*. In Rumelhart, D. E.; McClelland, J.L. *Parallel Distributed Processing*, Volume 1, MIT Press. pp. 319–362
3. LeCun, Y., Bottou L., Bengio,Y., & Haffner P., (1998). "*Gradient-based learning applied to document recognition*" . Proceedings of the IEEE 86 (11): 2278–2324, http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf

4. LeCun, Y. et al. *Handwritten digit recognition with a back-propagation network.* In Proc. Advances in Neural Information Processing Systems 396–404 (1990). http://yann.lecun.com/exdb/publis/pdf/lecun-90c.pdf
5. Hinton, G. E., Osindero, S. & Teh, Y.-W. *A fast learning algorithm for deep belief nets*. Neural Comp. 18, 1527–1554 (2006) https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf
6. Raina, R., Madhavan, A. & Ng, A. Y. *Large-scale deep unsupervised learning using graphics processors*. In Proc. 26th ICML 873–880 (2009). http://www.andrewng.org/portfolio/large-scale-deep-unsupervised-learning-using-graphics-processors/
7. Krizhevsky, A., Sutskever, I. & Hinton, G. *ImageNet classification with deep convolutional neural networks*. In Proc. Advances in NIPS 25 1090–1098 (2012). https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
8. Krizhevsky, A., cuda-convnet https://code.google.com/p/cuda-convnet/
9. Chellapilla K., Puri S., Simard P., High Performance Convolutional Neural Networks for Document Processing , 2006 https://hal.archives-ouvertes.fr/inria-00112631/document
10. Vasilache N., Johnson J., Mathieu M., Chintala S., Piantino S, LeCun Y. , *Fast Convolutional Nets With fbfft: A GPU Performance Evaluation, 2014* http://arxiv.org/pdf/1412.7580v3.pdf
11. Chetlur S., Woolley C., Vandermersch P., Cohen J., Tran J., *cuDNN: Efficient Primitives for Deep Learning NVIDIA,* 2014
12. Tan G., Li L., Treichler S., Phillips E., Bao Y., Sun N, *Fast implementation of DGEMM on Fermi GPU. In Supercomputing 2011*, SC '11, 2011
13. Lavin A., Gray S., Fast Algorithms for Convolutional Neural Networks http://arxiv.org/abs/1509.09308

Chapter 18

Programming a Heterogeneous Computing Cluster

With special contributions from Isaac Gelado

Keywords: MPI, message passing, communication, overlapping communication with computation, asynchronous, domain partition, collective, point-to-point communication, pinned memory, CUDA streams, barrier

CHAPTER OUTLINE

So far, we have focused on programming a heterogeneous computing system with one host and one device. In High-Performance Computing (HPC), applications require the aggregate computing power of a cluster of computing nodes. Many of the HPC clusters today have one or more hosts and one or more devices in each node. Historically, these clusters have been programmed predominately with Message Passing Interface (MPI). In this chapter, we will present an introduction to joint MPI/CUDA programming. The reader should be able to easily extend the material to joint MPI/OpenCL, MPI/OpenACC, and so on. We will only present the MPI concepts that a programmer needs to understand in order to scale their heterogeneous applications to multiple nodes in a cluster environment. In particular, we will

focus on domain partitioning, point-to-point communication, and collective communication in the context of scaling a CUDA kernel into multiple nodes.

18.1. Background

While there was practically no top supercomputer using GPUs before 2009, the need for better energy efficiency has led to fast adoption of GPUs in the recent years. Many of the top supercomputers in the world today use both CPUs and GPUs in each node. The effectiveness of this approach is validated by their high rankings in the Green500 list, which reflects their high energy efficiency.

The dominating programming interface for computing clusters today is Message Passing Interface (MPI) [Gropp1999], which is a set of API functions for communication between processes running in a computing cluster. MPI assumes a distributed memory model where processes exchange information by sending messages to each other. When an application uses API communication functions, it does not need to deal with the details of the interconnect network. The MPI implementation allows the processes to address each other using logical numbers, much the same way as using phone numbers in a telephone system: telephone users can dial each other using phone numbers without knowing exactly where the called person is and how the call is routed.

In a typical MPI application, data and work are partitioned among processes. As shown in Figure 18.1, each node can contain one or more processes, shown as clouds within nodes. As these processes progress, they may need data from each other. This need is satisfied by sending and receiving messages. In some cases, the processes also need to synchronize with each other and generate collective results when collaborating on a large task. This is done with collective communication API functions.

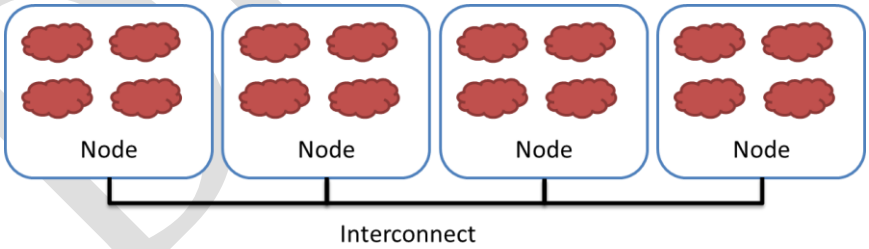


Figure 18.1 Programmer's view of MPI Processes.

18.2 A Running Example

We will use a 3D stencil computation introduced in Chapter 7 as a running example. We assume that the computation calculates heat transfer based on a finite difference method for solving a partial differential equation that describes the physical laws of heat transfer. In particular, we will use the Jacobi Iterative Method where in each iteration or time step, the value of a grid point is calculated as a weighted sum of neighbors (north, east, south, west,

up, down) and its own value from the previous time step. In order to achieve high numerical stability, multiple indirect neighbors in each direction are also used in the computation of a grid point. This is referred to as a *higher order stencil* computation. For the purpose of this chapter, we assume that four points in each direction will be used.

As shown in Figure 18.2, there are a total of 24 neighbor points for calculating the next step value of a grid point. As shown in Figure 18.2, each point in the grid has an x, y, and z coordinate. For a grid point where the coordinate value is x=i, y=j, and z=k, or (i,j,k) its 24 neighbors are (i-4,j,k), (i-3,j,k), (i-2,j,k), (i-1,j,k), (i+1,j,k), (i+2,j,k), (i+3,j,k), (i+4,j,k), (i,j-4,k), (i,j-3,k), (i,j-2,k), (i,j-1,k), (i,j+1,k), (i,j+2,k), (i,j+3,k), (i,j+4,k), (i,j,k-4), (i,j,k-3), (i,j,k-2), (i,j,k-1), (i,j,k+1), (i,j,k+2), (i,j,k+3) and (i,j,k+4). Since the data value of each grid point for the next time step is calculated based on the current data values of 25 points (24 neighbors and itself), the type of computation is often called 25-stencil computation.

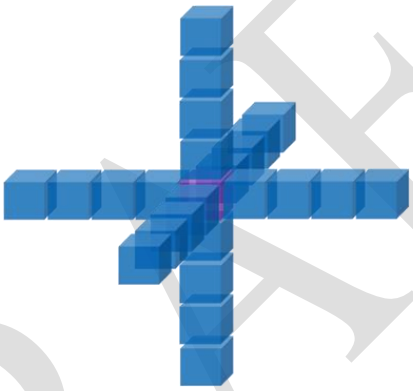


Figure 18.2 A 25-stencil computation example, with neighbors in the x, y, z directions.

