Chapter 1
Introduction

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 to the desktop and hundreds of GFLOPS to cluster servers. This relentless drive of performance improvement has allowed application software to perform more functionality, have better user interfaces, and generate more useful results with more. The users, in turn, demand even more improvements once they become accustomed to these improvements, creating a positive cycle for the computer industry.

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

Traditionally, the vast majority of software applications are written as sequential programs, as described by von Neumann in his seminal paper in 1947(?). For these sequential programs, their execution can be understood by sequentially stepping through the code. Historically, computer users have become accustomed to the expectation that these programs run faster with each new generation of microprocessors. Such expectation is no longer valid from this day onward. A sequential program will only run on one of the processor cores, which will not become any faster than those in use today. Without performance improvement, application developers will no longer be able to introduce new features and capabilities into their software as new microprocessors are introduced, reducing the growth opportunities of the entire computer industry.

Rather, the applications software that will continue to enjoy performance improvement with each new generation of microprocessors will be parallel programs, in which multiple threads of execution cooperate to achieve the functionality faster. This new, dramatically
escalated incentive for parallel program development has been referred to as the parallelism revolution [Larus ACM Queue article]. The practice of parallel programming is by no means new. The high-performance computing community has been developing parallel programs for decades. These programs run on large scale, expensive computers. Only a few elite applications can justify the use of these expensive computers, thus limiting the practice of parallel programming to a small number of application developers. Now that all new microprocessors are parallel computers, the number of applications that need to be developed as parallel programs has increased dramatically. There is now a great need for software developers to learn about parallel programming, which is the focus of this book.

1.1. GPUs as Parallel Computers

Since 2003, a class of many-core processors called Graphics Processing Units (GPUs), have led the race for floating-point performance. This phenomenon is illustrated in Figure 1.1. While the performance improvement of general-purpose microprocessors has slowed significantly, the GPUs have continued to improve relentlessly. As of 2008, the ratio of peak floating-point calculation throughput between many-core GPUs and multi-core CPUs is about 10. These are not necessarily achievable speeds, merely the raw speed that the execution resources can potentially support in these chips: 367 gigaflips vs. 32 gigaflops. NVIDIA has subsequently delivered software driver and clock improvements that allow a G80 Ultra to reach 518 gigaflops, only about seven months after the original chip. In June 2008, NVIDIA introduced the GT200 chip, which delivers almost 1 teraflop (1,000 gigaflops) of single precision and almost 100 gigaflops of double precision performance.

![Figure 1.1. Enlarging Performance Gap between GPUs and CPUs.](http://www.gpgpu.org/s2007/slides/15-GPGPU-physics.pdf)
Such a large performance gap between parallel and sequential processors has amounted to a significant “electrical potential build-up,” and at some point, something will have to give. We may be nearing that point now. To date, this large performance gap has already motivated many applications developers to move the computationally intensive parts of their software to 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 up the work amount cooperating threads of execution.

One might ask why there is such a large performance gap between many-core 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 1.2. The design of a CPU is optimized for sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data access latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed. As of 2008, the new general-purpose multi-core microprocessors typically have four large processor cores designed to deliver strong sequential code performance.

Memory bandwidth is another important issue. Graphics chips have been operating at approximately 10x the bandwidth of contemporaneously available CPU chips. In late 2006, G80 was capable of about 80 gigabytes per second (GB/S) into the main DRAM. Because of frame buffer requirements and the relaxed memory model (and without relying too heavily on architecture details of other coherence models and memory models), general-purpose processors have to satisfy requirements from legacy operating systems and applications that make memory bandwidth more difficult to increase. And with simpler memory models, the GPUs must be able to get about 80 GB/S into the memory. The most recent chip supports more than 100 GB/S. Microprocessor system memory bandwidths will probably not grow beyong 20 GB/S for about three years, so CPUs will continue to be at a disadvantage in terms of memory bandwidth for some time.

The design philosophy of the GPUs is forced 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 the GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The general philosophy for GPU design is to optimize for the execution of massive number of threads. The hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, minimizing the control logic required for each execution thread. 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. As a result, much more chip area is dedicated to the floating-point calculations.
It should be clear now that GPU is designed as a numeric computing engine and it will not perform well on some tasks that CPUs are designed to perform well. Therefore, one should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numeric intensive parts on the GPUs. This is why the CUDA programming model is designed to support joint CPU-GPU execution of an application.

It is also important to note that performance is not the only decision factor when application developers choose the processors for running their applications. Several other factors can be even more important. First and foremost, the processors of choice must have a very large presence in the market place, referred to as the installation base of the processor. The reason is very simple. The cost of software development is best justified by a very large customer population. Applications that can be 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 processing 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 processing systems. This has changed with many-core GPUs. Due to their popularity in the PC market, GPUs have been sold by the hundreds of millions. Virtually all PCs have GPUs in them. The G80 family of CUDA-capable processors and its successors have shipped almost 100 million units to date. This is the first time that massively parallel computing is part of a mass-market product. Such a large market presence has made these GPUs economically attractive for application developers.

Another important decision factor is practical form factors and easy accessibility. Until 2006, with the advent of parallel programming and newly developed parallel software, production work usually ran on servers or datacenters on departmental clusters. But the use of these applications tends to be limited. 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 are all based on some combination of a PC and special hardware accelerators. The simple reason is that manufacturers such as GE and Siemens cannot sell MRIs with racks and racks of clusters 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 in the foreseeable future.

Another important consideration in selecting a processor for executing numeric computing applications is the support for IEEE Floating-Point Standard. The standard makes it possible to have predictable results across processors from different vendors. While the support for IEEE Floating-Point Standard was not strong in early GPUs, this has also changed for the new generations of GPUs such as the GeForce 8 series. As we will discuss in Chapter 5, GPU support for 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 values as the CPUs. A major remaining issue is that the GPUs floating-point arithmetic units are primarily single precision today. Applications that truly require double precision floating-point will not be suitable for GPU execution in the immediate future. Nevertheless, we have already seen many applications where single-precision floating is sufficient.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphic API to access the processor cores, meaning that open GL or direct 3D techniques were needed to program these chips. This technique was called GPGPU, for General Purpose Programming using a Graphics Processing Unit. Even with a higher level programming environment, the underlying code is still limited by the APIs. These APIs limit the kinds of applications that one can actually write for these chips. That’s why only a few people could master the skills necessary to use these chips to achieve performance. Consequently, it did not become a widespread programming phenomenon. Nonetheless, this technology was sufficiently exciting to inspire some heroic efforts and excellent results.

But everything changed in 2007 with the release of CUDA. NVIDIA actually devoted silicon area to facilitate the ease of parallel programming, so this does not represent software changes alone; additional hardware was added to the chip. Therefore, if you use the G80 and follow-up chips for GPU Computing, the programming interface will not go through the graphics interface at all. Instead, a new general-purpose interface on the silicon will enable this. Moreover, all the other software layers were redone as well, so that you can do essential reprogramming. This makes a huge difference. Some of our students tried to do their machine problems (MPs) using the old programming interface and MPIs after finishing the first few MPs, and they tremendously appreciated the difference.

1.2. Architecture of a modern GPU

Figure 1.3 shows the architecture of a typical GPU today? It is organized into 16 highly threaded Streaming Multiprocessors (SMs). A pair of SMs form a building block in Figure 1.3. Each SM has 8 streaming processors (SPs), for a total of 128 (16*8). Each SP has a multiply-add (MAD) unit, and and an additional multiply (MUL) unit, all running at 1.35
gigahertz (GHz). If you do the math, that’s almost 367 gigaflops for the MADs and a total of over 500 gigaflops if you include the MULs as well. In addition, special function units perform floating point functions such as SQRT and RCP SQRT as well as transcendental functions. Each GPU currently comes with 1.5 megabytes of DRAM. These DRAMs differ from the system memory DIMM DRAMs on the motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold high-definition video images, and texture information for 3D rendering as in games. But for computing, they function like very high bandwidth off-chip cache, though with somewhat more latency regular cache or system memory. If the chip is programmed properly, the high bandwidth makes up for the large latency.

Figure 1.3. Architecture of a CUDA-capable GPU

The processor has 86.4 GB/s of memory bandwidth, plus 4 gigabytes of bandwidth each way across the PCI-express bus – a total of 8 GB/s for communication with the CPU. You can transfer data from the system memory at 4 GB/s, and you can upload data back to the system memory at 4 GB/s. Altogether, there is a combined total of 8 gigabytes/second, but for practical purposes, you’ll likely work one way or the other at a time unless you are overlapping data transfers before and after a sequence of computations. This may seem like a limitation, but the PCI-E bandwidth is comparable to the system memory and CPU front-side bus bandwidth, so it’s really not the limitation it would seem at first.

