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 Such expectation is no longer valid from this day onward.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 gigaflops 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 beyond 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.
Figure 2.2. CPUs and GPUs have fundamentally different design philosophies.

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 int5ensive 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 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.

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.

Figure 1.3. Architecture of a CUDA-capable GPU
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. Coverage of sequential and parallel application portions

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 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 focus 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 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 as 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 snapshot 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.

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`.
int main(void) {  
  // Allocate and initialize matrices M, N, P  
  // I/O to read the input matrix M and N  
  ...  
  2. // M*N on the device  
     MatrixMulOnDevice(M, N, P, width);  
  3. // 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 then 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 cudaMalloc() can be called from the host code to allocate a piece of Global Memory for an object. The first parameter for the cudaMalloc() 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 cudaMalloc() 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, cudaFree() 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

• cudaMemcpy()
  – 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.

\[
\text{cudaMemcpy}(M, M, \text{size}, \text{cudaMemcpyHostToDevice});
\]
\[
\text{cudaMemcpy}(P, Pd, \text{size}, \text{cudaMemcpyDeviceToHost});
\]
• `cudaMemcpy()`
  – memory data transfer
  – Requires four parameters
    • Pointer to destination
    • Pointer to source
    • Number of bytes copied
    • Type of transfer
      – Host to Host
      – Host to Device
      – Device to Host
      – Device to Device
  – Transfer is asynchronous

Figure 2.6 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
    cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}

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:cr] 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, Thread2,3 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**

![CUDA Thread Organization](image)

**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 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.

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 runtime 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:

```cpp
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:

```cpp
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 and threaded 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.

![Figure 3.2 Matrix Multiplication using multiple blocks by using blockId and tiling Pd.](image)

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].

© David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008
Figure 3.3 shows an example of using multiple blocks to calculate $P_d$. For simplicity, we use a very small tile_width value (2). The $P_d$ matrix is now divided into 2X2 tiles. Each dimension of $P_d$ is now divided into sections of two elements. Each block now needs to calculate four $P_d$ elements. We can do so by creating blocks of four threads, organized into a 2X2 array each; with each thread calculating one $P_d$ element. In the example, thread(0,0) of block(0,0) calculates $P_d[0][0]$ whereas thread(0,0) of block(1,0) calculates $P_d[2][0]$. It is easy to verify that one can identify the $P_d$ element calculated by thread(0,0) of block(1,0) with the formula given above: $P[b\times\text{tile_width}+tx][b\times\text{tile_width}+ty] = P[1\times2+0][0\times2+0] = P[2][0]$. 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 $P_d$ element calculated by a thread, we also have identified the row (y) index of $M_d$ and the column (x) index of $N_d$ as input values needed for calculating the $P_d$ element. As shown in Figure 3.2, the row index of $M_d$ used by thread(tx,ty) of block(bx,by) is $(by\times\text{tile_width}+ty)$. The column index of $N_d$ used by the same thread is $(bx\times\text{tile_width}+tx)$. We are now ready to revise the kernel of Figure 2.8 into a version that uses multiple blocks to calculate $P_d$. 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 = blockIdx.y * TILE_WIDTH + threadIdx.y;
  // Calculate the column index of Pd and N
  int Col = blockIdx.x * TILE_WIDTH + threadIdx.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 blockIdx and threadIdx 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.

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.

© David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008
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*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 \( \text{float* ptr = &GlobalVar;} \) 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. 

© David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008

5
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 \(M_d_{1,0}\) 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 \(N_xN\) 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.

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

// Figure 4.1 Tiled Matrix Multiplication Kernel using shared memories.

global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  // 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];
    Nd[(tx)[ty] = Nd[Col][m*TILE_WIDTH + ty];

    // Compute the Pd element at the given row and column
    Pvalue += Mds[tx][k] * Nds[k][ty];

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

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 threadId and blockIdx 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 \( \text{m} \) 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, \( \text{m} \cdot \text{TILE\_WIDTH} \) 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 \text{threadID} \text{variable allows them to identify the part of the data they are to process.} Also recall that the thread with \( \text{by} = \text{blockId.y} \) and \( \text{ty} = \text{threadId.y} \) is to process row \( (\text{by} \cdot \text{TILE\_WIDTH} + \text{ty}) \) of Md, as shown at the left side of in Figure 4.8. Line 5 stores this number into the \text{Row} variable of each thread. Likewise, the thread with \( \text{bx} = \text{blockId.x} \) and \( \text{tx} = \text{threadId.x} \) is to process column \( (\text{bx} \cdot \text{TILE\_WIDTH} + \text{tx}) \) of Nd, as shown at the top side of Figure 4.8. Line 6 stores this number into the \text{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 \( \text{TILE\_WIDTH}^2 \) threads that will collaborate to load \( \text{TILE\_WIDTH}^2 \) 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.
Chapter 5
Performance Considerations

Although a CUDA kernel can run correctly on any CUDA device, its execution speed can vary greatly depending on the resource constraints of each device. In this chapter, we will discuss the major dimensions of resource constraints in an actual CUDA device and how they can constrain the level of parallel execution in this device. In order to achieve his/her goals, a programmer often has to find ways to achieve a required level of performance that is higher than that of an initial version of the application. In different applications, different constraints may dominate and become the limiting factors. One can improve the performance of an application on a particular CUDA device, sometimes dramatically, by trading one resource usage for another. This strategy works well if the resource constraint thus alleviated was actually the dominating constraint before the strategy was applied and the one thus exacerbated does not have worse effects on parallel execution. Without such understanding, performance tuning would be a guess work; plausible strategies may or may not lead to performance enhancements. Beyond insights into these resource constraints, this chapter further offers principles and case studies designed to cultivate intuition about the type of algorithms that can result in high performance execution CUDA devices. It is also establishes idioms and ideas that will likely lead to good performance improvements during your performance tuning efforts.

5.1. More on Thread Execution

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

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

The G80/G280 implementation bundles several threads for execution. Each block is partitioned into warps. This implementation technique helps to reduce hardware cost and enable some optimizations in servicing memory accesses. In the foreseeable future, we expect that warp partitioning will remain as a popular implementation technique. However, the size of warp can easily vary from implementation to implementation. In G80/G280, each warp consists of 32 threads. We will use the G80/G280 implementation to explain warp partitioning for the rest of this chapter.

Thread blocks are partitioned into warps based on thread IDs. If a thread block is organized into a one-dimensional array, i.e., only threadIdx.x is used, the partition is straightforward. Thread IDs within a warp are consecutive and increasing. For warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63. In general, warp n starts with thread 32*n and ends with thread 32(n+1)-1. For a block whose number of threads is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32 threads. For example, if a block has 48 threads, it will be partitioned into 2 warps, and its warp 1 will be padded with 16 extra threads.

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

![Figure 5.1. Placing threads into linear order](image)
Figure 5.1, shows an example of placing threads of a two dimensional block into linear order. The upper part shows the two-dimensional view of the block. Each thread is shown as $T_{x,y}$, x being the threadIdx.x and y being threadIdx.y for the thread. The lower part shows the linear view of the block. The first four threads are those threads whose threadIdx.y is 0; they are placed with increasing threadIdx.x values. The next four threads are those threads whose threadIdx.y is 1. They are also placed with their increasing threadIdx.x values. For this example, all 16 threads form half a warp. The warp will be padded with another 16 threads to complete a 32-thread warp. Imagine a 2 dimensional block with 8X8 threads. The 64 threads will form 2 warps. The first warp start from $T_{0,0}$ and ends with $T_{3,7}$. The second warp starts with $T_{4,0}$ and ends with $T_{7,7}$. It would be a useful exercise to draw out the picture as an exercise.

For a three dimensional block, we first place all threads whose threadIdx.z is 0 into the linear order. Among these threads, they are treated as a 2-dimensional block as shown in Figure 5.1. All threads whose threadIdx.z is 1 will then be placed into the linear order, and so on. A three dimensional 4X8X2 (4 in the x dimension, 8 in the y dimension, and 2 in the z dimension), the 64 threads will be partitioned into 2 warps, with $T_{0,0,0}$ through $T_{3,7,0}$ in the first warp and $T_{0,0,1}$ through $T_{3,7,1}$ in the second warp.

At any point in time, the hardware selects and executes one warp at a time. An instruction is run for all threads in the same warp, before moving to the next instruction. This style of execution is motivated by hardware cost constraints: it allows the cost of fetching and processing an instruction to be amortized among a large number of threads. It works well when all threads within a warp follow the same control flow path when working their data. For an if-then else construct, the execution works well when either all threads execute the then part or all execute the else part. When threads within a warp take different control flow paths, that is when some threads execute the then part and others execute the else part, the simple execution style no longer works well. In such situation, the execution of the warp will require multiple passes through these divergent paths. One pass will be needed for those threads that follow the then part and another pass for those that follow the else part. These passes are sequential to each other, thus will add to the execution time.

When threads in the same warp follow different paths of control flow, we say that these threads diverge in their execution. Divergence can arise in other constructs. For example, if threads execute a loop whose number of iterations can vary across threads, one additional pass must be taken for each case among the threads. For example, if threads in a warp execute the same for loop whose loop bound can be 6, 7, 8, or 9 iterations. All threads will finish the first 6 iterations together. One pass will be used to execute all threads that need the 7th iteration. One more pass will be used to execute all threads that need the 8th iteration. Yet another pass will be used to execute those that need the 9th iteration.

An if-then else construct can result in thread divergence is when its decision condition is based on thread ID. For example, the statement “if (threadIdx.x > 2) {}” causes the threads to follow two divergent control flow paths. Threads 0, 1, and 2 follow a different path than
threads 3, 4, 5, etc. Similarly, if a loop can cause thread divergence if its loop condition is based on thread ID. Such usages arise naturally in some important parallel algorithms. We will use a reduction algorithm to illustrate this point.