We assume that the system is modeled as a structured grid, where spacing between grid points is constant within each direction. This allows us to use a 3D array where each element stores the state of a grid point. The physical distance between adjacent elements in each dimension can be represented by a spacing variable. Note that this grid data structure is similar to that used in the electrostatic potential calculation in Chapter 15. Figure 18.3 illustrates a 3D array that represents a rectangular ventilation duct, with x and y dimensions as the cross sections of the duct and the z dimension the direction of the heat flow along the duct.

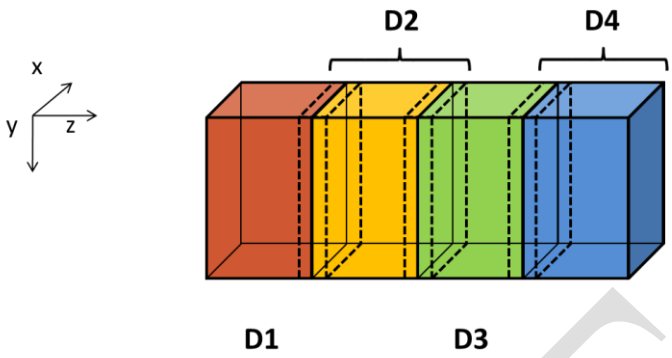


Figure 18.3 3D Grid array for the modeling heat transfer in a duct.

We assume that the data is laid out in the memory space so that x is the lowest dimension, y is the next, and z is the highest. That is, all elements with y=0 and z=0 will be placed in consecutive memory locations according to their x coordinate. Figure 18.4 shows a small example of the grid data layout. This small example has only 16 data elements in the grid: two elements in the x dimension, two in the y dimension, and four in the z dimension. Both x elements with y=0 and z=0 are placed in memory first. They are followed by all elements with y=1 and z=0. The next group will be elements with y=0 and z=1. The reader should verify that this is simply a 3D generalization of the row-major layout convention of C/C++ discussed in Chapter 3.

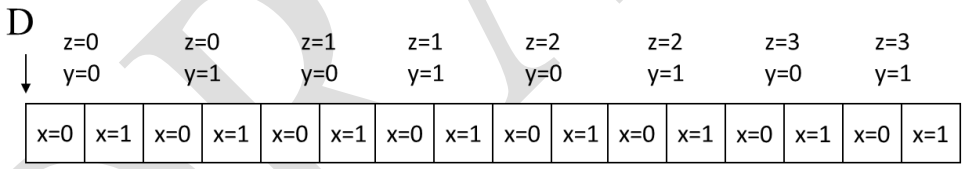


Figure 18.4 A small example of memory layout for the 3D grid.

When one uses a computing cluster, it is common to divide the input data into several partitions, called domain partitions, and assign each partition to a node in the cluster. In Figure 18.3, we show that the 3D array is divided into four domain partitions: D0, D1, D2, and D3. Each of the partitions will be assigned to an MPI compute process.

The domain partitions can be further illustrated with Figure 18.4. The first section, or slice, of four elements (z=0) in Figure 18.4 are in the first partition, the second section (z=1) the second partition, the third section (z=2) the third partition, and the fourth section (z=3) the fourth partition. This is obviously a toy example. In a real application, there are typically hundreds or even thousands of elements in each dimension. For the rest of this chapter, it is useful to remember that all elements in a z slice are in consecutive memory locations.

18.3 MPI Basics

Like CUDA, MPI programs are based on the SPMD parallel execution model. All MPI processes execute the same program. The MPI system provides a set of API functions to establish communication systems that allow the processes to communicate with each other. Figure 18.5 shows five essential MPI functions that set up and tear down the communication system for an MPI application. We will use a simple MPI program shown in Figure 18.6 to illustrate the usage these API functions. To launch an MPI application in a cluster, a user needs to supply the executable file of the program to the *mpirun* command or the *mpiexec* command in a cluster.

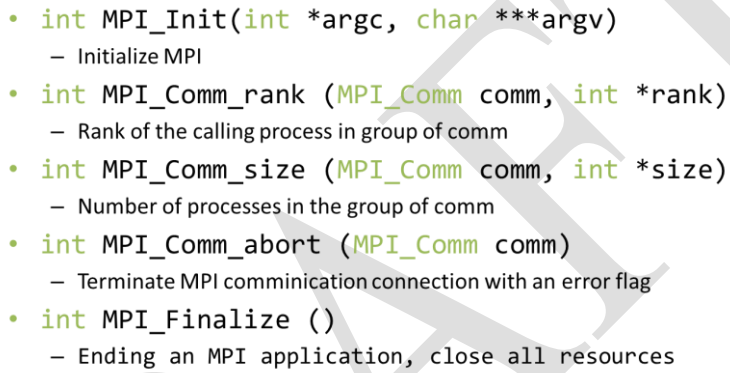


Figure 18.5 Five basic MPI functions for establishing and closing a communication system.

Each process starts by initializing the MPI runtime with a MPI_Init() call. This initializes the communication system for all the processes running the application. Once the MPI runtime is initialized, each process calls two functions to prepare for communication. The first function is MPI_Comm_rank() that returns a unique number to calling each process, which is called the *MPI rank* or process id for the process. The numbers received by the processes vary from 0 to the number of processes minus 1. MPI rank for a process is equivalent to the expression blockIdx.x*blockDim.x+threadIdx.x for a CUDA thread. It uniquely identifies the process in a communication, similar to the phone number in a telephone system.

```
#include "mpi.h"

int main(int argc, char *argv[]) {
    int pad = 0, dimx  = 480+pad, dimy  = 480, dimz  = 400, nreps = 100;
    int pid=-1, np=-1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    MPI_Comm_size(MPI_COMM_WORLD, &np);

    if(np < 3) {
        if(0 == pid) printf("Needed 3 or more processes.\n");
        MPI_Abort( MPI_COMM_WORLD, 1 ); return 1;
    }
    if(pid < np - 1)
        compute_process(dimx, dimy, dimz / (np - 1), nreps);
    else
        data_server( dimx,dimy,dimz );

    MPI_Finalize();
    return 0;
}
```

Figure 18.6 A simple MPI main program.

The MPI_Comm_rank() function takes two parameters. The first one is an MPI built-in type MPI_Comm that specifies the scope of the request. Each variable of the MPI_comm type is commonly referred to as a communicator. MPI_Comm and other MPI built-in types are defined in "mpi.h" header file that should be included in all C program files that use MPI. This is similar to the "cuda.h" header file for CUDA programs. An MPI application can create one or more *communicators* each of which is a group of MPI processes for the purpose of communication. MPI_Comm_rank() assigns a unique id to each process in a communicator. In Figure 18.6, the parameter value passed is MPI_COMM_WORLD, which means that the communicator includes all MPI processes running the application. [1]

The second parameter to the MPI_Comm_rank() function is a pointer to an integer variable into which the function will deposit the returned rank value. In Figure 18.6, a variable pid is declared for this purpose. After the MPI_Comm_rank() returns, the pid variable will contain the unique id for the calling process.

The second API function is MPI_Comm_size(), which returns the total number of MPI processes running in the communicator. The MPI_Comm_size() function takes two parameters. The first one is of MPI_Comm type that gives the scope of the request. In Figure 18.6, the parameter value passed in is MPI_COMM_WORLD, which means the scope of the MPI_Comm_size() is all the processes of the application. Since the scope is all MPI

[1] Interested readers should refer to the MPI reference manual [Gropp1999] for details on creating and using multiple communicators in an application, in particular the definition and use intracommunicators and intercommunicators.

processes, the returned value is the total number of MPI processes running the application. This is value requested by a user when the application is submitted using the mpirun command or the mpiexec command. However, the user may not have requested sufficient number of processes. Also, the system may or may not be able to create all the processes requested. Therefore, it is a good practice for an MPI application program to check the actual number of processes running.