Some important characteristics: peak performance is about 10 times better than the current highest end microprocessors. One of my students, John Stone, forwarded an ion placement application, a forced calculation application, onto this machine, and got 265 gigaflops sustained performance for his application. This is about 100 times the speed that he had achieved on the CPU before. His impressive results are part of the benchmark suite developed in the courses.
The G80 chip is massively parallel, with 128 processor cores. Because it is massively threaded, it sustains thousands of threads per application. A good application will run 5,000 to 12,000 threads simultaneously on this chip. For those who are used to simultaneous multithreading, note that Intel supports 2 or 4 threads, depending on the machine model, per core. The G80 chip supports up to 768 threads per core, and 128 altogether, which adds up to about 12,000 threads from this chip. It is very important to understand this particular view so that you can write effective programs.

1.3. Why more speed or parallelism?

One might ask why applications will continue to demand increased speed of computing systems. Many applications that we have today seem to be running quite fast enough. As we will discuss in case studies, when an application is suitable for GPU execution, a good implementation on a GPU can achieve more than 100 times (100x) of speedup over a CPU. If the application includes what we call “data parallelism,” it’s a simple task to achieve a 10x speedup with just a few hours of work. For anything beyond that, we invite you to keep reading!

The answer to why more speed is in new, innovative applications. Despite the myriad of computing applications in today’s world, the exciting applications of the future will be what we currently consider “supercomputing applications.” For example, the biology world 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 what we can do with these instruments. They can be greatly improved by using a model to simulate the underlying system with boundary conditions set to enable the simulation. From the simulation we can measure even more details, more principles, and more hypothesis verification than can ever be imagined with direct instrumentation alone. These simulations will continue to benefit from the increasing computing power in the foreseeable future in terms of the size of the biological system and the amount of reaction time that can be simulated within tolerable response time. The enhancements will have tremendous implications to science and medicine.

For applications such as video and audio coding and manipulation, try to compare your satisfaction with digital high-definition (HD) TV vs. older technology. Once you 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. Massively parallel processors will continue to enhance the size and fidelity of the pictures of HDTVs in the coming years.

Among the benefits offered by more computing speed are much better user interfaces. Consider Apple’s I-Phone interfaces, compared to other cell phones, even though the I-
Phone still has a limited window. Doubtlessly, future versions of these devices will incorporate higher definition and three-dimensional perspectives, requiring even more computing speed.

We are just at the beginning of these developments, consistent with the new but increasing demands of consumer gaming physics. Imagine driving a car in a game today: the game is in fact simply a prearranged set of scenes. If you bump into an obstacle, the course of your driving does not change; only the game score changes. Your wheels do are bent or damaged, and it’s 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. You can expect to see more of these realistic effects in the future: accidents will damage your wheels and your online driving experience will be affected.

All the new applications that we mentioned actually simulate a concurrent world in different ways and at different levels, with tremendous amounts of data being processed. And with this huge quantity of data, much of the computation can be done on different parts of the data in parallel, although they will have to be reconciled at some point. But techniques for doing that are well known to those who work with such applications regularly. Thus, various granularities of parallelism do exist, but the programming model must not hinder parallel implementation, and the data delivery must be properly managed.

How many times speedup can be expected from this type of 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 29.7%. The speed up for the entire application will be only 1.4X. In fact, even infinite amount of speedup in the parallel portion can only slash less 30% off execution time. On the other hand, if 99% of the execution time is in the parallel portion, a 100X speedup will reduce the application execution to 1.99% of the original time. This gives the entire application a 50X speedup. Conversely, it is very important that an application had the vast majority of its execution in parallel portion for a massively parallel processor to effectively speedup its execution.

Researchers at Illinois have achieved speedups of more than 100x for some applications. However, this is typically achieved only after extensive optimization and tuning even after the algorithms have been enhanced so that more than 99.9% of the application execution time is in parallel execution. In general, applications often saturate the memory (DRAM) bandwidth, resulting in about 10X speedup rather. The trick is to figure out how to get around memory bandwidth limitations, which involved 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, optimize the code to get around limitations such as limited on-chip memory capacity. Those developers who successfully got around these limitations got 25x - 400x speedups. Our goal is to help you to achieve the same.
Keep in mind that the level of speedup achieved over CPU execution can also reflect the suitability of the CPU to the application: in some applications, CPUs perform very well, making it harder to speed up performance using 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. This is precisely what the CUDA programming model promotes, as we will further explain in the book.

Figure 1.4 illustrates the key parts of a typical application. Much of a real application’s code tends to sequential. These portions 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 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. For example, most of today’s medical imaging applications are still running on combinations of microprocessor clusters and special-purpose hardware. The cost and size benefit of the GPUs can drastically improve the quality of these applications. Early GPGPUs cover a small variety of such meat portion, which accounts for only a small portion of the most exciting applications coming in the next ten years. As we will see, the CUDA programming model is designed to cover a much larger variety of the peach meat portions of exciting application.

1.3. 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. Someone once said that if you don’t care about performance, parallel programming is very easy. You can literally develop a parallel program in an hour. But we’re going to dedicate many pages to showing you how to do high-performance parallel programming. And, we believe that it is still very easy if you have the right insight and go about it the right way. In particular, we will focus on computational thinking techniques
that will enable you to think about problems in ways that are amenable to parallel computing.

Note that hardware architecture features have constraints. For high-performance parallel programming on most of the chips that will come out in the next five to ten years, you will need some knowledge of how the architecture actually works. It will probably take ten more years before we can build tools and machines so that most programmers can work without this knowledge. But for our purposes, that won’t be necessary. We will aim to complete a suite of API programming tools and techniques at least once, so that you will be able to apply the experience to other APIs and other tools in the future.

Our second goal is to teach parallel programming for correct functionality and dependability, 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 later debug the code, reproduce the bugs when they reappear, and support the code. We will show that with the CUDA programming model that focuses on data parallelism, one can achieve both high-performance and high-reliability in their applications.

Our third goal is scalability across future hardware generations by exploring ways to design architecture and do parallel programming so that future machines, which will be more and more parallel, can take advantage of your code. We want to help you to master parallel programming so that you can achieve high performance regardless of the particular hardware you’re working on. We want you to be able to write code that will be able to scale up to the level of performance of new generations of machines.

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

1.4. Organization of the Book

Chapter 2 introduces CUDA programming. This chapter relies on the fact that students have had previous experience with C programming. It first introduces CUDA as a simple, small extension to C and an instance of widely used Single Program Multiple Data (SPMD) parallel programming models. 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 individual threads in the parallelized part, (5) launching the execution of kernel function by parallel threads, and (6) eventually transferring the data back to the host processor with an API function call.

Chapter 2 also covers the use of the CUDA software development kit (SDK) to compile, link, run, and debug the programs. While the objective of Chapter 2 is to prepare the students for writing and debugging a simple parallel CUDA program, it actually covers several basic skills needed to develop a parallel application based on any parallel programming model. We use a running example of matrix multiplication to make this lecture concrete.

Chapters 3 through 6 give students more in-depth understanding of the CUDA programming model. Chapter 3 covers the thread organization and execution model required for students to fully understand the execution behavior of threads and prepare them for the performance concepts. Chapter 4 is dedicated to the special memories that can be used to hold CUDA variables for improved program execution speed. Chapter 5 introduces the major factors that contribute to the performance of a CUDA kernel function. Chapter 6 introduces the basic concept of floating-point representation and computation so that students can understand concepts such as precision and accuracy.

While these chapters are based on CUDA, they provide a solid foundation for students to understand parallel programming in general. We believe that students understand best when they learn from the bottom up. That is, they must first learn the concepts in the context of a particular programming model, which provides them with solid footing when they generalize their knowledge to other programming models. As they do so, they can draw on their concrete experience from the CUDA model. An in-depth experience with the CUDA model also enables them to gain maturity, which will help them learn concepts that may not even be pertinent to the CUDA model.

Chapter 7 introduces the fundamentals of parallel programming and computational thinking. It does so by covering the concept of organizing the computation tasks of a program so that one can more easily identify high-level parallel execution opportunities. We start by discussing the translational process of organizing abstract scientific concepts into computational tasks, which is a very important first step in producing quality scientific application software, serial or parallel. It is also an important step towards understanding computational experimentation; students develop better intuition with the capabilities and limitations of computational models.