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

When there are a large number of elements in the array, the time needed to visit all elements of an array becomes large enough to motivate parallel execution. A parallel reduction algorithm typically resembles that of a soccer tournament. In fact, the elimination process of the world cup is a reduction of “maximal” where the maximal is defined as the team that “beats” all other teams. The tournament “reduction” is done by multiple rounds. The teams are divided into pairs. During the first round, all pairs play in parallel. Winners of the first round advance to the second round, whose winners advance to the third round, etc. With 16 teams entering a tournament, the 8 winners will emerge from the first round, 4 winners the second round, 2 winners the third round, and 1 final winner the fourth round. It should be easy to see that even with 1024 teams, it takes only 10 rounds to determine the final winner. The trick is to have enough soccer fields to hold the 512 games in parallel during the first round, 256 games in the second round, 128 games in the third round, and so on. With enough fields, even with sixty thousand teams, we can determine the final winner in just 16 rounds. Of course, one would need to have enough soccer fields and enough officials to accommodate the thirty thousand games in the first round, etc.

Figure 5.2 shows a kernel function that performs sum reduction. The original array is in the global memory. Each thread block reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction. The reduction is done in place, which means the elements in the shared memory will be replaced by partial sums. Each iteration of the while loop in the kernel function implements a round of reduction. The syncthreads() statement in the while loop ensures that all threads are ready to enter the next iteration before any thread is allowed to do so. Therefore, all threads that enter the second iteration will be using the values produced in the first iteration. After the first round, the even elements will be replaced by the partial sums generated in the first round. After the second round, the elements whose indices are multiples of four will be
replaced with the partial sums. After the final round, the total sum of the entire section will be in element 0.

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

Figure 5.2 A simple sum reduction kernel.

In Figure 5.2, the Line 3 initializes the stride variable to 1. During the first iteration, the if statement in Line 7 is used to select only the even threads to perform addition between two neighboring elements. The execution of the kernel is illustrated in Figure 5.3. The threads and array elements are shown in the as columns and the iterations are shown as rows. As shown in row 1 in Figure 5.3, the even elements of the array now hold the pair-wise partial sums. Before the second iteration, the value of the stride variable in doubled to 2. During the second iteration, only those threads whose indices are multiples of four, shown as orange and yellow columns in Figure 5.3, will execute the add statement. Each thread
generates a partial sum that covers four elements, as shown in the row 2 in Figure 5.3. With 512 elements in each section, the kernel function will generate the total for the entire section after 9 iterations.

The kernel in Figure 5.3 clearly has thread divergence. During the first iteration of the loop, only those threads whose threadIdx.x are even will execute the add statement. One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute the add statement. In each successive iteration, fewer threads will execute the add statement but two passes will be still needed to execute all the threads. This divergence can be reduced with a slight change to the algorithm.

```
1. __shared__ float partialSum[]

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

Figure 5.4 A kernel with fewer thread divergence

Figure 5.4 shows a modified kernel with a slightly different algorithm for sum reduction. Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other during the first round. It does so by initializing the stride to be half the size of the section. All pairs added during the first round are half the section size away from each other. After the first iteration, all the pair-wise sums are stored in the first half of the array. The loop divides the stride by 2 before entering the next iteration. This is done by shifting the stride value to the right by one bit, a much less expensive way to implement divide by 2 than a real integer division.

Figure 5.5 illustrates the execution of the revised kernel. During the first iteration, all threads whose threadIdx.x value are less than half of the size of the section execute the add statement. For a section of 512 elements, Threads 0 through 255 execute the add statement during the first iteration while threads 256 through 511 do not. The pair-wise sums are stored in elements 0 through 255 after the first iteration. Since the warps consists of 32 threads with consecutive threadIdx.x values, all threads in warps 1 through warp 8 execute
the add statement whereas warps 9 through warp 15 execute all skip the add statement. Since all threads in each warp take the same path, there is no thread divergence!

![Figure 5.5 Execution of the revised algorithm](image)

### 5.2. Global Memory Performance

One of the most important dimensions of CUDA kernel performance is accessing data in the global memory. CUDA applications exploit massive data parallelism that comes from processing as massive amount of data simultaneously. Therefore, CUDA applications typically process a massive amount of data within a short period of time. In particular, a CUDA kernel must be able to access a massive amount of data from the global memory within a very short period of time. In Chapter 4, we discussed tiling techniques that utilize shared memories to reduce the total amount of data that must be accessed by a collection of threads in the thread block. In this Chapter, we will further discuss memory coalescing techniques that can more effectively move data from the global memory into shared memories and registers. Memory coalescing techniques are often used in conjunction with tiling techniques to allow CUDA devices to reach their performance potential in the presence of limited data access bandwidth of the global memory.

Global memory in a CUDA system is typically implemented with Dynamic Random Access Memories, or DRAMs. Data bits are stored in DRAM cells that are very weak capacitors, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1. Reading data from a DRAM cell that contains a 1 requires the weak capacitor to share its tiny amount of charge to a sensor and set off a detection mechanism that determines whether a sufficient amount of charge is present in the capacitor. Because this is a very slow process, modern DRAMs use a parallel process to increase their rate of data access. Each time a location is to be accessed, many consecutive
locations that includes the requested location are accessed. Many sensors are provided in each DRAM chip and they work in parallel, each sensing the contents of a bit location within these consecutive locations. Once detected by the sensors, the data from all these consecutive locations can then be transferred at very high speed to the processor. If an application can make use of data from multiple, consecutive locations before moving on to other locations, the DRAMs can supply the data at much higher rate than if a truly random sequence of locations were accessed. In order to achieve anywhere close to the advertised 84.6GB/sec global memory bandwidth for G80, a kernel must arrange its data accesses so that each request to the DRAMs is for a large number of consecutive DRAM locations.

Recognizing the organization of modern DRAMs, G80/280 designs employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads to exhibit favorable access patterns. This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive global memory locations. That is, the most favorable access pattern is achieved when the same instruction for all threads in a warp accesses consecutive global memory locations. In this case, the hardware combines, or coalesces, all these accesses into a consolidated access to the DRAMs that requests all consecutive locations involved. For example, for a given load instruction of a warp, if thread 0 accesses global memory location N, thread 1 location N+1, thread 2 location N+2, etc, all these accesses will be coalesced, or combined into a single request for all consecutive locations when accessing the DRAMs. Such coalesced access allows the DRAMs to deliver data at a rate close to the maximal global memory bandwidth.

Figure 5.6 illustrates the favorable vs. unfavorable C program matrix data access patterns for memory coalescing. In part (a), illustrates the data access pattern of a loop where each thread reads a row of matrix Md. Assume that threads in a warp read adjacent rows. That is, during iteration 0, threads in warp 0 read element 0 of rows 0 through 31. During iteration 1, these same threads read element 1 of rows 0 through 31. None of the accesses will be coalesced. A more favorable access pattern is shown in Figure 5.6(b), where each thread reads a column of Nd. During iteration 0, threads in warp 0 reads element 1 of columns 0 through 31. All these accesses will be coalesced. In order to understand why the
pattern in 5.6(b) is more favorable than that in 5.6(b), we need to understand how these
data elements are placed into the global memory.

All locations in the global memory form a single, consecutive address space. That is, every
location in the global memory has a unique address. This is analogous to a very long street
where every house has a unique address. For example, if the global memory contains 1,024
locations, these locations will be accessed by address 0 through 1023. The G208 can have
up to 4GB \(2^{32}\) locations; the addresses range from 0 to \(2^{32} - 1\). All variables of a CUDA
program are placed into this linear address space and will be assigned an address.

Matrix elements are placed into the linearly addressed locations according to the *row
major* convention. That is, the elements of row 0 of a matrix are first placed in order into
consecutive locations. They are followed by the elements of row 1 of the matrix, and so on.
In other words, all elements in a row are placed into consecutive locations and entire rows
are placed one after another. The term row major refers to the fact that the placement of
data preserves the structure of rows, all adjacent element in a row are placed into
consecutive locations in the address space. This is illustrated with an example in Figure
5.7, where the 16 elements of a 4X4 matrix M are placed into linearly addressed locations.
The four elements of row 0 are first placed in their order of appearance in the row.
Elements in row 1 are then placed, followed by elements of row 2, followed by elements of
row 3. It should be clear that M0,0 and M0,1, though appear to be consecutive in the two
dimensional matrix, are placed four locations away in the linearly addressed memory.

Now that we understand the placement of matrix elements into global memory, we are
ready to understand the favorable vs. unfavorable matrix data access patterns in Figure 5.6.
Figure 5.8 shows an example of the favorable access pattern in accessing a 4X4 matrix.
The arrow in the top portion of the Figure shows the access pattern of the kernel code for
one thread. The accesses are generated by a loop where threads in a warp access element 0.
of the columns in the first iteration. As shown in the bottom portion of Figure 5.8, these elements are in consecutive locations in the global memory. The hardware detects that these accesses are to consecutive locations in the global memory and coalesces these accesses into a consolidated access. This allows the DRAMs to supply data at high rate.

Figure 5.8 A coalesced access pattern.

Figure 5.9 shows an example of matrix data access pattern that are not coalesced. The arrow in the top portion of the figure shows that the kernel code for each thread accesses elements of a row in sequence. The accesses are generated by a loop where threads in Warp 0 access element 0 of the columns during the first iteration. As shown in the bottom portion of Figure 5.9, these elements are in locations that are four elements away from each other. As a result, the hardware will determine that accesses to these elements cannot be coalesced. As a result, when a kernel loop iterates through a row, the accesses to global memory are much less efficient than the case where a kernel iterates through a column.

Figure 5.9 An un-coalesced access pattern.
If the algorithm intrinsically requires a kernel code to iterate through data within rows, one can use the shared memory to enable memory coalescing. The technique is illustrated in Figure 5.10 for matrix multiplication. Each thread reads a row from Md, a pattern that cannot be coalesced. A tiled algorithm can be used to enable coalescing. As described in Section 4, threads of a block first cooperatively load the tiles into the shared memory. Care can be taken to ensure that these tiles are loaded in a coalesced pattern. Once the data is in shared memory, they can be accessed either on a row basis or a column basis without any performance penalty because the shared memories are implemented as intrinsically high-speed on-chip memory that does not require coalescing to achieve high data access rate.

![Diagram](image)

Figure 5.10 Using shared memory to enable coalescing.

However, one has to take care so that the loading of tiles from global memory to shared memory is coalesced. We replicate Figure 4.7 here as Figure 5.11, where the matrix multiplication kernel loads two tiles of matrix Md and Nd into the shared memory. Note that each thread in a thread block is responsible for loading one Md element and one Nd element into Mds and Nds in each iteration of the for loop defined in line 8. Recall that there are TILE_WIDTH \( ^2 \) threads involved in each tile. The threads use threadIdx.y and threadIdx.x to determine the element of each matrix to load.

For Md, the index calculation for each thread uses m to locate the left end of the tile. Each row of the tile is then loaded by TILE_WIDTH threads whose thread IDs differ in the x dimension. Since these threads have consecutive threadIdx.x values, they are in the same warp. Also, recall that elements in the same row are placed into consecutive locations of the global memory. The hardware detects that these threads in the same warp access consecutive locations in the global memory and combine them into a coalesced access.

In the case of Nd, the index calculation for each thread uses m to locate the upper end of the tile. Each row of the thread is then also loaded by TILE_WIDTH threads whose thread IDs differ only in the x dimension. Once again, these threads are in the same warp because they have consecutive threadIdx.x values. The hardware detects that these threads in the
same warp access consecutive location in the global memory and combine them into a coalesced access.

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
3. int bx = blockIdx.x;  int by = blockIdx.y;
4. int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5. int Row = by * TILE_WIDTH + ty;
6. int Col = bx * TILE_WIDTH + tx;
7. float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {
 // Collaborative loading of Md and Nd tiles into shared memory
9.   Mds[ty][tx] = Md[Row][m*TILE_WIDTH + tx];
10.  Nds[ty][tx] = Nd[m*TILE_WIDTH + ty][Col];
11. __syncthreads();
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue += Mds[ty][k] * Nds[k][tx];
14.   Pd[Row][Col] = Pvalue;
}
}
```