The second parameter is a pointer to an integer variable into which the MPI_Comm_size() function will deposit the return value. In Figure 18.6, a variable np is declared for this purpose. After the function returns, the variable np contains the number of MPI processes running the application. In Figure 18.6, we assume that the application requires at least 3 MPI processes. Therefore, it checks if the number of processes is at least 3. If not, it calls MPI_Comm_abort() function to terminate the communication connections and return with an error flag value 1.

Figure 18.6 also shows a common pattern for reporting errors or other chores. There are multiple MPI processes but we need to report the error only once. The application code designates the process with pid=0 to do the reporting. This is similar to the pattern in CUDA kernels where some tasks only need to be done by one of the threads in a thread-block.

As shown in Figure 18.5, the MPI_Comm_abort() function takes two parameters. The first sets the scope of the request. In Figure 18.6, the scope is set as MPI_COMM_WORLD, which means all MPI processes running the application. The second parameter is a code for the type of error that caused the abort. Any number other than 0 indicates that an error has happened.

If the number of processes satisfies the requirement, the application program goes on to perform the calculation. In Figure 18.6, the application uses np-1 processes (pid from 0 to np-2) to perform the calculation and one process (the last one whose pid is np-1) to perform I/O service for the other processes. We will refer to the process that performs the I/O services as the data server and the processes that perform the calculation as compute processes. In Figure 18.6, if the pid of a process is within the range from 0 to np-2, it is a compute process and call the compute_process() function. If the process pid is np-1, it is the data server and calls data_server() function. This is similar to the pattern where threads perform different actions according to their thread ids.

After the application completes its computation, it notifies the MPI runtime with a call to the MPI_Finalize(), which frees all MPI communication resources allocated to the application. The application can then exit with a return value 0, which indicates that no error occurred.

18.4 MPI Point-to-Point Communication

MPI supports two major types of communication. The first is point-to-point type, which involves one source process and one destination process. The source process calls the MPI_Send() function and the destination process calls the MPI_Recv() function. This is analogous to a caller dialing a call and a receiver answering a call in a telephone system.

- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Buf: starting address of send buffer (pointer)
  - Count: Number of elements in send buffer (nonnegative integer)
  - Datatype: Datatype of each send buffer element (MPI_Datatype)
  - Dest: Rank of destination (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)

Figure 18.7 Syntax for the MPI_Send() function.

Figure 18.7 shows the syntax for using the MPI_Send() function. The first parameter is a pointer to the starting location of the memory area where the data to be sent can be found. The second parameter is an integer that gives that number of data elements to be sent. The third parameter is of an MPI built-in type MPI_Datatype. It specifies the type of each data element being sent. The MPI_Datatype is defined in mpi.h and includes MPI_DOUBLE (double precision floating point), MPI_FLOAT(single precision floating point), MPI_INT (integer), and MPI_CHAR (character). The exact sizes of these types depend on the size of the corresponding C types in the host processor. See the MPI reference manual for more sophisticated use of MPI types [Gropp1999].

The fourth parameter for MPI_Send is an integer that gives the MPI rank of the destination process. The fifth parameter gives a tag that can be used to classify the messages sent by the same process. The sixth parameter is a communicator that selects the processes to be considered in the communication.

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - buf: starting address of receive buffer (pointer)
  - Count: Maximum number of elements in receive buffer (integer)
  - Datatype: Datatype of each receive buffer element (MPI_Datatype)
  - Source: Rank of source (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)
  - Status: Status object (Status)

Figure 18.8 Syntax for the MPI_Recv() function.

Figure 18.8 shows the syntax for using the MPI_Recv() function. The first parameter is a pointer to the area in memory where the received data should be deposited. The second parameter is an integer that gives the maximal number of elements that the MPI_Recv() function is allowed to receive. The third parameter is an MPI_Datatype that specifies the type (size) of each element to be received. The fourth parameter is an integer that gives the process id of the source of the message.

The fifth parameter is an integer that specifies the particular tag value expected by the destination process. If the destination process does not want to be limited to a particular tag value, it can use MPI_ANY_TAG, which means that the receiver is willing to accept messages of any tag value from the source.

We will first use the data server to illustrate the use of point-to-point communication. In a real application, the data server process would typically perform data input and output operations for the compute processes. However, input and output have too much system dependent complexity. Since I/O is not the focus of our discussion, we will avoid the complexity of I/O operations in a cluster environment. That is, instead of reading data from a file system, we will just have the data server to initialize the data with random numbers and distribute the data to the compute processes. The first part of the data server code is shown in Figure 18.9.

```
void data_server(int dimx, int dimy, int dimz, int nreps) {
1.  int np,
/* Set MPI Communication Size */
2.  MPI_Comm_size(MPI_COMM_WORLD, &np);

3.  num_comp_nodes = np – 1, first_node = 0, last_node = np - 2;
4.  unsigned int num_points = dimx * dimy * dimz;
5.  unsigned int num_bytes  = num_points * sizeof(float);
6.  float *input=0, *output=0;
        /* Allocate input data */
7.  input = (float *)malloc(num_bytes);
8.  output = (float *)malloc(num_bytes);
9.  if(input == NULL || output == NULL) {
        printf("server couldn't allocate memory\n");
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    /* Initialize input data */
10. random_data(input, dimx, dimy ,dimz , 1, 10);
    /* Calculate number of shared points */
11. int edge_num_points = dimx * dimy * ((dimz / num_comp_nodes) + 4);
12. int int_num_points  = dimx * dimy * ((dimz / num_comp_nodes) + 8);
13. float *send_address = input;
```

Figure 18.9 Data server process code (Part 1).

The data server function takes four parameters. The first three parameters specify the size of the 3D grid: number of elements in the x dimension dimx, the number of elements in the y dimension dimy, and the number of elements in the z dimension dimz. The fourth parameter specifies the number of iterations that need to be done for all the data points in the grid.

In Figure 18.9, Line 1 declares variable np that will contain the number of processes running the application. Line 2 calls MPI_Comm_size(), which will deposit the information into np. Line 3 declares and initializes several helper variables. The variable num_comp_procs contains the number of compute processes. Since we are reserving one process as data server, there are np-1 compute processes. The variable first_proc gives the process id of the first compute process, which is 0. The variable last_proc gives the process id of the last compute process, which is np-2. That is, Line 3 designates the first np-1 processes, 0 through np-2 as compute processes. This reflects the design decision and the process with the largest rank serves as the data server. This decision will also be reflected in the compute process code.

Line 4 declares and initializes the num_points variable that gives the total number of grid data points to be processed, which is simply the product of the number of elements in each dimesion, or dimx * dimy * dimz. Line 5 declares and initializes the num_bytes variable that gives the total number of bytes needed to store all the grid data points. Since each grid data point is a float, this value is num_points * sizeof(float).

Line 6 declares two pointer variables: input and output. These two pointers will point to the input data buffer and the output data buffer. Line 7 and Line 8 allocate memory for the input

and output buffers and assign their addresses to their respective pointers. Line 9 checks if the memory allocations were successful. If either of the memory allocation fails, the corresponding pointer will receive a NULL pointer from the malloc() function. In this case, the code aborts the application and reports an error.