Computational thinking is arguably the most important skill for computational scientists. Like any other thought process or problem-solving skill, computational thinking is an art that varies across scientific disciplines. We believe that the skill can be best taught through a combination of basic principles, systematic case studies, and hands-on exercise. In terms of basic principles, we discuss the major steps in decomposing a large computational problem into smaller, coordinated tasks that can each be realized with numerical methods.
and well-known algorithms. When writing parallel programs, it is particularly important for the programmer to analyze and understand the structure of the domain problem: which parts are inherently serial tasks, and which parts are amenable to parallel execution. As part of computational thinking, we hope to teach the students to analyze the problem structure and plan the strategy for organizing their serial host and parallel device computation. This is not just a necessary step before parallel programming, and it is also a valuable skill that will serve them well throughout their computing careers.

Chapter 8 discusses parallel algorithm structures and their effects on application performance. It is grounded in students’ performance tuning experience with CUDA. Starting with concrete needs and observations, students are motivated to understand more deeply the connection between algorithmic steps and execution performance. Through the course of the lecture, students develop the ability to reason about the performance effects of their own algorithms and implementation decisions.

Chapter 9 covers parallel programming styles, enabling students to place their knowledge in a wider context. With this lecture, students begin to broaden their knowledge from the SPMD programming style to other styles of parallel programming, such as loop parallelism in OpenMP and fork-join in p-thread programming [6]. The main objective of the lecture is to make an initial connection, so students can understand how these models relate to each other. Although we do not go deeply into these alternative parallel programming styles, we expect that the students will be able to learn to program in any of them with the foundation they gain in this course.

Chapters 9 and 10 are case studies of two real applications [7, 8, 9], which take students 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 reason about the advantages and disadvantages of each alternative. We then go through the steps of code transformation needed to achieve high performance. These two lectures help students put all the materials from the previous lectures together and prepare for their own application development projects.
Chapter 2
CUDA Programming Model

To a CUDA programmer, the computing system consists of a host that is a traditional Central Processing Unit (CPU), such an Intel Architecture microprocessor in personal computers today, and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, there are often program sections that exhibit rich amount of data parallelism, a property where many arithmetic operations can be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Since data parallelism plays such an important role in CUDA, we will first discuss the concept of data parallelism before introducing the basic features of CUDA.

2.1. Data Parallelism

Many software applications that process a large amount of data, and thus incur long execution time on today’s computers, are designed to model real-world, physical phenomena. Images and video frames are snap shots of a physical world where different parts of a picture capture simultaneous, independent physical events. Rigid body physics and fluid dynamics model natural forces and movements that can be independently evaluated within small time steps. Such independent evaluation is the basis of data parallelism in these applications.

As we mentioned earlier, data parallelism refers to the program property where many arithmetic operations can be safely performed on the data structures in a simultaneous manner. We illustrate the concept of data parallelism with a matrix multiplication example in Figure 2.1. In this example, each element of the product matrix P is generated by performing a dot product between a row of input matrix M and a column of input matrix N. This is illustrated in Figure 2.1, where the highlighted element of P is generated by taking the dot product of the highlighted row of M and the highlighted column of N. Note that the dot product operations for computing different P elements can be simultaneously performed. That is, none of these dot products will affect the results of each other. For large matrices, the number of dot products can be very large. For example, for a 1,000 X
1,000 matrix multiplication, there are 1,000,000 independent dot products, each involves 1,000 multiply and 1,000 accumulate arithmetic operations. Therefore, matrix multiplication of large dimensions can have very large amount of data parallelism. By executing many dot products in parallel, a CUDA device can significantly accelerate the execution of the matrix multiplication over a tradition host CPU. The data parallelism in real applications is not always as simple as that in our matrix multiplication example. In a later chapter, we will discuss these more sophisticated forms of data parallelism.

![Figure 2.1. Data parallelism in matrix multiplication.](image)

### 2.2. CUDA Program Structure

A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU. The phases that exhibit little or no data parallelism are implemented in host code. The phases that exhibit rich amount of data parallelism are implemented in the device code. The program supplies a single source code encompassing both host and device code. The NVIDIA C Compiler (NVCC) separates the two. The host code is straight ANSI C code and is compiled with the host's standard C compilers and runs as an ordinary process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called *kernels*, and their associated data structures. The device code is typically further compiled by the NVCC and executed on a GPU device. In situations where there is no device available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU. We will discuss this option in more detail in Chapter mCUDA.
The kernel functions, or simply kernels, typically generate a large number of threads to exploit data parallelism. In the matrix multiplication example, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of the output (P) matrix. In this example, the number of threads used by the kernel is a function of the matrix dimension. For a 1,000 X 1,000 matrix multiplication, the kernel that uses one thread to compute one P element would generate 1,000,000 threads when it is invoked. It is worth noting that CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically take thousands of clock cycles to generate and schedule.

The execution of a typical CUDA program is illustrated in Figure 2.2. The execution starts with host (CPU) execution. When a kernel function is invoked, the execution is moved to a device (GPU), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a grid. Figure 2.2 shows the execution of two grids of threads. We will discuss how these grids are organized soon. When all threads of a kernel complete their execution, the corresponding grid terminates, the execution continues on the host until another kernel is invoked.

![Figure 2.2. Execution of a CUDA program.](image)

### 2.3 A Matrix Multiplication Example

At this point, it is worthwhile to introduce a code example that concretely illustrates the CUDA program structure. Figure 2.3 shows a simple host code skeleton for matrix multiplication. For simplicity, we assume that the matrices are square in their shapes with the dimension of each matrix specified by a parameter `width`. 

© David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008
```c
int main(void) {
    // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N

    // M*N on the device
    MatrixMulOnDevice(M, N, width);

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

Figure 2.3 A simple CUDA host code skeleton for matrix multiplication.

The main program first allocates the M, N, and P matrices and then performs I/O to read in M and N, in Part 1. These are ANSI C operations so we are not showing the actual code for simplicity. The detailed code of the main function and some user defined ANSI C function is shown in Appendix I. Similarly, after completing the matrix multiplication, the main function performs I/O to write the product matrix P and the free all the allocated matrices. The details of Part 2 are also shown in Appendix I. Part 2 is the main focus of our example. It calls a function, MatrixMulOnDevice() to perform matrix multiplication on a device. We will use more details of MatrixMulOnDevice() to explain the basic CUDA programming model.

### 2.4. Device Memories and Data Transfer

In CUDA, host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own Dynamic Random Access Memory (DRAM). For example, the NVIDIA GeForce 8800 GTX card that we will use as the device through the book comes with 768 MB (million bytes, or mega-bytes) of DRAM. In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer the pertinent data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer result data from device back to the host and free up the device memory that is no longer needed. The CUDA runtime system provides Application Programming Interface (API) function calls to perform these activities for use by the programmer.

Figure 2.4 shows an overview of the CUDA device memory model for programmers to reason about the allocation, movement, and usage of the various memory types available on a device. At the bottom of the picture, we see global memory and constant memory. These are the memories that the host code can write (W) to and read (R) from. Constant memory allow read-only access by the device; we will describe them in Chapter [CUDA-Memoy]. For now, we will focus on the use of Global memory. Note that the host memory is not explicitly shown in Figure 2.4, but assumed to be contained in the host.
• Device code can:
  – R/W per-thread registers
  – R/W per-thread local memory
  – R/W per-block shared memory
  – R/W per-grid global memory
  – Read only per-grid constant memory

• Host code can
  – R/W per grid global and constant memories

Figure 2.4 Overview of the CUDA device memory model.

The concept CUDA memory model is supported by the API functions that can be called by CUDA programmers. Figure 2.5 shows the two most important API functions for allocating and de-allocating device Global Memory. Function cudaMemcpy() can be called from the host code to allocate a piece of Global Memory for an object. The first parameter for the cudaMemcpy() function is the address of a pointer that needs to point to the allocated object after a piece of Global Memory is allocated to it. The second parameter gives the size of the object to be allocated.

The use of cudaMemcpy() can be illustrated with a simple code example that continues from Figure 2.3. Let’s assume that a programmer wishes to perform a 64x64 single-precision matrix multiplication on the device and have a pointer variable Md that can point to the first element of a single precision array. For clarity, we will end a variable with letter “d” to indicate that the variable is used as an object in the device memory space. In the following code example, we assume that Width is a variable or constant that is set to 64. The programmer specifies that Md is the pointer that should point to the data region allocated for the 64x64 matrix. Since Width is set at 64, the size of the allocated array will be 64*64*(size of a single-precision floating number). After the computation, cudaMemcpy() is called with pointer Md as input to free the storage space for the 64x64 matrix from the Global Memory.