Figure 5.10 Tiled Matrix Multiplication Kernel using shared memories.

The reader shall find it useful to draw a picture based on the kernel code in Figure 5.10 and identify the threadIdx.y and threadIdx.x values of the thread that loads each element of the tile. Lines 5, 6, 9, 10 in Figure 5.10 form a frequently used programming pattern for loading elements into shared memory in tiled algorithms. We would also like to encourage the reader to analyze the data access pattern by the dot-product loop in lines 12 and 13. Note that the threads in a warp do not access consecutive location of Mds. This is not a problem since Mds is in shared memory, which does not require coalescing to achieve high speed data access.

5.3. Dynamic Partitioning of SM Resources

The execution resources in a Streaming Multiprocessor, or SM, include registers, thread block slots, and thread slots. These resources are dynamically partitioned and assigned to threads to support their execution. In Chapter 3, we have seen that each SM has 768 thread slots, each of which can accommodate one thread. These thread slots are partitioned and assigned to thread blocks during runtime. If each thread block consists of 256 threads, the 768 threads slots are partitioned and assigned to three blocks. In this case, each SM can accommodate up to three thread blocks due to limitations on thread slots. If each thread block contains 128 threads, the 768 thread slots are partitioned and assigned to 6 thread blocks. The ability to dynamically partition the thread slots among thread blocks makes the streaming multiprocessors versatile. They can either execute many thread blocks each of which consists of few threads or execute few thread blocks each of which consists of many threads. This is in contrast to a fixed partitioning method where each block receives a fixed amount of resource regardless of their real needs. Such fixed partitioning results in wasted
thread slots when a block has few threads and fails to support blocks that require more thread slots than the fixed partition allows.

Dynamic partitioning of resources can result in subtle interactions between resource limitations, which in turn cause underutilization of resources. Such interactions can occur between block slots and thread slots. If each block has 64 threads, the 768 thread slots can be partitioned and assigned to 12 blocks. However, since there are only 8 block slots in each SM, only 8 blocks will be allowed. This means that only 512 of the thread slots will be utilized. Therefore, to fully utilize both the block slots and thread slots, one needs at least 96 threads in each block.

The register file is another dynamically partitioned resource. The number of registers in each CUDA device is not specified in the language and varies across implementations. In G80, there is an 8192-entry register file in each SM. These registers are used to hold frequently used programmer and compiler-generated variables to reduce their access latency and to conserve memory bandwidth. As we mentioned in Chapter 4, the automatic variables declared in a CUDA kernel are placed into registers. Some kernels may use lots of automatic variables and others may use few of them. Thus, one should expect that some kernels require many registers and some require fewer. By dynamically partitioning the registers among blocks, the SM can accommodate more blocks if they require few registers and fewer blocks if they require more registers. One does, however, need to be aware of potential interactions between register limitations and other resource limitations.

![Figure 5.12 Interaction of resource limitations](image)

In the matrix multiplication example, assume that the kernel code uses 10 registers per thread. If we have 16X16 thread blocks, how many threads can run on each SM? We can answer this question by first calculate the number of registers needed for each block, which is 10*16*16 = 2560. The number of registers required by three blocks is 7680, which is under the 8,120 limit. Adding another block would require 10240 registers, which exceeds
the limit. Therefore, as shown in Figure 5.12(a), the register limitation allows 3 blocks that altogether have 768 threads to run on each SM, which also fits within the limit of 8 block slots and 768 thread slots.

Now assume that the programmer declares another automatic variable in the kernel and bump the number of registers used by each thread to 11. Assuming the same 16X16 blocks, each block now requires 11*16*16 = 2,816 registers. The number of registers required by three blocks is now 8,448, which exceeds the register limitation. As shown in Figure 5.12(b), the SM deals with this situation by reducing the number of blocks by one, thus reducing the number of registered required to 5,632. This, however, reduces the number of threads running on an SM from 768 to 512. That is, by using one extra automatic variable, the program saw a 1/3 reduction in the thread-level parallelism in G80 execution! This is sometimes referred to as a “performance cliff” where a slight increase in resource usage can result in dramatic reduction in parallelism and performance achieved.

In some cases, adding an automatic variable may allow the programmer to improve the execution speed of individual threads by initiating time consuming memory accesses early, as we will explain in more detail later in this chapter. The improvement within each thread may be sufficient to overcome the loss of thread-level parallelism. For example, assume that in the original kernel, there are four independent instructions between a global memory load and its use. In G80, each instruction takes 4 clock cycles to process. So the 4 independent instructions give a 16 cycle slack for the memory access. With a 200-cycle global memory latency, we need to have at least 200/(4*4) = 14 warps available for zero-overhead scheduling to keep the execution units fully utilized.

Assume that the additional register allows the programmer or the compiler to use a program transformation technique to increase the number of independent instructions from 4 to 8. These independent instructions give 1 32 cycle slack for the memory access. With the same 200-cycle global memory latency, we now only need 200/(4*8) = 7 warps available for zero-overhead scheduling to keep the execution units fully utilized. That is, even though we just reduced the number of blocks from 3 to 2, and thus the number of warps from 24 to 16, we may have enough warps to fully utilize the execution units in each SM. Thus, the performance may actually increase! A programmer typically needs to experiment with each alternative and choose the best performing code. This can be a labor intensive, tedious process. Ryoo et al have proposed a methodology for automating the experimentation process to reduce the programming efforts required to reach an optimal arrangement for each variety of CUDA hardware [RyooCGO2008].

5.4. Data Prefetching

One of the most important resource limitations for parallel computing in general is that global memory has limited bandwidth in serving data accesses and these accesses take a long time to complete. The tiling techniques for using shared memory address the problem
of limited global memory bandwidth. The CUDA threading model tolerates long memory access latency by allowing some warps to make progress while others wait for their memory access results. While this is a very powerful mechanism, it may not be sufficient in some cases where all threads are waiting for their memory access results. Such a situation can arise if all threads have very small number of independent instructions between memory access instructions and the consumer of the data accessed.

```c
Loop {
    Load current tile to shared memory
    syncthreads()
    Compute current tile
    syncthreads()
}
```

(a) Without prefetching

```c
Loop {
    Load next tile from global memory
    Loop {
        Deposit current tile to shared memory
        syncthreads()
        Load next tile from global memory
        Compute current tile
        syncthreads()
    }
    Compute current tile
    syncthreads()
}
```

(b) With prefetching

Figure 5.13 Data Prefetching.

A useful, complementary solution to the problem is to prefetch the next data elements while consuming the current data elements, which increases the number of independent instructions between the memory accesses and the consumers of the data accessed. Prefetch techniques are often combined with tiling to simultaneously address the problems of limited bandwidth and long latency. We show such a combined approach in Figure 5.13.

The algorithm in Figure 5.13(a) corresponds to the tiled matrix multiplication kernel in Figure 5.10. Lines 9 and 10 in Figure 5.10 correspond to “load current tile to shared memory” in Figure 5.13(a). This is the part that loads data from global memory into shared memory. The dot-product code (lines 12 and 13) in Figure 5.10 correspond to “compute current tile” in Figure 5.13(a). This is the part that consumes the loaded data. Note that there is no substantial activity between the two parts. That is, there are few independent instructions between these two parts.

Figure 5.13(b) shows a prefetch version of matrix multiplication kernel. This technique allocates twice the amount of shared memory for each tile, one holds the tile currently being processed and one holds the tile to be processed next. Before we enter the while loop, we load the first tile into the registers. Once we enter the loop, we move the loaded data into shared memory. Since this is a consumer of the loaded data, the threads will likely need to be put to sleep, waiting for its loaded data while other threads make progress. When the first tile of data arrive, threads in the block pass the barrier synchronization and deposit the tile data from their registers to the shared memory. When
all threads of a block complete depositing their data, they pass the barrier synchronization point and begin to load the next tile into their registers. The key is that the next tile data loaded is not immediately consumed. Rather, the current block is processed from the shared memory by the dot-product loop of lines 12 and 13 in Figure 5.10.