Line 11 and Line 12 calculate the number of grid point array elements that should be sent to each compute process. As shown in Figure 18.3, there are two types of compute processes. The first process (process 0) and the last process (process 3) compute an "edge" partition that has neighbors only on one side. Partition 0 assigned to the first process has neighbor only on the right side (partition 1). Partition 3 assigned to the last process has neighbor only on the left side (partition 2). We call the compute processes that compute edge partitions the *edge processes*.

Each of the rest of the processes computes an internal partition that has neighbors on both sizes. For example, the second process (process 1) computes a partition (partition 1) that has a left neighbor (partition 0) and a right neighbor (partition 2). We call the processes that compute internal partitions *internal processes*.

Recall that in the Jacobi Iterative Method, each calculation step for a grid point needs the values of its immediate neighbors from the previous step. This creates a need for halo cells for grid points at the left and right boundaries of a partition, shown as slices defined by dotted lines at the edge of each partition in Figure 18.3. Note that these halo cells are similar to those in convolution pattern presented in Chapter 7. Therefore, each process also needs to receive four slices of halo cells that contains all neighbors for each side of the boundary grid points of its partition. For example, in Figure 18.3, partition D2 needs four halo slices from D1 and four halo slices from D3. Note that a halo slice for D2 is a boundary slice for D1 or D3.

Recall that the total number of grid points is dimx*dimy*dimz. Since we are partitioning the grid along the z dimension, the number of grid points in each partition should be dimx*dimy*(dimz / num_comp_procs). Recall that we will need four neighbor slices in each direction in order to calculate values within each slice. Because we need to send four slices of grid points for each neighbor, the number of grid points that should be sent to each internal process should be dimx*dimy*((dimz/num_comp_procs) + 8). As for an edge process, there is only one neighbor. Like in the case of convolution, we assume that zero values will be used for the ghost cells and no input data needs to be sent for them. For example, partition D1 only needs the neighbor slice form D2 on the right side. Therefore, the number of grid points to be sent to an edge process is dimx*dimy*((dimz/num_comp_procs) + 4). That is, each process receives four slices of halo grid points from the neighbor partition on each side.

Line 13 of Figure 18.9 sets the send_address pointer to point to the beginning of the input grid point array. In order to send the appropriate partition to each process, we will need to add the appropriate offset to this beginning address for each MPI_Send(). We will come back to this point later.

We are now ready to complete the code for the data server, shown in Figure 18.10. Line 14 sends Process 0 its partition. Since this is the first partition, its starting address is also the starting address of the entire grid, which was set up in Line 13. Process 0 is an edge process and it does not have a left neighbor. Therefore, the number of grid points to be sent is the value edge_num_points, i.e., dimx*dimy*((dimz/num_comp_procs) + 4). The third parameter specifies that the type of each element is an MPI_FLOAT which is C float (single precision, 4 bytes). The fourth parameter specifies that the value of first_node, i.e., 0, is the MPI rank of the destination process. The fifth parameter specifies 0 for the MPI tag. This is because we are not using tags to distinguish between messages sent from the data server. The sixth parameter specifies that the communicator to be used for sending the message should be all MPI processes for the current application.

```
    /* Send data to the first compute node */
14. MPI_Send(send_address, edge_num_points, MPI_FLOAT, first_node,
             0, MPI_COMM_WORLD );

15. send_address += dimx * dimy * ((dimz / num_comp_nodes) - 4);
    /* Send data to "internal" compute nodes */
16. for(int process = 1; process < last_node; process++) {
17.     MPI_Send(send_address, int_num_points, MPI_FLOAT, process,
                 0, MPI_COMM_WORLD);
18.     send_address += dimx * dimy * (dimz / num_comp_nodes);
    }

    /* Send data to the last compute node */
19. MPI_Send(send_address, edge_num_points, MPI_FLOAT, last_node,
             0, MPI_COMM_WORLD);
```

Figure 18.10 Data server process code (Part 2)

Line 15 of Figure 18.10 advances the send_address pointer to the beginning of the data to be sent to Process 1. From Figure 18.3, there are dimx*dimy*(dimz/num_comp_procs) elements in partition D1, which means D2 starts at location that is dimx*dimy*(dimz/num_comp_procs) elements from the starting location of input. Recall that we also need to send the halo cells from D1 as well. Therefore, we adjust the starting address for the MPI_Send() back by four slices, which results in the expression for advancing the send_address pointer in Line 15: dimx*dimy*((dimz/num_comp_procs) - 4).

Line 16 is a loop that sends out the MPI messages to Process 1 through Process np-3. In our small example for four compute processes, np is 5. The loop sends the MPI messages to Process 1 and Process 2. These are internal processes. They need to receive halo grid points for neighbors on both sides. Therefore, the second parameter of the MPI_Send() in Line 17 uses int_num_nodes, i.e., dimx*dimy*((dimz/num_comp_procs) + 8). The rest of the parameters are similar to that for the MPI_Send() in Line 14 with the obvious exception that

the destination process is specified by the loop variable process, which is incremented from 1 to np-3 (last_node is np-2).

Line 18 advances the send address for each internal process by the number of grid points in each partition: dimx*dimy*dimz/num_comp_nodes. Note that the starting locations of the halo grid points for internal processes are dimx*dimy*dimz/num_comp_procs points apart. Although we need to pull back the starting address by four slices to accommodate halo grid points, we do so for every internal process so the net distance between the starting locations remain as the number of grid points in each partition.

Line 19 sends the data to the Process np-2, the last compute process that has only one neighbor on the left. The reader should be able to reason through all the parameter values used. Note that we are not quite done with the data server code. We will come back later for the final part of the data server that collects the output values from all compute processes.

```
void compute_node_stencil(int dimx, int dimy, int dimz, int nreps ) {
    int np, pid;
1.  MPI_Comm_rank(MPI_COMM_WORLD, &pid);
2.  MPI_Comm_size(MPI_COMM_WORLD, &np);
3.  int server_process = np - 1;


4.  unsigned int num_points       = dimx * dimy * (dimz + 8);
5.  unsigned int num_bytes        = num_points * sizeof(float);
6.  unsigned int num_halo_points = 4 * dimx * dimy;
7.  unsigned int num_halo_bytes  = num_halo_points * sizeof(float);

    /* Alloc host memory */
8.  float *h_input  = (float *)malloc(num_bytes);
       /* Alloca device memory for input and output data */
9.  float *d_input = NULL;
10. cudaMalloc((void **)&d_input,  num_bytes );
11. float *rcv_address = h_input + num_halo_points * (0 == pid);
12. MPI_Recv(rcv_address, num_points, MPI_FLOAT, server_process,
            MPI_ANY_TAG, MPI_COMM_WORLD, &status );
13. cudaMemcpy(d_input, h_input, num_bytes, cudaMemcpyHostToDevice);
```

Figure 18.11 Compute process code (Part 1).

We now turn our attention to the compute processes that receive the input from the data server process. In Figure 18.11, Line 1 and Line 2 establish the process id for the process and the total number of processes for the application. Line 3 establishes that the data server is Process np-1. Line 4 and Line 5 calculates the number of grid points and the number of bytes that should be processed by each internal process. Line 6 and Line 7 calculate the number of grid points and the number of bytes in each halo (4 slices).

Line 8, Line 9, and Line 10 allocate the host memory and device memory for the input data. Although the edge processes need less halo data, they still allocate the same amount of memory for simplicity; part of the allocated memory will not be used by the edge processes.