```c
float *Md
int size = Width * Width * sizeof(float);

cudaMalloc((void**)&Md, size);
...
cudaFree(Md);
```
• `cudaMalloc()`
  - Allocates object in the device Global Memory
  - Two parameters
    • Address of a pointer to the allocated object
    • Size of allocated object

• `cudaFree()`
  - Frees object from device Global Memory
  • Pointer to freed object

Figure 2.5 CUDA API Functions for Device Global Memory Management.

Once a program has allocated device memory for the data objects, it can request that data be transferred from the host to the device memory. This is accomplished by calling one of the CUDA API functions for data transfer between memories. Figure 2.6 shows the API function for such data transfer. The `cudaMemcpy()` function requires four parameters. The first parameter is a pointer to the source data object to be copied. The second parameter points to the destination location for the copy operation. The third parameter specifies the number of bytes to be copied. The fourth parameter indicates the types of memory involved in the copy: from host memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory. For example, the memory copy function can use used to copy data from one location of the device memory to another location of the device memory.

For the matrix multiplication example, the host code calls the `cudaMemcpy()` function to copy `M` and `N` matrices from the host memory to the device memory before the multiplication and then to copy the `P` matrix from the device memory to the host memory after the multiplication is done. Assume that `M`, `P`, `Md`, `Pd` and `size` have already been set as we discussed before, the two function calls are shown below. Note that the two symbolic constants, `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`, are recognized, predefined constants of the CUDA programming environment. 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.

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
```
CUDA API Functions for Data Transfer Between Memories.

Now we are ready to complete the details of invoking a kernel in the matrix multiplication example. As shown in Figure 2.3, the host code calls matrixMulOnDevice(), which is also executed on the host. It is responsible for allocating device memory, performing data transfers, and then activating the kernel that performs the actual matrix multiplication. We often refer to this type of host code as the stub function for invoking a kernel. After the matrix multiplication, matrixMulOnDevice() also At this point, the reader should be able to write this function. We show the the function in Figure 2.7. The code consists of three parts. The first part allocates device memory for Md, Nd, and Pd, the device counter part of M, N, and P and transfer M to Md and N to Nd. The second part actually invokes the kernel and will be described later. The third part reads the product from device memory variable Pd to host memory variable P so that the value will be available to main(). It then frees Md, Nd, and Pd from the device memory.
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)  
{
    int size = Width * Width * sizeof(float);

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

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

    2. // Kernel invocation code – to be shown later
    ...

    3. // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device matrices
    cudaMemcpy(Md, Md, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(Nd, Nd, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(Pd, Pd, size, cudaMemcpyDeviceToHost);

}

Figure 2.7 The MatrixMulOnDevice() Function.

2.5. Kernel Functions and Threading

We now discuss the CUDA kernel functions and the organizations of threads generated by
the invocation of kernel functions. In CUDA, a kernel function specifies the code to be
executed by all threads of a parallel phase. Since all threads of a parallel phase execute
the same code, CUDA programming is an instance of the well-known Single-Program
Multiple-Data (SPMD) [algorithms:98:crc] parallel programming style, a popular
programming style for massively parallel computing systems.

Figure 2.8 shows the kernel function for matrix multiplication. The syntax is ANSI C with
some notable extensions. First, there is a CUDA specific keyword “__global__” in front of
the declaration of MatrixMulKernel(). This keyword indicates that the function is a kernel
and that it can be called from a host functions to generate a grid of threads.

The second notable extension to ANSI C is the keywords “threadIdx.x” and “threadIdx.y”
that refer to the thread indices of a thread. Note that all threads execute the same kernel
code. There needs to be a mechanism to allow them to distinguish themselves and direct
themselves towards the particular parts of the data structure they are designated to work
on. These keywords allow a thread to access the hardware registers associated with it at
runtime that provides the identity to the thread. For simplicity, we will refer to a thread as
Thread[threadIdx.x, threadIdx.y]. Note that the indices reflect a multi-dimensional organization for
the threads. We will come back to this point soon.
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

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

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

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

Figure 2.8 The Matrix Multiplication Kernel Function.

In Figure 2.8, each thread uses the two indices to identify the row of Md and the column of Nd to perform dot product operation in the for loop and to select the Pd element that it is responsible for. It does so by calculating the starting positions in the input matrices based on their unique block and thread coordinates. They then iterate through a loop to calculate the result, and store it to memory. For example, Thread 2,3 will perform a dot product between row 2 of Md and column 3 of Nd and write the result into element (2,3) of Pd. This way, the threads collectively generate all the elements of the Pd matrix.

When a kernel is invoked, or launched, it is executed as grid of parallel threads. In Figure 2.9, the launch of Kernel 1 creates Grid 1. Each CUDA thread grid typically comprises thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example each element of a large array might be computed in a separate thread.

Threads in a grid are organized into a two-level hierarchy, as illustrated in Figure 2.9. For simplicity, the number of threads shown in Figure 2.9 is set to be small. In reality, a grid will typically consist of many more threads. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. In Figure 2.9, Grid 1 consists of 6 thread blocks that are organized into a 2x3 two-dimensional array of threads. Each thread block has a unique two dimensional coordinate given by the CUDA specific keywords blockIdx.x and blockIdx.y. All thread blocks must have the same number of threads organized in the same manner. For simplicity, we assume that the kernel
in Figure 2.8 is launched with only one thread block. It will become clear soon that a practical kernel will create much large number of thread blocks.

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

**Figure 2.9 CUDA Thread Organization.**

Each thread block is in turn organized as a three dimensional array of threads with a total size of up to 512 threads. The coordinates of threads in a block are uniquely defined by three thread indices: threadIdx.x, threadIdx.y, and threadIdx.z. Not all applications will use all the three dimensions of a thread block. In Figure 2.9, each thread block uses only two of the dimensions and is organized into a 3x5 array of threads. This gives Grid 1 a total of 15*6=90 threads. This is obviously a toy example.

In the matrix multiplication example, a grid is invoked to compute the product matrix. The code in Figure 2.8 can use only one thread block organized as a 2-dimensional array of threads in the grid. Since a thread block can have only up to 512 threads and each thread is to calculate one element of the product matrix, the code can only calculate a product matrix of up to 512 elements. This is obviously not acceptable. As we explained before, the product matrix needs to have millions of elements in order to have sufficient amount of data parallelism to benefit from execution on a device. We will come back to this point in Chapter [CUDA threading model] and discuss the use of multiple blocks.
// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

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

Figure 2.10 Example of host code that launches a kernel.

When the host code invokes a kernel, it sets the grid and thread block dimensions by passing them as parameters. This is illustrated in Figure 2.10. Two structures of type \textit{dim3} are declared: the first is for blocks, which are defined as 16x16 groups of threads. There is a thread computing each element of the result matrix. The final line of code invokes the kernel. The special syntax between the name of the kernel function and the traditional C parameters of the function is a CUDA extension to ANSI C. It provides the dimensions of grids in terms of number of blocks and the dimensions of blocks in terms of number of threads.

\section*{2.6 Summary}

We have now finished an overview tour of the CUDA programming model. The matrix multiplication program developed through the chapter is a fully functional CUDA program. You can now compile the code and run the code using the CUDA runtime system. In the next few chapters, we will give more complete description of each of the main aspects of CUDA and begin to learn about the techniques for writing high performance CUDA applications.
Chapter 3
CUDA Threads

Fine-grained, data-parallel threads are the fundamental means of parallel execution in CUDA. As we explained in Chapter 2, launching a CUDA kernel creates a grid of threads that all execute the kernel function. That is, the kernel function specifies the statements that are executed by each individual thread created when the kernel is launched at run-time. This chapter presents more details on the organization, resource assignment, and scheduling of threads in a grid. A CUDA programmer who understands these details are well equipped to writing and understanding efficiency CUDA applications.

3.1. CUDA Thread Organization

Since all threads in a grid execute the same kernel function, they rely on unique 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 using unique coordinates, called blockIdx and threadIdx, assigned to them by the CUDA runtime system. The blockIdx and threadIdx appear as built-in variables that are initialized by the runtime system and can be accessed within the kernel functions. When a thread executes the kernel function, references to the blockIdx and threadIdx variables return the appropriate values that form coordinates of the thread.