When the loop iterates, the “next tile” in the current iteration becomes “current tile” of the next iteration. Thus, the deposit of the “current tile” into the shared memory corresponds to the “next tile” loaded in the previous iteration. The execution of the dot-product loop provides many independent instructions between the two parts. This reduces the amount of time the threads will need to wait for their global memory access data.

We would like to encourage the reader to revise the kernel in Figure 5.10 to use prefetch. A cost of the data prefetch is that it uses two additional automatic variables (registers). As we discussed in Section 5.3, using additional registers can reduce the number of blocks that can run on an SM. However, this technique can still win if it significantly reduces the amount of time each thread waits for its global memory load data.

5.5. Instruction Mix

In current generation CUDA GPUs, each processor core has limited instruction processing bandwidth. Every instruction consumes instruction processing bandwidth, whether it is a floating point calculation instruction, a load instruction, or a branch instruction. Figure 5.14(a) shows the dot-product loop of the matrix multiplication kernel. The loop incurs extra instructions to update loop counter k and performs conditional branch at the end of each iteration. Furthermore, the use of loop counter k to index the Ms and Ns matrices incurs address arithmetic instructions. These instructions compete against floating point calculation instructions for the limited instruction processing bandwidth.

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

(a) loop incurs overhead instructions

```
Pvalue += Ms[ty][k] * Ns[k][tx] + ...
    Ms[ty][k+15] * Ns[k+15][tx];
```

(b) loop unrolling eliminates overhead

Figure 5.14 Loop unrolling improves instruction mix

For example, the kernel loop in 5.14(a) executes 2 floating point arithmetic instructions, one loop branch instruction, two address arithmetic instructions, and one loop counter increment instruction. That is, only 1/3 of the instructions executed are floating-point
calculation instructions. With limited instruction processing bandwidth, this instruction mixture limits the achievable performance to no more than 1/3 of the peak bandwidth.

A common way to improve the instruction mix is to unroll the loop, as shown in Figure 5.14(b). Given a tile size, one can simply unroll all the iterations and simply express the dot product computation as one long multiply-add expression. This eliminates the branch instruction and the loop counter update. Furthermore, since the indices are constants rather than K, the compiler can use the addressing mode offsets of the load instructions to eliminate address arithmetic instructions. As a result, the long expression can execute at close to peak performance!

Ideally, loop unrolling should be automatically done by the compiler. This is one of the areas where compiler technology will likely be improved rapidly in the near future. Until the tools mature, many programmers will still unroll loops in their source code to achieve high performance.

5.6. Thread Granularity

An important algorithmic decision in performance tuning is the granularity of threads. It is often advantageous to put more work into each thread and use fewer threads. Such advantage arises when some redundant work exists between threads. Figure 5.14 illustrates such an opportunity in matrix multiplication. The tiled algorithm in Figure 4.8 uses one thread to compute one element of the output Pd matrix. This requires a dot product between on row of Md and one column of Nd.

The opportunity of thread granularity adjustment comes from the fact that multiple threads redundantly load each Md row. As shown in Figure 5.14, the calculation of two Pd...
elements in adjacent tiles uses the same Md row. With the original tiled algorithm, the same Md row is redundantly loaded by the two thread blocks assigned to generate these two Pd tiles. One can eliminate this redundancy by merging the two thread blocks into one. Each thread in the new thread block now calculates two Pd elements. This is done by revising the kernel so that each two dot-products are computed by the kernel. Both dot products use the same Mds row but different Nds columns. This reduces the global memory access by \( \frac{1}{4} \). It also increases the number of independent instructions in the case of a prefetch algorithm in Figure 5.13 since there are two dot-products calculated between the loading of the tiles into registers and depositing these tiles into shared memories.

The potential downside is that the new kernel now uses more registers and shared memory. Thus the number of blocks that can be running on each SM may decrease. It also reduces the total number of thread blocks by half, which may result in insufficient amount of parallelism for matrices of smaller dimensions. For G80/G280, we found that combining four adjacent horizontal blocks to compute for adjacent horizontal tiles gives the best performance for a 2048x2048 matrix multiplication.

5.7. Experimental Performance Tuning

The combined effects of various performance enhancement techniques on the matrix multiplication kernel are shown in Figure 5.14. The dimensions covered are tiling size, loop unrolling, data prefetching, and thread granularity. We can make at least four observations.

![Figure 5.14: Effects of Performance Improvement Techniques](image)

Figure 5.15 Effects of Performance Improvement Techniques

First, the tile size plays a major role in the performance. Until the tile size reaches 16X16, neither loop unrolling nor data prefetch helps. This is reflected by the fact that all eight bars in granularity bracket are of the same height. For small tile sizes such as 8x8, the saturated global memory bandwidth so severely limits the execution performance that transformations such as loop unrolling and data prefetching simply do not matter. On the
other hand, since rectangular tiling can reduce global memory accesses, one would expect that it should improve performance. That is, 1x2 rectangular tiling reduces the global memory access by ¼ and resulted and 1X4 tiling by 3/8. Note that 1X8 would have reduced the global traffic by only 7/16, a diminishing return that makes it much less attractive than using a larger tile size. The reductions in global memory accesses indeed help improve the performance shown in Figure 5.15.

Second, once the tile size becomes sufficiently large, 16X16 in this case, to alleviate the saturation of global memory bandwidth, loop unrolling and data prefetching become much more important. In most cases, complete unrolling the loop can result in more than 20% performance improvement.

The third observation is that while data prefetching is very beneficial for 1x1 tiling, it does not help much for 1x2 rectangular tiling. In fact, for 1x4 rectangular tiling, the register usage by one block of data prefetching kernel exceeded the total number of registers in the SM. This makes the code not executable in G80! This is a good illustration that as one applies multiple techniques to a kernel, these techniques will likely interact by reducing the resources available to other techniques.

Finally, the appropriate combinations of performance tuning techniques can make a tremendous difference in the performance achieved by the matrix multiplication kernel. In Figure 5.15, the speed of the kernel executing on G80 increased from 18 GFOPS to 120 GLOPS! However, the programming efforts required to search through these combinations is currently quite large. Much work is being done in both academia and industry to reduce the amount of programming efforts needed to achieve these performance improvements with automation tools.
Chapter 6
Floating Point Considerations

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

6.1. Floating Point Format

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

A floating point number system starts with the representation of a numerical value as bit patterns. In the IEEE Floating Point Standard, a numerical value is represented in three groups of bits: sign (S), exponent (E), and mantissa (M). Each (S, E, M) pattern uniquely identifies a numeric value according to the following formula:

$$\text{value} = (-1)^S \times M \times \{2^E\}, \text{ where } 1.0 \leq M < 2.0$$  \hspace{1cm} (1)

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

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

**Normalized representation of M**

Formula (1) requires that \(1.0B \leq M < 10.0B\), which makes the mantissa value for each floating point number unique. For example, the only one mantissa value allowed for \(0.5D\) is \(M = 1.0\):

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

Another potential candidate would be \(0.1_B \times 2^0\), but the value of mantissa would be too small according to the rule. Similarly, \(10.0_B \times 2^{-2}\) is not legal because the value of the mantissa is too large. In general, with the restriction that \(1.0B \leq M < 10.0B\), every floating point number has exactly one legal mantissa value. The numbers that satisfy this restriction will be referred to as normalized numbers. Because all mantissa values that satisfy the restriction are of the form \(1.XX\), we can omit the “1.” part from the representation. Therefore, the mantissa value of 0.5 in a 2-bit mantissa representation is 00, which is derived by omitting “1.” from 1.00. This makes a 2-bit mantissa effectively a 3-bit mantissa. In general, with IEEE format, an n-bit mantissa is effectively an (n+1)-bit mantissa.

**Excess encoding of E**

If n bits are used to represent the exponent E, the value \(2^{n-1}-1\) is added to the two's complement representation for the exponent to form its excess representation. A two’s complement representation is a system where the negative value of a number can be derived by first complementing every bit of the value and add one to the result. In our 3-bit exponent representation, there are three bits in the exponent. Therefore, the value \(2^{3-1}-1 = 011\) will be added to the 2’s complement representation of the exponent value. The following table shows the 2’s complement representation and the excess presentation of each decimal exponent value. In our example, the exponent for \(0.5D\) is -1. The two’s
complement representation of -1 can be derived by first complementing 001, the representation of 1, into 110 and then adding 001 to get 111. The excess presentation adds another 011 to the 2’s complement representation, as shown in the table, which results in 010.

<table>
<thead>
<tr>
<th>2’s complement</th>
<th>Decimal value</th>
<th>Excess-representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>011</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>110</td>
</tr>
<tr>
<td>100</td>
<td>(reserved pattern)</td>
<td>111</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
<td>000</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
<td>001</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
<td>010</td>
</tr>
</tbody>
</table>

Figure 1 Excess-3 encoding, sorted by 2’s complement ordering

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

<table>
<thead>
<tr>
<th>2’s complement</th>
<th>Decimal value</th>
<th>Excess-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>(reserved pattern)</td>
<td>111</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
<td>000</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
<td>001</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
<td>010</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>011</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>101</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>110</td>
</tr>
</tbody>
</table>

Figure 2 Excess-3 encoding, sorted by excess-3 ordering

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

0.5_D = 0 010 00, where S = 0, E = 010, and M = (±)00
That is, the 6-bit representation for 0.5_10 is 001000.

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

\[ (-1)^S \times 1.M \times 2^{(E - (2^{(n-1)})+1)} \]

### 6.2. Representable Numbers

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

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

*Figure 3 Representable numbers of a 3-bit unsigned format*

Neither -1 nor 9 can be represented in the format given above. We can draw a number line to identify all the representable numbers, as shown in Figure 4 where all representable numbers of the 3-bit unsigned integer format are marked with stars.