Line 11 sets the starting address of the host memory for receiving the input data from the data server. For all compute processes except Process 0, the starting receiving location is simply the starting location of the allocated memory for the input data. However, we adjust the receiving location by 4 slices. This is because for simplicity, we assume that the host memory for receiving the input data is arranged the same way for all compute processes: 4 slices of halo from the left neighbor followed by the partition, followed by 4 slices of halo from the right neighbor. However, we showed in Line 4 of Figure 18.10, the data server will not send any halo data from the left neighbor to Process 0. That is, for Process 0, the MPI message from the data server only contains the partition and the halo from the right neighbor. Therefore, Line 10 adjusts the starting host memory location by 4 slices so that process 0 will correctly interpret the input data from the data server.

Line 12 receives the MPI message from the data server. Most of the parameters should be familiar. The last parameter reflects any error condition that occurred when the data is received. The second parameter specifies that all compute processes will receive the full amount of data from the data server. However, the data server will send less data to Process 0 and Process np-2. This is not reflected in the code because MPI_Recv() allows the second parameter to specify a larger number of data points than what is actually received and will only place the actual number of bytes received from the sender into the receiving memory. In the case of Process 0, the input data from the data server contain only the partition and the halo from the right neighbor. The received input will be placed by skipping the first 4 slices of the allocated memory, which should correspond to the halo for the (non-existent) left neighbor. This effect is achieved with the term num_halo_points*(pid==0) in Line 11. In the case of Process np-2, the input data contain the halo from the left neighbor and the partition. The received input will be placed from the beginning of the allocated memory, leaving the last four slices of the allocated memory unused.

Line 13 copies the received input data to the device memory. In the case of Process 0, the left halo points are not valid. In the case of Process np-2, the right halo points are not valid. However, for simplicity, all compute nodes send the full size to the device memory. The assumption is that the kernels will be launched in such a way that these invalid portions will be correctly ignored. After Line 13, all the input data are in the device memory.

```
14. float *h_output = NULL, *d_output = NULL, *d_vsq = NULL;
15. float *h_output = (float *)malloc(num_bytes);
16. cudaMalloc((void **)&d_output, num_bytes );

17. float *h_left_boundary = NULL, *h_right_boundary = NULL;
18. float *h_left_halo = NULL, *h_right_halo = NULL;

    /* Alloc host memory for halo data */
19. cudaHostAlloc((void **)&h_left_boundary, num_halo_bytes, cudaHostAllocDefault);
20. cudaHostAlloc((void **)&h_right_boundary,num_halo_bytes, cudaHostAllocDefault);
21. cudaHostAlloc((void **)&h_left_halo,     num_halo_bytes, cudaHostAllocDefault);
22. cudaHostAlloc((void **)&h_right_halo,    num_halo_bytes, cudaHostAllocDefault);

    /* Create streams used for stencil computation */
23. cudaStream_t stream0, stream1;
24. cudaStreamCreate(&stream0);
25. cudaStreamCreate(&stream1);
```

Figure 18.12 Compute process code (Part 2).

Figure 18.12 shows Part 2 of the compute process code. Line 14, Line 15, and Line 16 allocate host memory and device memory for the output data. The output data buffer in the device memory will actually be used as a ping-pong buffer with the input data buffer. That is, they will switch roles in each iteration. Recall that we used a similar scheme in the BFS pattern in Chapter 12. We will return to this point later.

We are now ready to present the code that performs computation steps on the grid points.

18.5 Overlapping Computation and Communication

A simple way to perform the computation steps is for each compute process to perform a computation step on its entire partition, exchange halo data with the left and right neighbors, and repeat. While this is a very simple strategy, it is not very effective. The reason is that this strategy forces the system to be in one of the two modes. In the first mode, all compute processes are performing computation steps. During this time, the communication network is not used. In the second mode, all compute processes are exchange halo data with their left and right neighbors. During this time, the computation hardware is not well utilized. Ideally, we would like to achieve better performance by utilizing both the communication network and computation hardware all the time. This can be achieved by dividing the computation tasks of each compute process into two stages, as illustrated in Figure 18.13.

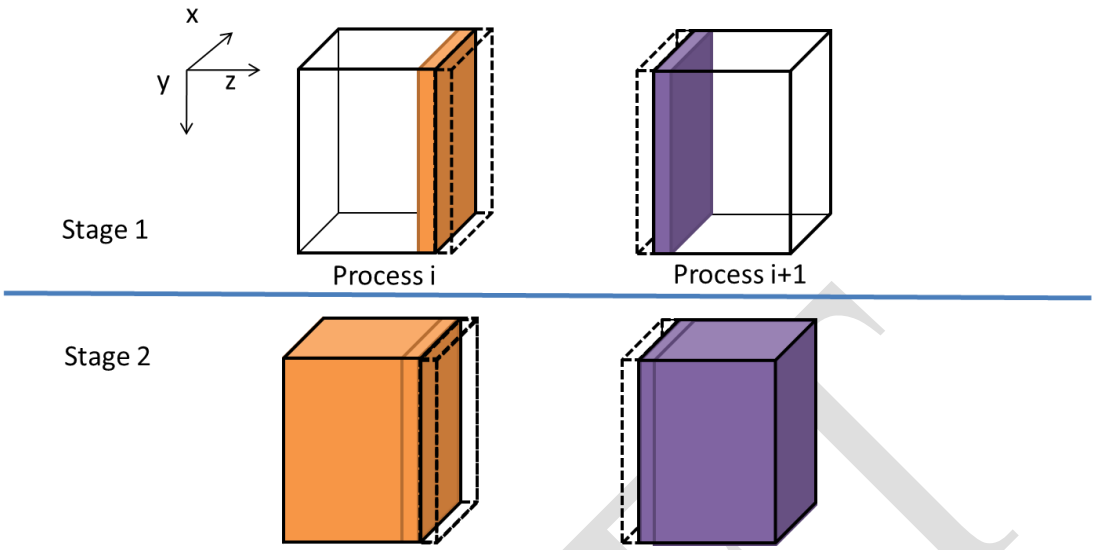


Figure 18.13 A two-stage Strategy for overlapping computation with communication.

During the first stage (Stage 1), each compute process calculates its boundary slices that will be needed as halo cells by its neighbors in the next iteration. Let's continue to assume that we use 4 slices of halo data. Figure 18.13 shows that the collection of four halo slices as a dashed transparent piece and the four boundary slices as a colored piece. Note that the colored piece of Process i will be copied into the dashed piece of Process i+1 and vice versa during the next communication. For Process 0, the first phase calculates the right 4 slices of boundary data. For an internal node, it calculates the left 4 slices and the right 4 slices of its boundary data. For Process n-2, it calculates the left 4 pieces of its boundary data. The rationale is that these boundary slices are needed by their neighbors for the next iteration. By calculating these boundary slices first, the data can be communicated to the neighbors while the compute processes calculate the rest of its grid points.

During the second stage (Stage 2), each compute process performs two parallel activities. The first is to communicate its new boundary values to its neighbor processes. This is done by first copying the data from the device memory into the host memory, followed by sending MPI messages to the neighbors. As we will discuss later, we need to be careful that the data received from the neighbors are used in the next iteration, not the current iteration. The second activity is to calculate the rest of the data in the partition. If the communication activity takes shorter amount of time than the calculation activity, we can hide the communication delay and fully utilize the computing hardware all the time. This is usually achieved by having enough slices in the internal part of each partition allow each compute process to perform computation steps in between communications.