At the top level of the hierarchy, a grid is organized as a two dimensional array of blocks. The number of blocks in each dimension is specified by the first special parameter given at the kernel launch. For the purpose of our discussions, we will refer to the special parameters that specify the number of blocks in each dimension as a struct variable gridDim, with gridDim.x specifying the number of blocks in the x dimension and gridDim.y the y dimension. The values of gridDim.x and gridDim.y can be anywhere between 1 and 65,536. The values of gridDim.x and gridDim.y can be supplied by run-time variables at kernel launch time. Once a kernel is launched, its dimensions cannot change in the current CUDA run-time implementation. All threads in a block share the same blockIdx values. The blockIdx.x value ranges between 0 and gridDim.x-1 and the blockIdx.y value between 0 and gridDim.y-1.
Figure 3.1 shows a small grid that consists of four blocks organized into a 2X2 array. Each block in the array is labeled with (blockId.x, blockId.y). For example, Block(1,0) has its blockId.x=1 and blockId.y=0. It should be clear to the reader that the grid was generated by launching the kernel with both gridDim.x and gridDim.y set to 2. We will show the code that does so momentarily.

At the bottom level of the hierarchy, all blocks of a grid are organized into a three-dimensional array of threads. All blocks in a grid have the same dimensions. Each threadId consists of three components: the x coordinate threadId.x, the y coordinate threadId.y, and the z coordinate threadId.z. The number of threads in each dimension of a block is specified by the second special parameter given at the kernel launch. For the purpose of our discussion, we refer to the second special parameter as blockDim variable given at the launch of a kernel. The total size of a block is limited at 512 threads, with total flexibility of distributing these elements into the three dimensions as long as the total number of threads does not exceed 512. For example, (512,1,1), (8, 16, 2) and (16,16, 2) are all allowable dimensions but (32, 32, 1) is not allowable since the total number of threads would be 1024.

Figure 3.1 also illustrates the organization of threads within a block. Since all blocks within a grid have the same dimensions, we only need to show one of them. In this example, each block is organized into 4X2X2 arrays of threads. Figure 3.1 expands block(1,1) by showing this organization of all 16 threads in block(1,1). For example, thread(2,1,0) has its threadId.x=2, threadId.y=1, and threadId.z=0. Note that in this example, we have 4 blocks of 16 threads each, with a grand total of 64 threads in the grid. Note that we use these small numbers to keep the illustration simple. Typical CUDA grids contain thousands to millions of threads.

We now come back to the point that the exact organization of a grid is determined by the special parameters provided during kernel launch. The first special parameter of a kernel
launch specifies the dimensions of the grid in terms of number of blocks. The second specifies the dimensions of each block in terms of number of threads. Each such parameter is a dim3 type, which is essentially a struct with three fields. Since grids are 2D array of block dimensions, the third field of the grid dimension parameter is ignored; one should set it to one for clarity. At this point, the reader should be able to tell that the thread organization shown in Figure 3.1 is created through a kernel launch of the following form:

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

The first two statements initialize the dimension parameters. The third statement is the actual kernel launch.

In situations where a kernel does not need one of the dimensions given, the programmer can simply initialize that field of the dimension parameter to 1. For example, if a kernel is to have a 1D grid of 100 blocks and each block has 16X16 threads, the kernel launch sequence can be done as follows:

```c
    dim3 dimBlock(16, 16, 1);   
    dim3 dimGrid(100, 1, 1);   
    KernelFunction<<dimGrid, dimBlock>>>(...);
```

Note that the dimension variables can be given as contents of variables. They do not need to be compile-time constants.

### 3.2. More on BlockId and ThreadId

From the programmer’s point of view, the main functionality of blockId andThreadId variables is to provide threads with a means to distinguish among themselves when executing the same kernel. One common usage for threadId and blockId is to determine the area of data that a thread is to work on. This was exemplified by the simple matrix multiplication code in Figure 2.8, where the dot product loop uses threadId.x and threadId.y to identify the row of Md and column of Nd to work on. We will now cover more sophisticated usage of these variables.

One limitation of the simple code in Figure 2.8 is that it can only handle matrices of up to 16 elements in each dimension. This limitation comes from the fact the code uses only one block of threads to calculate Pd. Since the kernel function does not use blockId, all threads implicitly belong to the same block. With each thread calculating one element of Pd, we can calculate up to 512 Pd elements with the code. For square matrices, we are limited to 16X16 since 32X32 results in more than 512 Pd elements.
In order to accommodate larger matrices, we need to use multiple blocks. Figure 3.2 shows the basic idea of such an approach. Conceptually, we break Pd into square tiles. All the Pd elements of a tile are computed by a block of threads. By keeping the dimensions of these Pd tiles small, we keep the total number of threads in each block under 512, the maximal allowable block size. In Figure 3.2, for simplicity, we abbreviate threaded.x and threaded.y into tx and ty. Similarly, we abbreviate blockId.x and blockId.y into bx and by.

Each thread still calculates one Pd element. The difference is that it needs to use its blockId values to identify the tile that contains its element before it uses its threadId values to identify its element inside the tile. That is, each thread now uses both threadId and blockId to identify the Pd element to work on. This is portrayed in Figure 3.2 with the blockId and threadId values of threads calculating the Pd elements marked in both x and y dimensions. All threads calculating the Pd elements within a tile have the same blockId values. Assume that the dimensions of a block are specified by variable tile_width. Each dimension of Pd is now divided into sections of tile_width elements each, as shown on the left and top edges of Figure 3.2. Each block handles such a section. Thus, a thread can find the x index of its Pd element as (bx*tile_width + tx) and the y index as (by*tile + ty). That is, thread(tx,ty) in block(bx,by) is to calculate Pd[bx*tile_width+tx][by*tile_didtch+ty].
Figure 3.3 shows an example of using multiple blocks to calculate Pd. For simplicity, we use a very small tile_width value (2). The Pd matrix is now divided into 2x2 tiles. Each dimension of Pd is now divided into sections of two elements. Each block now needs to calculate four Pd elements. We can do so by creating blocks of four threads, organized into a 2x2 array each; with each thread calculating one Pd element. In the example, thread(0,0) of block(0,0) calculates Pd[0][0] whereas thread(0,0) of block(1,0) calculates Pd[2][0]. It is easy to verify that one can identify the Pd element calculated by thread(0,0) of block(1,0) with the formula given above: P[by*tile_width+ty][bx*tile_width+ty] = P[1*2+0][0*2+0] = P[2][0]. The reader should work through the index derivation for as many threads as it takes to become comfortable with the concept.

Once we identified the indices for the Pd element calculated by a thread, we also have identified the row (y) index of Md and the column (x) index of Nd as input values needed for calculating the Pd element. As shown in Figure 3.2, the row index of Md used by thread(tx,ty) of block(bx,by) is (by*tile_width+ty). The column index of Nd used by the same thread is (bx*tile_width+tx). We are now ready to revise the kernel of Figure 2.8 into a version that uses multiple blocks to calculate Pd. Figure 3.4 shows such a revised matrix multiplication kernel function.
global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
    // Calculate the row index of the Pd element and M
    int Row = blockId.y * TILE_WIDTH + threadId.y;
    // Calculate the column index of Pd and N
    int Col = blockId.x * TILE_WIDTH + threadId.x;
    Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row][k] * Nd[k][Col];
    Pd[Row][Col] = Pvalue;
}

Figure 3.4 Revised Matrix Multiplication Kernel using multiple blocks.

In Figure 3.4, each thread uses its blockId and threadId values to identify the row index and the column index of the Pd element it is responsible for. It then performs a dot product on the row of Md and column of Nd to generate the value of Pd element. It eventually writes the Pd value to the appropriate global memory location. Note that this kernel can handle matrices of up to 16*65,536 elements in each dimension. In the unlikely situation where matrices larger than this new limit are to be multiplied, one can divide up the Pd matrix into sub-matrices whose size is permissible by the kernel. Each sub-matrix would still have ample number of blocks (65,536*65,536) to fully utilize parallel execution resources of any processors in the foreseeable future.

### 3.3. Transparent Scalability

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function syncthreads(). When a kernel function calls syncthreads(), all threads in a block will be held at the calling location until everyone else 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 they all move on to the next phase.