*Figure 4 representable numbers of a 3-bit unsigned integer format*

The representable numbers of a floating point format can be visualize in a similar manner. In Table 1, we show all the representable numbers of what we have so far and two variations. We use a 5-bit format to keep the size of the table manageable. The format consists of 1-bit S, 2-bit E (excess-1 coded), and 2-bit M (with “1.” part omitted). The no-zero column gives the representable numbers of the format we discussed thus far. Note that with this format, 0 is not one of the representable numbers.
<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>S=0</th>
<th>S=1</th>
<th>Abrupt underflow</th>
<th>S=0</th>
<th>S=1</th>
<th>Denorm</th>
<th>S=0</th>
<th>S=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>2^{-1}</td>
<td>-2^{-1}</td>
<td>0</td>
<td>0</td>
<td>1*2^{-2}</td>
<td>-1*2^{-2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>2^{-1}+1*2^{-3}</td>
<td>-(2^{-1}+1*2^{-3})</td>
<td>0</td>
<td>0</td>
<td>2*2^{-2}</td>
<td>-2*2^{-2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>2^{-1}+2*2^{-3}</td>
<td>-(2^{-1}+2*2^{-3})</td>
<td>0</td>
<td>0</td>
<td>3*2^{-2}</td>
<td>-3*2^{-2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>2^{-1}+3*2^{-3}</td>
<td>-(2^{-1}+3*2^{-3})</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>2^n</td>
<td>-(2^n)</td>
<td>2^n</td>
<td>-2^n</td>
<td>-(2^n)</td>
<td>-2^n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>2^n+1*2^{-2}</td>
<td>-(2^n+1*2^{-2})</td>
<td>2^n+1*2^{-2}</td>
<td>-(2^n+1*2^{-2})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>2^n+2*2^{-2}</td>
<td>-(2^n+2*2^{-2})</td>
<td>2^n+2*2^{-2}</td>
<td>-(2^n+2*2^{-2})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2^n+3*2^{-2}</td>
<td>-(2^n+3*2^{-2})</td>
<td>2^n+3*2^{-2}</td>
<td>-(2^n+3*2^{-2})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>2^n+4*2^{-2}</td>
<td>-(2^n+4*2^{-2})</td>
<td>2^n+4*2^{-2}</td>
<td>-(2^n+4*2^{-2})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 Representable numbers of no-zero, abrupt underflow, and denorm formats

A look at how these representable numbers populate the number line, as shown in Figure 5, provides further insights about these representable numbers. In Figure 5, we show only the positive representable numbers. The negative numbers are symmetric to their positive counterparts on the other side of 0.

![Figure 5. Representable numbers of the no-zero representation](image)

Three things stand out. First, 0 is not representable in this format. Because 0 is one of the most important numbers, not being able to represent 0 in a number representation system is a serious deficiency. Second, the representable numbers become closer to each other towards the neighborhood of 0. This is a desirable behavior because as the absolute value of these numbers become smaller, it is more important to represent them more accurately. Having the representable numbers closer to each other makes it possible to represent numbers more accurately. Unfortunately this trend does not hold for the very vicinity of 0, which leads to the third point: there is a gap of representable numbers in the vicinity of 0. This is because the range of normalized mantissa precludes 0. This is another serious deficiency. The representation introduces significantly larger errors when representing numbers between 0 and 0.5 compared to the errors for the larger numbers between 0.5 and 1.0.
One method that has been used to accommodate 0 into a normalized number system is the *abrupt underflow* convention, which is illustrated by the second column of Table 1. Whenever \( E = 0 \), the number is interpreted as 0. In our 5-bit format, this method takes away eight representable numbers (four positive and four negative) in the vicinity of 0 and makes them all 0. Although this method makes 0 representable, it does create an even larger gap between representable numbers in 0's vicinity, as shown in Figure 6. It is obvious, when compared with Figure 5, the gap of representable numbers has been enlarged significantly with the vicinity of 0. This is very problematic since many numeric algorithms rely on the fact that the accuracy of number representation is higher for the very small numbers near zero. These algorithms generate small numbers and eventually use them as denominators. The errors for these small numbers can be greatly magnified in the division process.

![Figure 6 Representable numbers of the abrupt underflow format](image)

The actual method adopted by the IEEE standard is called denormalization. The method relaxes the normalization requirement for numbers very close to 0. That is, whenever \( E = 0 \), the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. In general, if the \( n \)-bit exponent is 0, the value is

\[
0.M \times 2^{-2^{(n-1)} + 2}
\]

For example, in Table 1, the denormalized representation 00001 has exponent value 00 and mantissa value 01. Using the denormalized formula, the value it represents is \( 0.01 \times 2^0 = 2^{-2} \). Figure 7 shows the representable numbers for the denormalization format. The representation now has a uniformly spaced representable numbers in the close vicinity of 0. This eliminates the undesirable gap in the previous two methods.

![Figure 7. Representable numbers of a denormalization format.](image)

We are now ready to discuss the concept of precision. The precision of a floating point representation is measured by the maximal error that we can introduce to a floating point number by represententing that number as one of the representable numbers. The smaller the error is, the higher the precision. In Figure 7, the error introduced by a number representation is always smaller than 0.25, which occurs if we choose the largest representative number that is smaller than the number we would like to represent. Obviously, the precision is a floating point representation can be improved by adding more
bits to mantissa. Adding one bit to the representation in Figure 7 would improve the precision by reducing the maximal error by half. Thus, we say that a number system has higher precision when it uses more bits for mantissa.

6.3. Special Bit Patterns and Precision

The actual IEEE format has one more special bit pattern. When all exponent bits are 1s, the number represented is an infinity value if the mantissa is 0 or a Not a Number (NaN) if the mantissa is not 0. All special bit patterns of the IEEE floating point format are shown in the following table.

<table>
<thead>
<tr>
<th>exponent</th>
<th>mantissa</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>11…1</td>
<td>≠ 0</td>
<td>NaN</td>
</tr>
<tr>
<td>11…1</td>
<td>=0</td>
<td>(-1)(^e) * (\infty)</td>
</tr>
<tr>
<td>00…0</td>
<td>≠ 0</td>
<td>denormalized</td>
</tr>
<tr>
<td>00…0</td>
<td>=0</td>
<td>0</td>
</tr>
</tbody>
</table>

All other numbers are normalized floating-point numbers. Single precision numbers have 1-bit S, 8-bit E, and 23-bit M. Double precision numbers have 1-bit S, 11-bit E, and 52-bit M. Since a double precision number has 29 more bits for mantissa, the largest error for representing a number is reduced to \(1/2^{29}\) of that of the single precision format!

All representable numbers fall between \(-\infty\) (negative infinity) and \(+\infty\) (positive infinite). An \(\infty\) can be created by overflow, e.g., divided by zero. Any representable number divided by \(+\infty\) or \(-\infty\) results in 0.

NaN (Not a Number) is generated by operations whose input values do not make sense, for example, 0/0, \(0^{\infty}\), \(\infty/\infty\), \(\infty-\infty\). They are also used to for data that have not be properly initialized in a program. There are two types on NaN’s in the IEEE standard: signaling and quiet.

Signaling NaN causes an exception when used as input to arithmetic operations. For example the operation 1.0 + signaling NaN raises an exception. Signaling NaN’s are used in situations where the programmer would like to make sure that the program execution be interrupted whenever any NaN values are used in floating point computations. These situations usually mean that there is something wrong with the execution of the program. In mission critical applications, the execution cannot continue until the validity of the execution can be verified with a separate means. For example, software engineers often mark all the uninitialized data as signaling NaN. This practice ensures the detection of using uninitialized data during program execution.
Quiet NaN generates a quiet NaN when used as input to arithmetic operations. For example, the operation \((1.0 + \text{quiet NaN})\) generates a quiet NaN. Quiet NaN’s are typically used in applications where the user can review the output and decide if the application should be re-run with a different input for more valid results. When the results are printed, Quite NaN’s are printed as “NaN” so that the user can spot them in the output file easily.

6.4. Arithmetic Accuracy and Rounding of Mantissa

Now that we have a good understanding of the IEEE floating point format, we now move to the concept of arithmetic accuracy. The accuracy of a floating point arithmetic operation is measured by the maximal error introduced by the operation. The smaller the error is, the higher the accuracy. The most common source of error in floating point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding. Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly. The cause of rounding is typically pre-shifting in floating point arithmetic. When two input operands to a floating-point addition or subtraction have different exponents, the mantissa of the one with the smaller exponent is typically right-shifted until the exponents are equal. As a result, the final result can have more bits than the format can accommodate.

This can be illustrated with a simple example based on the 5-bit representation in Table 1. Assume that we need to add \(1.00*2^{-2}\) \((0, 00, 01)\) to \(1.00*2^1\) \((0, 10, 00)\), that is, we need to perform \(1.00*2^1 + 1.00*2^{-2}\). The ideal result would be \(1.001 * 2^1\). However, we can easily see that this ideal result is not a representable number in a 5-bit representation. Thus, the best one can do is to generate the closest representable number, which is \(1.01 * 2^1\). By doing so, we introduce an error, \(0.001 * 2^1\), which is half the place value of the least significant place. We refer to this as 0.5 ULP (Units in the Last Place). If the hardware is designed to perform arithmetic and rounding operations perfectly, the most error that one should introduce should be no more than 0.5 ULP. This is the accuracy achieved by the addition and subtraction operations in G80/280.

In practice, some of the arithmetic hardware units, such as division and transcendental functions, are typically implemented with iterative approximation algorithms. If the hardware does not perform sufficient number of iterations, the result may have an error larger than 0.5 ULP. For example, the division in G80/280 can introduce an error that is twice the place value of the least place of the mantissa.