In order to support the parallel activities in Stage 2, we need to use two advanced features of the CUDA programming model: *pinned memory allocation* and *streams*. A pinned memory allocation requests that the memory allocated will not be paged out by the operating system. This is done with the cudaHostAlloc() API call. Line 19 through Line 22 allocates

memory buffers for the left and right boundary slices and the left and right halo slices. The left and right boundary slices need to be sent from the device memory to the left and right neighbor processes. The buffers are used as a host memory staging area for the device to copy data into and then used as the source buffer for MPI_Send() to neighbor processes. The left and right halo slices need to be received from neighbor processes. The buffers are used as a host memory staging area for MPI_Recv() to use as destination buffer and then copied to the device memory.

Note that the host memory allocation is done with cudaHostAlloc() function rather than the standard malloc() function. The difference is that the cudaHostAlloc() function allocates a *pinned memory* buffer, sometimes also referred to as *page locked memory* buffer. We need to know a little more background on the memory management in operating systems in order to fully understand the concept of pinned memory buffers.

In a modern computer system, the operating system manages a virtual memory space for applications. Each application has access to a large, consecutive address space. In reality, the system has a limited amount of physical memory that needs to be shared among all running applications. This sharing is performed by partitioning the virtual memory space into pages and mapping only the actively used pages into physical memory. When there is much demand for memory, the operating system needs to "page out" some of the pages from the physical memory to mass storage such as disks. Therefore, an application may have its data paged out any time during its execution.

The implementation of cudaMemcpy() uses a type of hardware called Direct Memory Access (DMA) device. When a cudaMemcpy() function is called to copy between the host and device memories, its implementation uses a DMA to complete the task. On the host memory side, the DMA hardware operates on physical addresses. That is, the operating system needs to give a translated physical address to DMA. However, there is a chance that the data may be paged out before the DMA operation is complete. The physical memory locations for the data may be re-assigned to another virtual memory data. In this case, the DMA operation can be potentially corrupted since its data can be overwritten by the paging activity.

A common solution to this data corruption problem is for the CUDA runtime to perform the copy operation in two steps. For a host-to-device copy, the CUDA runtime first copies the source host memory data into a "pinned" memory buffer, which means the memory locations are marked so that the operating paging mechanism will not page out the data. It then uses the DMA device to copy the data from the pinned memory buffer to the device memory. For a device-to-host copy, the CUDA runtime first uses a DMA device to copy the data from the device memory into a pinned memory buffer. It then copies the data from the pinned memory to the destination host memory location. By using an extra pinned memory buffer, the DMA copy will be safe from any paging activities.

There are two problems with this approach. One is that the extra copy adds delay to the cudaMemcpy() operation. The second is that the extra complexity involved leads to a synchronous implementation of the cudaMemcpy() function. That is, the host program cannot continue to execute until the cudaMemcpy() function completes its operation and returns. This serializes all copy operations. In order to support fast copies with more parallelism, CUDA provides a cudaMemcpyAsync() function.

In order to use cudaMemcpyAsync() function, the host memory buffer must be allocated as a pinned memory buffer. This is done in Lines 19 through 22 for the host memory buffers of the left boundary, right boundary, left halo, and right halo slices. These buffers are allocated with the cudaHostAlloc() function, which ensures that the allocated memory are pinned or page locked from paging activities. Note that the cudaHostAlloc() function takes three parameters. The first two are the same as cudaMalloc(). The third specifies some options for more advanced usage. For most basic use cases, we can simply use the default value cudaHostAllocDefault.

The second advanced CUDA feature is *streams*, which supports managed concurrent execution of CUDA API functions. A stream is an ordered sequence of operations. When a host code calls a cudaMemcpyAsync() function or launches a kernel, it can specify a steam as one of its parameters. All operations in the same stream will be done sequentially. Operations from two different streams can be executed in parallel.

Line 23 of Figure 18.13 declares two variables that are of CUDA built-in type cudaStream_t. Recall that the CUDA built-in types are declared in cuda.h. These variables are then used in calling the cudaStreamCreate() function. Each call to the cudaStreamCreate() creates a new stream and deposits a pointer to the stream into its parameter. After the calls in Lines 24 and 25, the host code can use either steram0 or stream1 in subsequent cudaMemcpyAsync() calls and kernel launches.

Figure 18.14 shows Part 3 of the compute process. Line 27 and Line 28 calculate the process id of the left and right neighbors of the compute process. The left_neighbor and right_neighbor variables will be used by compute processes as parameters when they send message to and receive messages from their neighbors. For Process 0, there is no left neighbor, so Line 27 assigns an MPI constant MPI_PROC_NULL to left_neighbor to note this fact. For Process np-2, there is no right neighbor, so Line 28 assigns MPI_PROC_NULL to right_neighbor. For all the internal processes, Line 27 assigns pid-1 to left_neighbor and pid+1 to right_neighbor.

```
26. MPI_Status status;
27. int left_neighbor  = (pid > 0)    ? (pid - 1) : MPI_PROC_NULL;
28. int right_neighbor = (pid < np - 2) ? (pid + 1) : MPI_PROC_NULL;

    /* Upload stencil cofficients */
    upload_coefficients(coeff, 5);

29. int left_halo_offset   = 0;
30. int right_halo_offset  = dimx * dimy * (4 + dimz);
31. int left_stage1_offset  = 0;
32. int right_stage1_offset = dimx * dimy * (dimz - 4);
33. int stage2_offset       = num_halo_points;

34. MPI_Barrier( MPI_COMM_WORLD );
35. for(int i=0; I < nreps; i++) {
        /* Compute boundary values needed by other nodes first */
36.     launch_kernel(d_output + left_stage1_offset,
            d_input + left_stage1_offset, dimx, dimy, 12, stream0);
37.     launch_kernel(d_output + right_stage1_offset,
            d_input + right_stage1_offset, dimx, dimy, 12, stream0);

        /* Compute the remaining points */
38.     launch_kernel(d_output + stage2_offset, d_input + stage2_offset,
                    dimx, dimy, dimz, stream1);
```

Figure 18.14 Compute process code (Part 3).

Lines 31 through 33 set up several offsets that will be used to launch kernels and exchange data so that the computation and communication can be overlapped. These offsets define the regions of grid points that will need to be calculated at each stage of Figure 18.13. They are also visualized in Figure 18.15.

Note that the total number of slices in each device memory is 4 slices of left halo points (dashed white), plus 4 slices of left boundary points, plus dimx*dimy*(dimz-8) internal points, plus 4 slices of boundary points, and 4 slices of right halo points (dashed white). Variable left_stage1_offset defines the starting point of the slices that are needed in order to calculate the left boundary slices. This includes 12 slices of data: 4 slices of left-neighbor halo points, 4 slices of boundary points, and 4 slices of internal points. These slices are the leftmost in the partition so the offset value is set to 0 by Line 31. Variable right_stage2_offset defines the starting point of the slices that are needed for calculating the right boundary slices. This also includes 12 slices: 4 slices of internal points, 4 slices of right boundary points, and 4 slices of right halo cells. The beginning point of these 12 slices can be derived by subtracting the total number of slices dimz+8 by 12. Therefore, the starting offset for these 12 slices is dimx*dimy*(dimz-4).