Barrier synchronization is a popular method of coordinating parallel activities. In real life, we often use barrier synchronization to coordinate parallel activities of multiple persons. For example, assume that four friends go to a shopping mall in a car. They can all go to different stores to buy their own clothes. This is a parallel activity and is much more efficient than if they all remain as a group and sequentially visit the stores for their clothes. 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. Without the barrier synchronization, one or more persons can be left in the mall when the car leaves, which can seriously damage the friendship!
The ability of synchronizing with each other 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. CUDA run-time systems satisfy this constraint by assigning execution resources to all threads in a block as a unit. That is, when a thread is 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 waiting time during barrier synchronization.

This leads us to a major tradeoff in the design of CUDA barrier synchronization. By not allowing threads in different blocks to perform barrier synchronization with each other, CUDA run-time system does not need to deal with any constraint while executing different blocks. That is, blocks can execute 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 3.5. In a low-cost implementation with only few execution resources, one can execute a small number of blocks at the same time, shown as executing two blocks a time on the left hand side of Figure 3.5. 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 a time on the right hand side of Figure 3.5. The ability to execute the same application code at a wide range of speeds allows one to produce a wide range of implementations according the cost, power, and performance requirements of particular market segments. For example, one can produce a mobile processor that execute an application slowly but at extremely low power consumption and a desktop processor that executes 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 at different speeds is referred to as transparent scalability, which reduces the burden on application developers and improves the usability of applications.

Each block can execute in any order relative to other blocks.

Figure 3.5 Lack of synchronization across blocks enables transparent scalability of CUDA programs
3.4. Thread Assignment

Once a kernel is launched, the CUDA run-time system generates the corresponding grid of threads. These threads are assigned to execution resources on a block by block basis. In the GeForce-8 series hardware, the execution resources are organized into Streaming Multiprocessors. For example, the GeForce 8800GTX implementation has 16 Streaming Multiprocessors, two of which are shown in Figure 3.6. Up to 8 blocks can be assigned to each SM in the GeForce 8800GTX design as long as there are enough resources to satisfy the needs of all the blocks. In situations where there is an insufficient amount of any 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 Streaming Multiprocessor until the resource usage is under the limit. With 16 Streaming Multiprocessors in a GeForce 8800 GTX processor, up to 128 blocks can be simultaneously assigned to Streaming Multiprocessors. Most grids contain much more than 128 blocks. The run-time system maintains a list of blocks that need to execute and assigns new blocks to Streaming Multiprocessors as they complete the execution of blocks previously assigned to them.

![Figure 3.6 Thread Assignment in GeForce-8 Series GPU Devices.](image)

One of the Streaming Multiprocessor (SM) resource limitations is the number of threads that can be simultaneously tracked and scheduled. It takes hardware resources for Streaming Multiprocessors to maintain the thread and block IDs and track their execution status. In the GeForce 8800GTX design, up to 768 threads can be assigned to each SM. This could be in the form of 3 blocks of 256 threads each, 6 blocks of 128 threads each, etc. It should be obvious that 12 blocks of 64 threads each are not a viable option since each SM can only accommodate up to 8 blocks. With 16 SMs in GeForce 8800 GTX, there can be up to 12,288 threads simultaneously residing in SMs for execution.

3.5. Thread Scheduling

Thread scheduling is strictly an implementation concept and thus must be discussed in the context of specific implementations. In the GeForce 8800GTX, 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 and can vary from one implementation to another.
In fact, warps are not even part of the CUDA language definition. However, knowledge of the warps can be helpful in understanding and optimizing the performance of CUDA applications on GeForce-8 series processors. These warps are the unit of thread scheduling in SMs. Figure 3.7 shows the division of blocks into warps in GeForce 8800GTX. Each warp consists of 32 threads of consecutive threadId 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 in green, Block 2 in orange, and Block 3 in blue, all assigned to an SM. Each of the three blocks is further divided into warps for scheduling purposes.

At this point, the reader should be able to calculate the number warps that reside in a SM for a given size of blocks and a given number of blocks assigned to each SM. For example, in Figure 3.7, if each block has 256 threads, we should be able to determine the number of warps that reside in each Streaming Multiprocessor. Each block has 256/32 or 8 warps. With three blocks in each SM, we have $8 \times 3 = 24$ warps in each SM. This is in fact the maximal number of warps that can reside in each SM in GeForce 8800GTX, since there can be no more than 768 threads in each SM and this amounts to $768/32 = 24$ warps.

To summarize, for the GeForce-8 series processors, there can be up to 24 warps residing in each Streaming Multiprocessor at any point in time. We should also point out that the SMs are designed such that only one of these warps will be actually executed by the hardware at any point in time. A legitimate question is why we need to have so many warps in an SM considering the fact that it executes only one of them at any point in time. The answer is that this is how these processors efficiently execute long latency operations such as access to the global memory. When an instruction executed by threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is placed into a waiting area. One of the other resident warps who are no longer waiting for results is selected for execution. If more than one warp is ready for execution, a priority mechanism is used to select one for execution.
Figure 3.8 illustrates the operation of the warp-based thread scheduling scheme. It shows a snapshot of execution timeline in a Streaming Multiprocessor, where time increases from left to right. At the beginning of the snapshot, Warp 1 of Block 1 is selected for execution. Instruction 7 needs to wait for a result of a long latency operation so the warp is placed into a waiting area. Next, the scheduling hardware selects Warp 1 of Block 2 for execution. Instruction 3 needs to wait for a long latency operation so the warp is placed into a waiting area. During this time, the operation that will ultimately provide value to Instruction 7 of Warp 1 of Block 1 continue to make progress, shown as the stall time marked as “TB1, W1 stall” on top of the timeline in Figure 3.8. When the long-latency operation completes, Instruction 7 of Warp 1 or Block 1 will be ready for execution and will eventually be selected for execution, as shown in the Figure. With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long latency operations. The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as zero-overhead thread scheduling.

3.5. Summary

To summarize, special parameters at a kernel launch define the dimensions of a grid and its blocks. Unique coordinates in blockIdx and threadIdx variables allow threads of a grid to distinguish among them. It is the programmer’s responsibility to use these variables in the kernel functions so that the threads can properly identify the portion of the data to process. These variables compel the programmers to organize threads and their data into hierarchical and multi-dimensional organizations.

Once a grid is launched, its blocks are assigned to Streaming Multiprocessors in arbitrary order, resulting in transparent scalability of applications. The transparent scalability comes with a limitation: threads in different blocks cannot synchronize with each other. The only safe 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 SM for execution on a block-by-block basis. For GeForce-8 processors, each SM can accommodate up to 8 blocks or 768 threads, which ever becomes a limitation first. Once a block is assigned to SM, it is further partitioned into warps. At any time, the SM executes only one 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.
Chapter 4
CUDA Memories

So far, we have learned to write a CUDA kernel function which can be invoked by a massive number of threads. The data to be processed by these threads are first transferred from the host memory to the device global memory. The threads then access their portion of the data from the global memory using block and thread IDs. We have also learned the more details of the assignment and scheduling of threads for execution. Although this is a very good start, these simple CUDA kernels will likely achieve only a small fraction of the potential speed of the underlying hardware. This is due to the fact that global memory, which is typically implemented with Dynamic Random Access Memory (DRAM), tends to have long access latencies (hundreds of clock cycles) and limited 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 multiple Streaming Multiprocessors idle. In order to circumvent such congestion, CUDA provides a plethora of additional types of memories that can filter out a majority of data requests to the global memory. In this chapter, you will learn to use such memories to boost the execution efficiency of CUDA kernels.

4.1. Importance of Memory Access Efficiency

The effect of memory access efficiency can be illustrated by calculating the expected performance level of the simple matrix multiplication kernel code in Figure 3.4, replicated in Figure 4.1. The most important part of the kernel in terms of execution time is the for loop that performs inner product calculation. In every iteration of this loop, two global memory accesses are performed for one multiplication and one addition. 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 (CGMA) ratio, defined as the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.
global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
  // Calculate the row index of the Pd element and M
  int Row = blockId.y * TILE_WIDTH + threadId.y;
  // Calculate the column index of Pd and N
  int Col = blockId.x * TILE_WIDTH + threadId.x;
  Pvalue = 0;
  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row][k] * Nd[k][Col];
  Pd[Row][Col] = Pvalue;
}

Figure 4.1 Revised Matrix Multiplication Kernel using multiple blocks.