6.4. Algorithm Considerations

Numerical algorithms often need to sum up a large number of values. For example, the dot product in matrix multiplication needs to sum up pair-wise products of input matrix elements. Ideally, the order of summing these values should not affect the final total since addition is an associative operation. However, with finite precision, the order of summing these values can affect the accuracy of the final result. For example, if we need to add four numbers in our 5-bit representation: 
\[1.00*2^0 + 1.00*2^0 + 1.00*2^2 + 1.00*2^2.\]
If we add up the numbers in strict sequential order, we have the following sequence of operations:

\[
1.00\times2^0 + 1.00\times2^0 + 1.00\times2^{-2} + 1.00\times2^{-2} = 1.00\times2^1 + 1.00\times2^{-2} + 1.00\times2^{-2} \\
= 1.00\times2^1 + 1.00\times2^{-2} = 1.00\times2^1
\]

Note that in the second step and third step, the smaller operand simply disappears because they are too small compared to the larger operand. The pre-shifting of mantissa produces a zero value for the smaller operand.

Now, let’s consider a parallel algorithm where the first two values are added and the second two operands are added in parallel. The algorithm than add up the pair-wise sum:

\[
(1.00\times2^0 + 1.00\times2^0) + (1.00\times2^{-2} + 1.00\times2^{-2}) = 1.00\times2^1 + 1.00\times2^{-1} \\
= 1.01\times2^1
\]

Note that the results are different from the sequential result! This is because the sum of the third and fourth values is large enough that it remains non-zero after pre-shifting. This discrepancy between sequential algorithms and parallel algorithms often surprises application developers who are not familiar with floating point precision and accuracy considerations. Although we showed a scenario where a parallel algorithm produced a more accurate result than a sequential algorithm, the reader should be able to come up with a slightly different scenario where the parallel algorithm produces a less accurate result than a sequential algorithm. Experienced application developers either make sure that the variation in the final result can be tolerated or to ensure that the data is grouped in a way that the parallel algorithm results in the most accurate results.
Chapter 7
Application Case Study – Quantitative MRI Reconstruction

Application case studies teach computational thinking and programming in a concrete manner. They also help demonstrate how the individual techniques fit into a top-to-bottom development process. Most importantly, they help us to visualize the practical use of these techniques in solving problems. In this chapter, we start with the background and problem formulation of a relatively simple application. We show that parallel execution does not only speedup the existing approaches, but also allows applications experts to pursue approaches that are known to provide benefit but were previously ignored due to their excessive computational requirements. We then use an example algorithm and its implementation source code from such an approach to illustrate how a developer can systematically determine the kernel parallelism structure, assign variables into CUDA memories, steer around limitations of the hardware, validate results, and assess the impact of performance improvements.

7.1. Application Background

Magnetic resonance imaging (MRI) is commonly used by the medical community to safely and non-invasively probe the structure and function of biological tissues in all regions of the body. Images that are generated using MRI have made profound impact in both clinical and research settings. MRI consists of two phases, acquisition (scan) and reconstruction. During the acquisition phase, the scanner samples data in the k-space domain (i.e. the spatial-frequency domain or Fourier transform domain) along a predefined trajectory. These samples are then transformed into the desired image during the reconstruction phase.

The application of MRI is often limited by high noise levels, significant imaging artifacts, and/or long data acquisition times. In clinical settings, short scan times not only increase scanner throughput but also reduce patient discomfort, which tends to mitigate motion-related artifacts. High image resolution and fidelity are important because they enable earlier detection of pathology, leading to improved prognoses for patients. However, the goals of short scan time, high resolution, and high signal-to-noise ratio (SNR) often
conflict; improvements in one metric tend to come at the expense of one or both of the others. One needs new, disruptive technological breakthroughs to be able to simultaneously improve on all of three dimensions. This study presents a case where massively parallel computing provides such a disruptive breakthrough.

The reader is referred to MRI textbooks [Liang] for the physics principles behind MRI. For this case study, we will focus on the computational complexity in the reconstruction phase as affected by the k-space sampling trajectory. The k-space sampling trajectory used by the MRI scanner can significantly affect the quality of the reconstructed image, the time complexity of the reconstruction algorithm, and the time required for the scanner to acquire the samples. In general, the MRI reconstruction problem is defined by equation (1),

$$\hat{m}(r) = \sum_j W(k_j)s(k_j)e^{i2\pi k_j \cdot r} \tag{1}$$

Where $m(r)$ is the reconstructed image, $s(k)$ is the measured k-space data, and $W(k)$ is the weighting function that accounts for non-uniform sampling. That is, $W(k)$ decreases the influence of data from k-space regions where a higher density of samples points are taken.

If data are acquired by measuring at uniformly spaced Cartesian grid points in the k-space, then this weighting function is a constant and can thus be factored out of the summation in (1). As a result, the reconstruction of $m(r)$ becomes an inverse Fast Fourier Transform (FFT) on $s(k)$, an extremely efficient computation method. A collection of data measured at such uniformed spaced Cartesian grid points is referred as a **Cartesian scan trajectory**. Fig. 7.1(a) depicts a Cartesian scan trajectory. Because Cartesian scan trajectory allows image reconstruction to be performed quickly and efficiently as an inverse FFT on the acquired data, it is the approach used in most clinical MRI scanners today.

![Figure 7.1](image)

**Figure 7.1.** Scanner k-space trajectories and their associated reconstruction strategies: (a) Cartesian trajectory with FFT reconstruction, (b) Spiral (or non-Cartesian trajectory in general) followed by gridding to enable FFT reconstruction, (c) spiral (non-Cartesian) trajectory with linear solver based reconstruction.
Although the inverse FFT reconstruction of Cartesian scan data is computationally efficient, Cartesian scan trajectories often require longer scanner time than non-Cartesian trajectories. This stems from the fact that the number of trajectories required to satisfy Nyquist criterion in k-space sampling is different for different trajectory shapes. Non-Cartesian scan trajectories like spirals (shown in Figure 7.1(c)), radial lines (projection imaging) and rosettes cover the k-space much more efficiently. For instance, one needs fewer spiral trajectories than Cartesian trajectories to cover the same area in the k-space, thus reducing the scan time for spiral trajectories. The reduction of trajectories taken translates into less scan time, and thus improved patient comfort and reduced motion related artifacts. Furthermore, when time-dependent phenomena like bolus-spreading or movement of the heart need to be scanned, one must reduce the scan time for generating each video frame to achieve a frame rate needed to produce usable videos.

Image reconstruction from non-Cartesian trajectory data presents both challenges and opportunities. The main challenge arises from the fact that the W(k) function in (1) is not a constant function for non-Cartesian trajectory data and thus can no longer be factored out of the summation. Therefore, one can no longer perform reconstruction by directly applying an inverse FFT to the k-space sample. In a commonly used approach called gridding, the samples are first interpolated onto a uniform Cartesian grid and then reconstructed using the FFT (see Fig. 7.1(b)). Earlier gridding methods used simple bilinear interpolation to map the data onto a Cartesian grid. However, simple bilinear interpolation suffered from extra artifacts and low accuracy. Thus, a popular method for the reconstruction of non-Cartesian k-space data has been to (re)grid the data onto a Cartesian grid via a convolution approach. Each of the data points, which lie along some trajectory in k-space, is convolved with a gridding kernel, and the result sampled and accumulated on a Cartesian grid. Convolution is quite computationally intensive. Studies have shown that for standard gridding parameters, the reconstruction is approximately six times longer than that of Cartesian sampled data and can make the reconstruction on a CPU too long for clinical use. Accelerating gridding computation on many-core processors enables the application of the current FFT approach to non-Cartesian trajectory data. Since we will be examining a convolution-style computation in chapter 8, we will not cover it here.

An interesting opportunity arises when we rethink the entire reconstruction process. The use of non-Cartesian trajectory allows the scanner to take more samples at a given time budget and can potentially improve SNR. However, inverse FFT satisfies no optimality or bound criterion, does not model imaging physics. In applications where the pixels in the reconstructed image are used to provide general diagnosis impression and their numerical values are not used for critical diagnosis decisions, gridding followed by inverse FFT offers a cost-effective solution. However, it does not take advantage of the potential offered by increased amount of sample data to enable more applications where the reconstructed image pixel values are used for critical diagnosis decisions.
By contrast, iterative, statistically optimal image reconstruction methods can more accurately model imaging physics and bound the noise error in each image pixel value. This allows the reconstructed image pixel values to be used for measuring subtle phenomenon such as tissue chemical anomalies before they become anatomical pathology. However, such iterative reconstructions have been impractical for large-scale 3D problems due their excessive computational requirements compared inverse FFT. Recently, these reconstructions have become viable in clinical settings when accelerated on GPUs. In particular, we will show that an iterative reconstruction algorithm that explicitly models imaging physics used to take hours using a high-end sequential CPU but takes only minutes using G80 for a image of moderate resolution, a delay acceptable in clinical settings.

![Figure 7.2. The use of non-cartesian k-space sample trajectory and accurate linear-solver based reconstruction enables new MRI modalities with exciting medical applications. The faster scan time allows acquisition of more samples required to collect in-vivo concentration data on less abundant substance such as sodium in human tissues. The variation or shifting of sodium concentration gives early signs of disease development or tissue death. An example of sodium map of a human brain shown in this Figure can be used to give early indication of brain tumor tissue responsiveness to chemo-therapy protocols, enabling individualized medicine.](image_url)

The availability of fast, robust, and statistically optimal image reconstruction methods for non-Cartesian k-space trajectories enables new MRI modalities. The typical medical use of MRI today studies the human anatomical structures by mapping out the concentration levels of water in human tissues. The progress a disease development or treatment is measured by the anatomical changes such as enlargement or shrinkage of tumors. One can, however, measure much earlier progress that precedes the anatomical changes by measuring the concentration changes of sodium, a heavily regulated substance in normal human cells. Such a measurement is shown in Figure 7.2. Because sodium and other interesting chemical substance for medical applications are much less abundant than water molecules in human tissues, a reliable measurement of sodium concentration needs a drastic increased k-space coverage in order to achieve an acceptable level of SNR in the resulting image. The increased coverage requirement motivates the use of multiple scanners as well as non-Cartesian data trajectories in order to keep the scanning time reasonable for patients. Iterative, statistically optimal reconstruction methods are needed to deploy physics models needed to correctly stitch together the data from multiple scanners and to bound the noise error for each pixel point so that the pixel values are accurate measurements of the sodium level for early assessment of disease progress.
7.2. Iterative Reconstruction

Haldar and Liang proposed a linear solver based iterative reconstruction algorithm for non-Cartesian scan data, as shown in Fig. 7.1(c). The algorithm allows for explicit modeling and compensation for the physics of the scanner data acquisition process, and can thus reduce the artifacts in the reconstructed image. It is, however, computational intensive. The reconstruction time on high-end sequential CPUs has been hours for moderate-resolution images and thus impractical in clinical use. The objective here is to use the massive parallelism in GPUs to reduce the reconstruction time to a matter of seconds so that one can begin to deploy the new MRI modalities such as sodium imaging in clinical settings.