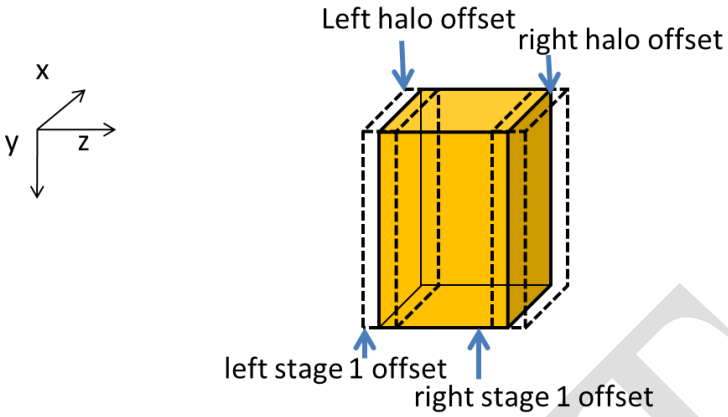


Figure 18.15 Device memory offsets used for data exchange with neighbor processes.

Line 35 is an MPI barrier synchronization, which is similar to the CUDA __syncthreads(). MPI barrier forces all MPI processes specified by the parameter to wait for each other. None of the processes can continue their execution beyond this point until everyone has reached this point. The reason why we want to barrier synchronization here is to ensure that all compute nodes have received their input data and are ready to perform the computation steps. Since they will be exchanging data with each other, we would like to make them all start at about the same time. This way, we will not be in a situation where a few tardy processes delay all other processes during the data exchange. MPI_Barrier() is a *collective communication* function. We will discuss more details about collective communication API functions in the next section.

Line 35 starts a loop that performs the computation steps. For each iteration, each compute process will perform one cycle of the 2-stage process in Figure 18.13.

Line 36 calls a function that will generate the 4 slices of the left boundary points in Stage 1. We assume that there is a kernel that performs one computation step on a region of grip points. The launch_kernel() function takes several parameters. The first parameter is a pointer to the output data area for the kernel. The second parameter is a pointer to the input data area. In both cases, we add the left_stage1_offset to the input and output data in the device memory. The next three parameters specify the dimensions of the portion of the grid to be processed, which is 12 slices in this case. Note that we need to have four slices on each side in order to correctly perform 4 computation steps for all the points in the four left boundary slices. Line 37 does the same for the right boundary points in Stage 1. Note that these kernels will be launched within stream0 and will be executed sequentially.

Line 38 launches a kernel to generate the dimx*dimy*(dimz-8) internal points in Stage 2. Note that this also requires 4 slices of input boundary values on each side so the total number of input slices is dimx*dimy*dimz. The kernel is launched in stream1 and will be executed in parallel with those launched by Line 36 and Line 37.


```
        /* Copy the data needed by other nodes to the host */
39.     cudaMemcpyAsync(h_left_boundary, d_output + num_halo_points,
                num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
40.     cudaMemcpyAsync(h_right_boundary,
                    d_output + right_stage1_offset + num_halo_points,
                    num_halo_bytes, cudaMemcpyDeviceToHost, stream0 );
41.     cudaStreamSynchronize(stream0);
```

Figure 18.16 Compute process code (Part 4)

Figure 18.16 shows Part 4 of the compute process code. Line39 copies the 4 slices of left boundary points to the host memory in preparation for data exchange with the left neighbor process. Line 40 copies the 4 slices of the right boundary points to the host memory in preparation for data exchange with the right neighbor process. Both are asynchronous copies in Stream 0 and will wait for the two kernels in Stream 0 to complete before they copy data. Line 40 is a synchronization that forces the process to wait for all operations in Stream 0 to complete before it can continue. This makes sure that the left and right boundary points are in the host memory before the process proceeds with data exchange.

- `int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`
  - Sendbuf: Initial address of send buffer (choice)
  - Sendcount: Number of elements in send buffer (integer)
  - Sendtype: Type of elements in send buffer (handle)
  - Dest: Rank of destination (integer)
  - Sendtag: Send tag (integer)
  - Recvcount: Number of elements in receive buffer (integer)
  - Recvtype: Type of elements in receive buffer (handle)
  - Source: Rank of source (integer)
  - Recvtag: Receive tag (integer)
  - Comm: Communicator (handle)
  - Recvbuf: Initial address of receive buffer (choice)
  - Status: Status object (Status). This refers to the receive operation.

Figure 18.17 Syntax for the MPI_Sendrecv() function.

During the data exchange phase, we will have all MPI processes to send their boundary points to their left neighbors. That is, all processes will have their right neighbors sending data to them. It is therefore convenient to have an MPI function that sends data to a destination and receives data from a source. This reduces the number of MPI function calls. MPI_Sendrecv() function in Figure 18.17 is such a function. It is essentially a combination of MPI_Send() and MPI_Recv() so we will not further elaborate on the meaning of the parameters.

```
        /* Send data to left, get data from right */
42.     MPI_Sendrecv(h_left_boundary, num_halo_points, MPI_FLOAT,
                     left_neighbor,  i, h_right_halo,
                     num_halo_points, MPI_FLOAT, right_neighbor, i,
                     MPI_COMM_WORLD, &status );
        /* Send data to right, get data from left */
43.     MPI_Sendrecv(h_right_boundary, num_halo_points, MPI_FLOAT,
                     right_neighbor, i, h_left_halo,
                     num_halo_points, MPI_FLOAT, left_neighbor,  i,
                     MPI_COMM_WORLD, &status );

44.     cudaMemcpyAsync(d_output+left_halo_offset,  h_left_halo,
                     num_halo_bytes, cudaMemcpyHostToDevice, stream0);
45.     cudaMemcpyAsync(d_output+right_halo_offset, h_right_halo,
                     num_halo_bytes, cudaMemcpyHostToDevice, stream0 );
46.     cudaDeviceSynchronize();

47.     float *temp = d_output;
48.     d_output = d_input; d_input = temp;
    }
```

Figure 18.18 Compute process code (Part 5).

Figure 18.18 shows Part 5 of the compute process code. Line 42 sends 4 slices of left boundary points to the left neighbor and receives 4 slices of right halo points from the right neighbors. Line 43 sends 4 slices of right boundary points to the right neighbor and receives 4 slices of left halo points from the left neighbor. In the case of Process 0, its left_neighbor has been set to MPI_PROC_NULL in Line 27 so the MPI runtime will not send out the message in Line 42 or receive the message in Line 43 for Process 0. Likewise, the MPI runtime will not receive the message in Line 42 or send out the message in Line 43 for Process np-2. Therefore, the conditional assignments in Lines 27 and 28 eliminate the need for special if-the-else statements in Lines 42 and 43.

After the MPI messages have been sent and received, Lines 44 and 45 transfer the newly received halo points to the d_output buffer of device memory. These copies are done in stream0 so they will execute in parallel with the kernel launched in Line 38.

Line 46 is a synchronize operation for all device activities. This call forces the process to wait for all device activities, including kernels and data copies to complete. When the cudaDeviceSynchronize() function returns, all d_output data from the current computation step are in place: left halo data from the left neighbor process, boundary data from the kernel launched in Line 36, internal data form the kernel launched in Line 38, right boundary data from the kernel launched in Line 37, and right halo data from the right neighbor.