CGMA has major implications on the performance of a CUDA kernel. For example, the GeForce 8800GTX processor supports 86.4 Giga (10^9) Bytes per second, or 86.4 GB/s, of global memory access bandwidth. With a CGMA of 1.0 and 4 bytes in each single-precision floating-point datum, one can expect that the matrix multiplication kernel will execute at no more than 21.6 Giga Floating Point Operations per Cycle (GFLOPS), since each floating point operation requires four bytes of global memory data and 86.4/4=21.6. While 21.6 GFLOPS is a respectable number, it is only a tiny fraction of the peak performance of 367 GFLOPS for GeForce 8800GTX. We will need to increase the CGMA ratio in order to achieve a higher level of performance for the kernel.

4.2. CUDA Device Memory Types

Each CUDA device has several memories that can be used by programmers to achieve high CGMA ratio and thus high execution speed in their kernels. Figure 4.2 shows these CUDA device memories as implemented in the GeForce 8800GTX hardware. At the bottom of the picture, we see global memory and constant memory. These are the memories that the host code can write (W) and read (R) by calling API functions. We have already introduced global memory in Chapter 2. The constant memory allows read-only access by the device and provides faster and more parallel data access paths for CUDA kernel execution than the global memory.

Above the thread execution boxes in Figure 4.2 are registers and shared memories. Variables that reside in these memories 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 memories are allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block. Shared memories are efficient means for threads to cooperate by sharing the results of their work.
Each thread can:
- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read/only per-grid constant memory

Figure 4.2 GeForce 8800GTX Implementation of CUDA Memories

Table 1 shows the CUDA syntax for declaring program variables into the various device memories. Each such declaration also gives its declared CUDA variable a scope and lifetime. Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of the entire grid. If a variable’s scope is a single thread, a private version of the variable will be created for each and every thread; every thread can only access its own local 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.

Table 1. CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic variables other than arrays</td>
<td>register</td>
<td>thread</td>
<td>kernel</td>
</tr>
<tr>
<td>Automatic array variables</td>
<td>global</td>
<td>thread</td>
<td>kernel</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>kernel</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

Lifetime specifies the portion of program execution duration when the variable is available for use: either within a kernel’s invocation or throughout the entire application. If a variable’s lifetime is within a kernel invocation, 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 contents of the variable are 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 the variable are maintained throughout the execution of the application and available to all kernels.
As shown in Table 1, all automatic variables except for arrays declared in kernel and device functions are placed into registers. We will refer to variables that are not arrays as *scalar* variables. The scopes of these automatic variables are within individual threads. When a kernel function declares an automatic variable, a private copy of that variable is generated for every thread that executes the kernel function. When a thread terminates, all its automatic variables also cease to exist. In Figure 4.1, variables tx, ty, and Pvalue are all automatic variables and fall into this category. Note that accessing these variables is extremely fast and parallel but one must be careful not to exceed the limited capacity of the register storage in the hardware implementations. We will address this point in Chapter 5.

Automatic array variables are not stored in registers. Instead, they are stored into the global memory and incur long access delays and potential access congestions. The scopes of these arrays are, same as automatic scalar variable, within individual threads. That is, a private version of such array is created and used for every thread. Once a thread terminates its execution, the contents of its automatic array variables also cease to exist. Due to the slow nature of automatic array variables, one should avoid using such variables. From our experience, one seldom needs to use automatic array variables in kernel functions and device functions.

If a variable declaration is preceded by keywords “__shared__” (each “__” consists of two “_” characters), it declares a shared variable in CUDA. One can also add an optional “__device__” in front of “__shared__” in the declaration to achieve the same effect. Such declaration must reside within a kernel function or a device function. 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. Shared variables are an efficient means for threads within a block to collaborate with each other. Accessing to shared memory is extremely fast and highly parallel. CUDA programmers often use shared memory 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 shortly with matrix multiplication.

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 “__device__” in front of “__constant__” to achieve the same effect. Declaration of constant variables must reside 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 variable 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 5.

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 global memory. Accesses to a global variable are very slow. However, global variable 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 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. Therefore, global variables are often used to pass information from one kernel execution to another kernel execution.

Note that there is a limitation on the use of pointers with CUDA variables declared into device memories. Pointers can only be used to point to data object in the global memory. There are two typical ways in which pointers usages arise 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 Md, Nd, and Pd 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;} assigns the address of GlobalVar into an automatic pointer variable ptr.

4.3. A Strategy to Reduce Global Memory Traffic

We have an intrinsic tradeoff in the use of device memories in CUDA: global memory is large but slow whereas the shared memory is small but fast. A common strategy is partition the data into subsets called tiles so that each tile fits into the shared memory. The term tile draws on the analogy that a large wall (i.e., the global memory data) can often be covered by tiles (i.e., subsets that each can fit into the shared memory). An important criterion is that the kernel computation on these tiles can be done independently of each other. Note that not all data structure can be partitioned into tiles given an arbitrary kernel function.

The concept of tiling can be illustrated with the matrix multiplication example. Figure 4.3 shows a small example of matrix multiplication using multiple blocks in Figure 4.1. This example assumes that we use four 2X2 blocks to compute the Pd matrix. Figure 4.3 highlights the computation done by the four threads of block(0,0). These four threads compute Pd_{0,0}, Pd_{1,0}, Pd_{0,1}, and Pd_{1,1}.
Figure 4.3 A small example of matrix multiplication using multiple blocks

Figure 4.4 shows the global memory accesses done by all threads in block\( _{0,0} \). Note that each thread accesses four elements of \( M_d \) and four elements of \( N_d \) during its execution. Among the four threads highlighted, there is a significant overlap of their accesses to \( M_d \) and \( N_d \). For example, thread\( _{0,0} \) and thread\( _{1,0} \) both access \( M_d \) as well as the rest of row 0 of \( M_d \). In Figure 4.1, the kernel is written so that both threads access these \( M_d \) elements from the global memory. If we manage to have thread\( _{0,0} \) and thread\( _{1,0} \) to collaborate so that these \( M_d \) elements are only loaded from global memory once, we can reduce the total number of accesses to the global memory by half. In general, we can see that every \( M_d \) and \( N_d \) element are accessed exactly twice during the execution of block\( _{0,0} \). Therefore, if we can have all the four threads to collaborate in their accesses to global memory, we can reduce the traffic to the global memory by half.

The reader should be able to verify that the potential reduction of global memory traffic in matrix multiplication is proportional to the dimension of the blocks used. With \( N \times N \) blocks, the potential reduction of global memory traffic would be \( N \). That is, if we use
16x16 blocks, one can potentially reduce the global memory traffic to 1/16 through collaboration between threads.

We now present an algorithm where threads collaborate to reduce the traffic to the global memory. The basic idea is to have the threads to collaboratively load Md and Nd elements into the shared memory before they individually use these elements in their dot product calculation. Keep 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 Md and Nd elements into the shared memory. This can be accomplished by dividing the Md and Nd 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 4.5.

In Figure 4.5, we further divide Md and Nd into 2X2 tiles. 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 Md and a tile of Nd into the shared memory. This is done by having every thread in a block to load one Md element and one Nd element into the shared memory, as illustrated in Figure 4.6. Each row of Figure 4.6 shows the execution activities of a thread. We only need to show the activities of threads in block0,0; the other blocks all have similar behavior. The shared memory locations for the Md elements are Mds and Nd elements Nds. At the beginning of Phase 1, the four threads of block0,0 collaboratively loads the a tile of Md into shared memory: thread0,0 loads Md0,0 into Mds0,0, thread1,0 loads Md1,0 into Mds1,0, thread0,1 loads Md0,1 into Mds0,1, and thread1,1 loads Md1,1 into Mds1,1. A tile of Nd is also loaded in a similar manner.

After the two tiles of Md and Nd are loaded into the shared memory, these values are used in the calculation of the dot product. Note that each value in the shared memory is used twice. For example, the Md1,1 value, loaded by Thread1,1 into Mds1,1, is used twice, once by thread0,1 and once by thread1,1. By loading each global memory value into shared memory so that it can be used multiple times, we reduce 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 NxN elements.