Figure 7.3 shows a quasi-Bayesian estimation problem formulation of the linear-solver based reconstruction approach, where $\rho$ is a vector containing voxel values for the reconstructed image, $F$ is a matrix that models the physics of imaging process, $d$ is a vector of data samples from the scanner, and $W$ is a matrix that can incorporate prior information such as anatomical constraints. In clinical settings, the anatomical constraints represented in $W$ are derived from one or more high resolution, high-SNR water molecule scans of the patient. These water molecule scans reveal features such as the location of anatomical structures. The matrix $W$ is derived from these reference images. The problem is to solve for $\rho$ given all the other matrices and vectors.

\[
(F^H F + \lambda W^H W)\rho = F^H d
\]

On the surface, the computational solution to the problem formulation in Figure 7.3 should be very straightforward. It involves matrix-matrix multiplications and addition ($F^H * F + \lambda W^H W$), matrix-vector multiplication ($F^H * d$), matrix inversion $(F^H F + \lambda W^H W)^{-1}$, and finally matrix-matrix multiplication $((F^H F + \lambda W^H W)^{-1} * F^H d)$. However, the sizes of the matrices make the solution extremely time consuming. $F^H$ and $F$ are 3D matrices whose dimensions are determined by the resolution of the reconstructed image $\rho$. Even in a
modest resolution 128³-voxel reconstruction, computation of \( F^H F + \lambda W^H W \) can take days to complete on a high-end CPU. For a more desirable resolution of 512³-voxel reconstruction, the matrix-matrix multiply will take \( 4^4 = 64 \) times the time. Fortunately, the computation of \( F^H F + \lambda W^H W \) does not have to be repeated for each data acquisition. It needs to be done only once for a scanner set up for a patient. However, due to the extreme long latency of the computation, it is still desirable to speed up the \( F^H F + \lambda W^H W \) computation to minimize the inconvenience of scanner setup for each patient. The acceleration of the \( F^H F + \lambda W^H W \) computation can be found in Stone, et al [Stone 2008].

The matrix-vector multiply to calculate \( F^H d \) takes about one order of magnitude less time than \( F^H F + \lambda W^H W \) but can still take about three hours for a 128³-voxel reconstruction on a high-end sequential PCU. Since \( F^H d \) needs to be computed for every image acquisition, it is critical to reduce its computation to minutes. We will show the details of this process. As it turns out, the core computational structure of \( F^H F + \lambda W^H W \) is identical to that of \( F^H d \). As a result, the same methodology can be used to accelerate the computation of both.

The inversion of the \( F^H F + \lambda W^H W \) matrix can be prohibitively expensive due to the sheer size of the inverted matrix. For a 128³-voxel reconstruction, the inverted matrix contains well over four trillion complex-valued elements (the number of elements in the inverted matrix equals the square of the number of voxels in the reconstructed image). An iterative method for matrix inversion, such as the conjugate gradient (CG) algorithm, is therefore preferred. The conjugate gradient algorithm reconstructs the image by iteratively solving the equation in Figure 7.3 for \( \rho \). During each iteration, the CG algorithm updates the current image estimate \( \rho \) to improve the value of the quasi-Bayesian cost function. The computational efficiency of the CG technique is largely determined by the efficiency of matrix–vector multiplication operations involving \( F^H F + \lambda W^H W \) and \( \rho \), as these operations are required during each iteration of the CG algorithm. Fortunately, matrix \( W \) often has a sparse structure that permits efficient multiplication by \( W^H W \), and matrix \( F^H F \) has a convolution structure that enables efficient matrix multiplication via the FFT. As a result, the “find \( \rho \)” step in Figure 7.3 is much less computationally intensive than \( F^H d \), and accounting only less than 1% of the execution of the reconstruction of each image on a sequential CPU. As a result, we will leave it out of the parallelization scope and focus on \( F^H d \) in this chapter. We will however, revisit its status at the end of the chapter.
7.3. Computing $F^H d$

```c
for (m = 0; m < M; m++) {
    phiMag[m] = rPhi[m]*rPhi[m] + iPhi[m]*iPhi[m];
    for (n = 0; n < N; n++) {
        expQ = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        rQ[n] += phiMag[m]*cos(expQ);
        iQ[n] += phiMag[m]*sin(expQ);
    }
}
```

(a) Q computation

```c
for (m = 0; m < M; m++) {
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
    for (n = 0; n < N; n++) {
        expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        cArg = cos(expFhD);
        sArg = sin(expFhD);
        rFhD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

(b) $F^H d$ computation

Figure 7.4 Computation of Q and $F^H d$

Figure 7.4 shows a sequential C implementation of the computations for the core step of $F^H F + \lambda W^H W$ (Part (a)) and that for $F^H d$ (Part (b)). It should be clear from a quick glance at Figure 7.4(a) and 7.4(b) that the core step of $F^H F + \lambda W^H W$ and $F^H d$ have identical structure. Both computations start with an outer loop, which encloses an inner loop. The only differences are the particular calculation done in each loop body and the fact that the core step of $F^H F + \lambda W^H W$ involves a much larger number of sample points (M), thus a much longer execution time. Thus it suffices to discuss one of them. We will focus on $F^H d$, since this is the one that will need to be run for each image being reconstructed.

A quick glance at Figure 7.4(b) shows that the C implementation of $F^H d$ is an excellent candidate for acceleration on the GPU because it exhibits substantial data-parallelism. The algorithm first computes the real and imaginary components of Mu at each sample point in the k-space, then computes the real and imaginary components of $F^H d$ at each voxel in the image space. The value of $F^H d$ at any voxel depends on the values of all k-space sample points. However, no voxel elements of $F^H d$ depend on any other elements of $F^H d$. Therefore, all elements of $F^H d$ can be computed in parallel. Specifically, all iterations of the outer loop can be done in parallel and all iterations of the inner loop can be done in parallel. The calculations of the inner loop, however, have a dependence on the calculation done in the same iteration of the outer loop.

Despite the algorithm’s abundant inherent parallelism, potential performance bottlenecks are evident. First, in the loop that computes the elements of $F^H d$, the ratio of floating-point operations to memory accesses is at best 3:1 and at worst 1:1. The best case assumes that
the $\sin$ and $\cos$ trigonometry operations are computed using five-element Taylor series that require 13 and 12 floating-point operations, respectively. The worst case assumes that each trigonometric operation is computed as a single operation in hardware. As we have seen in Chapter 4, a floating-point to memory access ratio of 16:1 or more is needed for the kernel to be not limited by memory bandwidth. Thus, the memory accesses will clearly limit the performance of the kernel unless the ratio is drastically increased.

Second, the ratio of FP arithmetic to FP trigonometry functions is only 13:2. Thus, GPU-based implementation must tolerate or avoid stalls due to long-latency $\sin$ and $\cos$ operations. Without a good way to reduce the cost of trigonometry functions, the performance will likely be dominated by the time spent in these functions.

We are now ready to take the steps in converting $F^Hd$ from sequential C code to CUDA kernel.

```c
__global__ void cmpFHd(float* rPhi, iPhi, phiMag, 
    kx, ky, kz, x, y, z, rMu, iMu, int N) {
    int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
    for (n = 0; n < N; n++) {
        expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        cArg = cos(expFhD);   sArg = sin(expFhD);
        rFhD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

Figure 7.5 First version of the $F^Hd$ kernel. The kernel will not execute correctly due to conflicts between threads in wiring into rFhD and iFhD arrays.

**Step 1: Determine the kernel parallelism structure**

The conversion of a loop into a CUDA kernel is conceptually straightforward. Since all iterations of the outer loop of Figure 7.4(b) can be executed in parallel, we can simply convert the outer loop into a CUDA kernel by assigning its iterations to CUDA threads. Figure 7.5 shows the kernel from such a straightforward conversion. Each thread implements an iteration of the original outer loop. The original outer loop has M iterations, and M can be in the millions. We obviously need to have multiple thread blocks to generate enough threads to implement all these iterations.
To make performance tuning easy, we declare a constant `FHD_THREADS_PER_BLOCK` that defines the number of threads in each thread block when we invoke the `cmpFHd` kernel. Thus, we will use `M/FHD_THREADS_PER_BLOCK` for the grid size and `FHD_THREADS_PER_BLOCK` for block size for kernel invocation. Within the kernel, each thread calculates the original iteration of the outer loop that it is assigned to cover using the formula: `blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x`. For example, assume that there are 65,536 k-space samples and we decided to use 512 threads per block. The grid size at kernel innovation would be 65536/512=128 blocks. The block size would be 512. The calculation of `m` for each thread would be equivalent to `blockIdx.x*512 + threadIdx.x`.

While the kernel of Figure 7.5 exploits ample parallelism, it suffers from a major problem: all threads write into all voxel elements. This means that the kernel must use the atomic operations in the global memory in the inner loop in order to keep threads from trashing each other’s contributions to the voxel value. This can seriously affect the performance of kernel. Note that as is, the code will not even execute correctly. We need to explore other options.

```
for (m = 0; m < M; m++) {
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];

    for (n = 0; n < N; n++) {
        expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        cArg = cos(expFhD);
        sArg = sin(expFhD);
        rFhD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

(a) F^Hd computation

```
for (m = 0; m < M; m++) {
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];

    for (n = 0; n < N; n++) {
        expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        cArg = cos(expFhD);
        sArg = sin(expFhD);
        rFhD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

(b) after loop fission

Figure 7.6 Loop fission on the F^Hd computation.