Lines 47 and 48 swap the d_input and d_output pointers. This changes the output of the d_ouput data of the current computation step into the d_input data of the next computation step. The execution then proceeds to the next computation step by going to the next iteration of the loop of Line 35. This will continue until all compute processes complete the number of computations specified by the parameter nreps.

```
    /* Wait for previous communications */
49. MPI_Barrier(MPI_COMM_WORLD);

50. float *temp = d_output;
51. d_output = d_input;
52. d_input = temp;

    /* Send the output, skipping halo points */
53. cudaMemcpy(h_output, d_output, num_bytes, cudaMemcpyDeviceToHost);
    float *send_address = h_output + num_ghost_points;
54. MPI_Send(send_address, dimx * dimy * dimz, MPI_REAL,
             server_process, DATA_COLLECT, MPI_COMM_WORLD);
55. MPI_Barrier(MPI_COMM_WORLD);

    /* Release resources */
56. free(h_input); free(h_output);
57. cudaFreeHost(h_left_ghost_own); cudaFreeHost(h_right_ghost_own);
58. cudaFreeHost(h_left_ghost); cudaFreeHost(h_right_ghost);
59. cudaFree( d_input ); cudaFree( d_output );
}
```

Figure 18.19 Compute process code (Part 6)

Figure 18.9 shows Part 6, the final part of the compute process code. Line 49 is a barrier synchronization that forces all processes to wait for each other to finish their computation steps. Lines 50 through 52 swap d_output with d_input. This is because Lines 47 and 48 swapped d_output with d_input in preparation for the next computation step. However, this is unnecessary for the last computation step. So, we use Lines 50 through 52 to undo the swap. Line 53 copies the final output to the host memory. Line 54 sends the output to the data server. Line 55 waits for all processes to complete. Lines 56 through 59 free all the resources before returning to the main program.

Figure 18.20 shows Part 3, the final part of the data server code, which continues from Figure 18.10. Line 20 is a barrier synchronization that waits for all compute nodes to complete their computation steps and send their outputs. This barrier corresponds to the barrier at Line 55 of the compute process. Line 22 receives the output data from all the compute processes. Line 23 stores the output into an external storage. Lines 24 and 25 free resources before returning to the main program.

```
    /* Wait for nodes to compute */
20. MPI_Barrier(MPI_COMM_WORLD);

    /* Collect output data */
21. MPI_Status status;
22. for(int process = 0; process < num_comp_nodes; process++)
        MPI_Recv(output + process * num_points / num_comp_nodes,
            num_points / num_comp_nodes, MPI_REAL, process,
            DATA_COLLECT, MPI_COMM_WORLD, &status );

    /* Store output data */
23. store_output(output, dimx, dimy, dimz);

    /* Release resources */
24. free(input);
25. free(output);
}
```

Figure 18.20 Data server code (Part 3)

18.6 MPI Collective Communication

The second type of MPI communication is collective communication, which involves a group of MPI processes. We have already seen an example of the second type of MPI communication API in the previous section: MPI_Barrier. The other commonly used group collective communication types are broadcast, reduction, gather, and scatter [Gropp1999].

Barrier synchronization MPI_Barrier() is perhaps the most commonly used collective communication function. As we have seen the stencil example, barriers are used to ensure that all MPI processes are ready before they begin to interact with each other. We will not elaborate on the other types of MPI collective communication functions but encourage the reader to read up on the details of these functions. . In general, collective communication functions are highly optimized by the MPI runtime developers and system vendors. Using them usually leads to better performance as well as readability and productivity than trying to achieve the same functionality with combinations of send and receive calls.

18.7 CUDA Aware MPI

Modern MPI implementations are aware of the CUDA programming model and are designed to minimize the communication latency between GPUs. Currently, direct interaction between CUDA and MPI is supported by MVAPICH2, IBM Platform MPI, and OpenMPI.

CUDA-aware MPI implementations are capable of sending messages from the GPU memory in one node to the GPU memory in a different node. This effectively removes the need of device-to-host data transfers before sending MPI messages, and host-to-device data transfers after receiving an MPI message. This has the potential of simplifying the host code and

memory data layout. Following with our stencil example, if we use a CUDA-aware MPI implementation we no longer need host-pinned memory allocations and asynchronous memory copies.

The first simplification is that we no longer need host-pinned memory buffers to transfer the halo points to the host memory. This means that we can safely remove lines 19-22 in Figure 18.12. However, we still need to use CUDA streams and two separate GPU kernels to start communicating across nodes as soon as the halo elements have been computed.

The second simplification is that we no longer need to asynchronously copy the halo data from the device to the host memory. As a result, we can also remove lines 39 and 40 in Figure 18.19. Since the MPI calls now accepts device memory addresses, we need to modify the calls to MPI_SendRecv to use them. Note that these memory addresses actually correspond to the device addresses of the asynchronous memory copies in the previous versions.

```
MPI_SendRecv(d_output + num_halo_points, num_halo_points, MPI_FLOAT,
        left_neighbor, i, d_output + left_halo_offset, num_halo_points,
        MPI_FLOAT, right_neighbor, i, MPI_COMM_WORLD, &status);
MPI_SendRecv(d_output + right_stage1_offset, num_halo_points,
        num_halo_points, MPI_FLOAT, right_neighbor, i,
        d_output + right_halo_offset, num_halo_points,
        MPI_FLOAT, left_neighbor, i, MPI_COMM_WORLD, &status);
```

Figure 18.21 Revised MPI SendRec calls when using CUDA-aware MPI

Since the CUDA-aware MPI implementations will directly update the contents of the GPU memory, we also remove lines 44and 45 in Figure 18.19.

Besides removing the data transfers during the halo exchange using MPI_SendRecv(), it would be also possible to remove the initial and final memory copies be receiving/sending the input/output directly from the GPU memory.

18.8. Summary

We have covered basic patterns of joint CUDA/MPI programming for HPC clusters with heterogeneous computing nodes. All processes in an MPI application run the same program. However, each process can follow different control flow and function call paths to specialize their roles, as illustrtaed by the data server and the compute processes in our example. We have also presented a common pattern where compute processes exchange data. We presented the use of CUDA streams and asynchronous data transfers to enable the overlap of computation and communication. We would like to point out that while MPI is a very different programming system, all major MPI concepts that we covered in this chapter, SPMD, MPI ranks, and barriers have counter parts in the CUDA programming model. This

confirms our belief that by teaching parallel programming with one model well, our students can quickly pick up other programming models easily. We would like to encourage the reader to build on the foundation from this chapter and study more advanced MPI features and other important patterns.

References

[Gropp1999] Gropp, William; Lusk, Ewing; Skjellum, Anthony (1999a). *Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series. ISBN 978-0-262-57132-6.

18.9 Exercises

1. For vector addition, if there are 100,000 elements in each vector and we are using 3 compute processes. How many elements are we sending to the last compute process?
   (A) 5
   (B) 300
   (C) 333
   (D) 334

2. If the MPI call MPI_Send(ptr_a, 1000, MPI_FLOAT, 2000, 4, MPI_COMM_WORLD) resulted in a data transfer of 40000 bytes, what is the size of each data element being sent?
   (A) 1 byte
   (B) 2 bytes
   (C) 4 bytes
   (D) 8 bytes

3. Which of the following statements is true?
   (A) MPI_Send() is blocking by default.
   (B) MPI_Recv() is blocking by default.
   (C) MPI messages must be at least 128 bytes.
   (D) MPI processes can access the same variable through shared memory.

4. Using the code base in Appendix A and examples in Chapters 3, 4, 5, and 6 to develop an OpenCL version of the matrix-matrix multiplication application.

5. Modify the example code to remove the calls to cudaMemcpy() on the compute node code by using GPU memory addresses on MPI_Send and MPI_Recv.