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{0.0}$</td>
<td>$T_{0.0}$</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$\downarrow$</td>
</tr>
<tr>
<td>$M_{0.0}$</td>
<td>$M_{0.0}$</td>
</tr>
<tr>
<td>$N_{0.0}$</td>
<td>$N_{0.0}$</td>
</tr>
<tr>
<td>$P_{Value_{0.0}} += M_{0.0} \times N_{0.0}$</td>
<td>$P_{Value_{0.0}} += M_{0.0} \times N_{0.0}$</td>
</tr>
<tr>
<td>$M_{2.0}$</td>
<td>$M_{2.0}$</td>
</tr>
<tr>
<td>$N_{0.2}$</td>
<td>$N_{0.2}$</td>
</tr>
<tr>
<td>$P_{Value_{0.0}} += M_{0.0} \times N_{0.0}$</td>
<td>$P_{Value_{0.0}} += M_{0.0} \times N_{0.0}$</td>
</tr>
</tbody>
</table>

Note that the calculation of each dot product in Figure 4.6 is now performed in two phases. In each phase, products of two pairs of the input matrix elements are accumulated into the PValue variable. In this example, the dot products are done in 2 phases. In an arbitrary case where the input matrix is of dimension N and the tile size is TILE_WIDTH, the dot product would be performed in N/TILE_WIDTH phases. The creation of these phases is key to the reduction of accesses to the global memory. With each phase focusing on a small subset of the input matrix values, the threads can collaboratively load the subset into the shared memory and use the values in the shared memory to satisfy the input needs of the phase of calculations.

Note also that the Mds and Nds locations are re-used to hold the input values. In each phase, the same locations are used to hold the subset of Md and Nd elements used in the phase. This allows a much smaller shared memory to screen away 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 exhibit locality, there is an opportunity to use small, high-speed memories to screen away most accesses to the global memory. We will return to the concept of locality in Chapter 5.

We are now ready to present the tiled kernel function that uses shared memory to reduce the traffic to global memory. This kernel shown in Figure 4.7 implements the phases illustrated in Figure 4.6. In Figure 4.7, Line 1 and Line 2 declare Mds as a shared memory variable. Recall that the scope of shared memory variables is a block. Thus, all threads of a block have access to the same Mds and Nds arrays. This is important since all threads in a
block must have access to the Md and Nd values loaded into Mds and Nds by each other so that they can avoid accessing global memory.

```c
// Block kernel
void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  // Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  int Pvalue = 0;
  // Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
    Md[tx][ty] = Md[m*TILE_WIDTH + tx][Row];
    Nds[tx][ty] = Nd[Col][m*TILE_WIDTH + ty];

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[tx][k] * Nds[k][ty];

    Pd[Row][Col] = Pvalue;
  }
}
```

Figure 4.1 Tiled Matrix Multiplication Kernel using shared memories.

Lines 3 and 4 save the threadId and blockId values into automatic variables and thus into registers for fast access. Recall that automatic non-array 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. They will reside in registers that are accessible by one thread. They are initialized with the threaded and blockId values and used many times during the lifetime of thread. Once the thread ends, the values of these variables also cease to exist.

Lines 5 and 6 identify the row index and column index of the Pd element that the thread is to produce. As shown in Figure 4.8, the column (x) index of the Pd 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 x dimension. A thread in block bx would have bx blocks before it that will cover bx*TILE_WIDTH elements of Pd. Another tx threads within the same block would cover another tx elements of Pd. Thus the thread with bx and tx should be responsible for covering the Pd element whose x index is bx*TILE_WIDTH+tx. For the example of Figure 4.5, the x index of the Pd element to be calculated by thread1,0 of block0,1 is 0*2+1 = 1. Similarly, the y index can be calculated as by*TILE_WIDTH+ty. In Figure 4.5, the y index of the Pd element to be calculated by thread1,0 of block0,1 is 1*2+0 = 2. Thus, the Pd element to be produced by this thread is Pd1,2.
Line 8 of Figure 4.7 shows the loop that iterates through all the phases of calculating the final Pd element. Each iteration of the loop corresponds to one phase of the calculation shown in Figure 4.6. The m variable indicates the number of phases that have already been done for the dot product. Recall that each phase uses one tile of Md and one tile of Nd elements. Therefore, at the beginning of each phase, m*TILE_WIDTH pairs of Md and Nd elements have been processed by previous phases.

Recall that all threads in a grid execute the same kernel function. The threadIdx variable allows them to identify the part of the data they are to process. Also recall that the thread with by=blockId.y and ty=threadId.y is to process row (by*TILE_WIDTH+ty) of Md, as shown at the left side of in Figure 4.8. Line 5 stores this number into the Row variable of each thread. Likewise, the thread with bx=blockId.x and tx=threadId.x is to process column (bx*TILE_WIDTH+tx) of Nd, as shown at the top side of Figure 4.8. Line 6 stores this number into the Col variable of each thread. This will be used when the threads load Md and Nd elements into the shared memory.

In each phase, Line 9 loads the appropriate Md element into the shared memory. Since we already know the row index of Md and column index of Nd elements to be processed by the thread, we will focus on the column index of Md and row index of Nd. As shown in Figure 4.8, each block has TILE_WIDTH^2 threads that will collaborate to load TILE_WIDTH^2 Md elements into the shared memory. Thus, all we need to do is to assign each thread to load one Md element. This is conveniently done using the block and thread IDs. Note that the beginning index of the section of Md elements to be loaded is
m*TILE_WIDTH. Therefore, an easy approach is to have every thread to load an element from that point on identified by the thread ID. This is precisely what we have in Line 9, where each thread loads Md[m*TILE_WIDTH+tx][Row]. Since the value of Row is a linear function of ty, each of the TILE_WIDTH^2 threads will load a unique Md element into the shared memory. Altogether, these threads will load the orange square subset of Md shown in Figure 4.8. The reader should use the small example in Figure 4.5 and Figure 4.6 to verify that the address calculation works correctly.

Once the tiles of Md and Nd are loaded in Mds and Nds, the loop in Line 11 performs the phase of the dot product based on these elements. The progression of the loop for thread(tx,ty) is shown in Figure 4.8, with the direction of the Md and Nd data usage marked with k, the loop variable in Line 11. Note that the data will be accessed from Mds and Nds, the shared memory location holding these Md and Nd elements.

The benefit of the tiled algorithms is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of TILE_WIDTH. If one uses 16X16 tiles, we can reduce the global memory accesses by a factor of 16. This reduction allows the 86.4GB/s global memory bandwidth to serve a much larger floating point computation rate than the original algorithm. More specifically, the global memory bandwidth can now support (86.4/4)*16 = 345.6 GFLOPS, very close to the peak floating-point performance of the GeForce 8800 GTX processor. This effectively removes the global memory bandwidth as the major limiting factor of matrix multiplication performance.

4.4. Memory as a Limiting Factor of Paralleism

While CUDA registers, shared memories, and constant memories can be extremely effective in reducing the number of accesses to the global memory, one must be careful not to exceed the capacity of these memories. Each processor implementation offers a limited amount of CUDA memories, which limits the number threads that can simultaneously reside in the Streaming Multiprocessors for a given application. In general, the more memory locations each thread requires, the fewer the number of threads can reside in each SM, and thus the fewer number of threads that can reside in the entire processor.

In the GeForce 8800 GTX implementation, each SM has 8K registers, which amounts to 128K registers for the entire processor. While this is a very large number, it only allows each thread to use a very limited number of registers. Recall that each SM can accommodate up to 768 threads. In order to achieve this maximal, each thread can use only 8K/768= 10 registers. If each thread uses 11 registers, the number of threads in each SM will be reduced. Such reduction is done at the block granularity. For example, if each block contains 256 threads, the reduction of threads will be done by reducing 256 threads at a time. Thus, the next lower number of threads from 768 would be 512, 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.
Shared memories can also limit the number of threads assigned to each SM. In the GeForce 8800 GTX processor, there are 16K bytes of shared memory in each SM. Keep in mind that shared memory is used by blocks. Recall that each SM 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 such that the total number of shared memories used by these blocks cannot 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, the shared memory can become a limiting factor. For a tile size of 16X16, each block needs a 16X16X4 = 1K bytes of storage of Mds. Another 1KB is needed for Nds. Thus each block uses 2K bytes of shared memory. The 16K bytes of shared memory allows 8 blocks to simultaneous reside in an SM. Since this is the maximum allowed by the threading hardware, shared memory is not a limiting factor for this tile size. If we chose 32X32 tiles, each block needs 32*32*4*2 = 8K bytes of shared memory. Thus, only two blocks would be allowed to reside in each SM.

4.5. Summary
In summary, CUDA defines registers, shared memory, and constant memory that can be accessed at higher speed and in a more parallel manner than the global memory. Using these memories effectively will likely require re-design of the algorithm. We use matrix multiplication as an example to illustrate tiled algorithms, a popular strategy to enable effective use of shared memories. We demonstrate that with 16X16 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 special memories. Their capacities are implementation dependent. Once their capacities are exceeded, they become limiting factors for the number of threads that can be assigned to each SM.