From a quick inspection of Figure 7.4(b), we see that the F^Hd calculation can be split into two separate loops, as shown in Figure 7.6 using a technique called loop fission or loop splitting. This transformation takes the body of a loop and splits it into two loops. In the case of F^Hd, the outer loop consists of two parts: the statements before the inner loop and the inner loop. As shown in Figure 7.6(b), we can perform loop fission on the outer loop by placing the statements before the inner loop into a loop and the inner loop into a second
loop. The transformation changes the execution order of the two parts of the original outer loop. In the original outer loop, both parts of the first iteration execute before the second iteration. After fission, the first part of all iterations will execute; they are then followed by the second part of all iterations. The reader should be able to verify that this change of execution order does not affect the execution results for $F^H_d$. This is because the execution of the first part of each iteration does not depend on the result of the second part of any preceding iterations of the original outer loop. Loop fission is a transformation often done by advanced compilers that are capable of analyzing the (lack of) dependence between statements across loop iterations.

```c
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)
{
    int m = blockIdx.x * MU_THREADS_PER_BLOCK + threadIdx.x;
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
```

Figure 7.7 cmpMu kernel

With loop fission, the $F^H_d$ computation is now done in two steps. The first step is a single-level loop that calculates the $rMu$ and $iMu$ elements for use in the second loop. The second step corresponds to the loop that calculates the $F^H_d$ elements based on the $rMu$ and $iMu$ elements calculated in the first step. Each loop can now be converted into a CUDA kernel. The two CUDA kernels will execute sequentially. Since the second loop needs to use the results from the first loop, separating these two loops into two kernels that execute in sequence does not sacrifice any parallelism.

The cmpMu kernel in Figure 7.7 implements the first loop. The conversion of the first loop from sequential C code to a CUDA kernel is straightforward: each thread implements one iteration of the original C code. Since the $M$ value can be very big, reflecting the large number of k-space samples, such a mapping can result in a large number of threads. Since each thread block can have only up to 512 threads, we will need to use multiple blocks to allow the large number of threads. This can be accomplished by having each thread block to contain a number of threads, specified by MU_THREADS_PER_BLOCK in Figure 7.4(c), and employ M/MU_THREADS_PER_BLOCK blocks needed to cover all M iterations of the original loop. For example, if there are 65,536 k-space samples, the kernel could be invoked with a configuration of 512 threads per block and 65536/512=128 blocks. This is done by assigning 512 to MU_THREADS_PER_BLOCK and using MU_THREADS_PER_BLOCK as block size and M/MU_THREADS_PER_BLOCK as grid size during kernel innovation.

Within the kernel, each thread can identify the iteration assigned to it using its blockIdx and threadIdx values. Since the threading structure is one dimensional, only blockIdx.x and threadIdx.x need to be used. Because each block covers a section of the original iterations, the iteration covered by a thread is blockIdx.x*MU_THREADS_PER_BLOCK + threadIdx. For example, assume that MU_THREADS_PER_BLOCK=512. The thread
with blockIdx.x = 0 and threadIdx.x = 37 covers the 37th iteration of the original loop, whereas the thread with blockIdx.x = 5 and threadIdx.x = 2 covers the 2,562nd (5*512+2) iteration of the original loop. Using this iteration number to access the Mu, Phi, and D arrays ensures that the arrays are covered by the threads in the same way they were covered by the iterations of the original loop. Because every thread writes into its own Mu element, there is no potential conflict between any of these threads.

Determining the structure of the second kernel requires a little more work. An inspection of the second loop in Figure 7.4(b) shows that there are at least three options in designing the second kernel. In the first option, each thread corresponds to one iteration of the inner loop. This option creates the most number of threads and thus exploits the most amount of parallelism. However, the number of threads would be N*M, with both N in the range of millions and M in the range of hundred thousands. Their product would result in too many threads in the grid.

A second option is to use each thread to implement an iteration of the outer loop. This option employs fewer threads than the first option. Instead of generating N*M threads, this option generates M threads. Since M corresponds to the number of k-space samples and a large number of samples, on the order of a hundred thousand, are typically used to calculate F^dH, this option still exploits a large amount of parallelism. However, this kernel suffers the same problem as the kernel in Figure 7.5. The problem is that each thread will write into all voxels, thus creating an extremely large number of conflicts between threads. As is the case of Figure 7.5, the code in Figure 7.8 will not correctly without adding atomic operations that will significantly slow down the execution. Thus, this option does not work well.

A third option is to use each thread to compute one voxel element. This requires interchange the inner and outer loops and then use each thread to implement an iteration of the new outer loop. This is shown in Figure 7.9. Loop interchange is necessary because the loop being implemented by the threads must be the outer loop. Loop interchange makes each of
the new outer loop iteration to process a voxel element. Loop interchange is permissible here because all iterations of both levels of loops are independent of each other. They can be executed in any order relative to one another. Loop interchange, which changes the order of the iterations, is allowed when these iterations can be executed in any order. This option generates N threads. Since N corresponds to the number voxels in the reconstructed image, the N value can be very large for higher-resolution images. For a $128^3$ images, there are $128^3 = 2,097,152$ threads, resulting in a large amount of parallelism. For higher resolutions, such as $512^3$, we may need to invoke multiple kernels, each kernel generates the value of a subset of the voxels. Note these threads now all accumulate into their own rFhD and iFhd elements. There is no conflict between threads. These threads can run totally in parallel. This makes the third option the best choice among the three options.

The kernel derived from the interchanged loops is shown in Figure 7.10. The threads are organized into a two-level structure. Each thread covers an iteration of the new outer (n) loop: blockIdx.x*$FHD_THREADS_PER_BLOCK + threadIdx.x$. Once this iteration (n) value is identified, the thread executes the inner loop based on that n value. This kernel can be invoked by having each thread block to contain a number of threads, specified by a global constant $FHD_THREADS_PER_BLOCK$. Assuming N is the variable that gives the number of voxels in the reconstructed image, $N/FHD_THREADS_PER_BLOCK$ blocks cover all N iterations of the original loop. For example, if there are 65,536 k-space samples, the kernel could be invoked with a configuration of 512 threads per block and 65536/512=128 blocks. This is done by assigning 512 to $FHD_THREADS_PER_BLOCK$ and using $FHD_THREADS_PER_BLOCK$ as block size and $N/FHD_THREADS_PER_BLOCK$ as grid size during kernel innovation.

```
for (m = 0; m < M; m++) {
    for (n = 0; n < N; n++) {
        expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        cArg = cos(expFhD);
        sArg = sin(expFhD);
        rFhD[n] += rMu[m]*cArg - IMu[m]*sArg;
        iFhD[n] += IMu[m]*cArg + rMu[m]*sArg;
    }
}
```

(a) before loop interchange

```
for (n = 0; n < N; n++) {
    for (m = 0; m < M; m++) {
        expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
        cArg = cos(expFhD);
        sArg = sin(expFhD);
        rFhD[n] += rMu[m]*cArg - IMu[m]*sArg;
        iFhD[n] += IMu[m]*cArg + rMu[m]*sArg;
    }
}
```

(b) after loop interchange

Figure 7.9 Loop interchange of the FHD computation
Step 2: Getting Around the Memory Bandwidth Limitation

The simple cmpFhD kernel in Figure 7.10 will provide limited speedup due to memory bandwidth limitations. A simple analysis shows that the execution is limited by the low compute to memory access ratio of each thread. In the original loop, each iteration performs at least 14 memory accesses: \(kx[m], ky[m], kz[m], x[n], y[n], z[n], rMu[m] \) twice, \(iMu[m] \) twice, \(rFhD[n] \) read and write, and \(iFhD[n] \) read and write. Meanwhile, about 13 floating point multiple, add, or trigonometry operations are performed in each iteration. Therefore, the compute to memory access ratio is approximately 1, which is too low according to the analysis in Chapter 4.

We can immediately improve the compute-to-memory-access ratio by assigning some of the array elements to registers. As we discussed in Chapter 4, this can be done via
automatic variables. A quick review of the kernel in Figure 7.10 shows that for each thread, the same $x[n]$, $y[n]$, and $z[n]$ elements are used across all iterations of the `for` loop. This means that we can load these elements into automatic variables before the execution enters the loop. The kernel can then use the automatic variables inside the loop, thus converting global memory accesses to register accesses. Furthermore, the loop repeatedly reads from and writes into $rFHd[n]$ and $iFHd[n]$. We can have the iterations to read from and write into two automatic variables and only write the contents of these automatic variables into $rFHd[n]$ and $iFHd[n]$ after the execution exits the loop. The resulting code is shown in Figure 7.11. By increasing the number of register usage by 5 for each thread, we have reduced the memory access done by each iteration from 14 to 7. Thus, we have improved the compute to memory access ratio from 13:14 to 6:14. This is a very good improvement and a good use of the precious register resource.

Recall that the register usage can limit the number of blocks that can run in an SM. By increasing the register usage by 5, we really increase the register usage of thread blocks by 5*$FHD_THREADS_PER_BLOCK$. Assuming that we have 128 threads per block, we just increased the block register usage by 640. Since each SM can accommodate a combined register usage of 8912 registers among all blocks assigned to it, we need be careful that any further increase of register usage can begin to limit the number of blocks that can be assigned to an SM.

We need to further improve the compute to memory access ratio to something closer 10:1. We need to further eliminate global memory accesses in the cmpFHD kernel. The next candidates to consider are the k-space samples $kx[m]$, $ky[m]$ and $kz[m]$. These array elements are accessed differently than the $x[n]$, $y[n]$ and $z[n]$ elements: different elements of $kx$, $ky$ and $kz$ is accessed in each iteration of the loop in Figure 7.11. This means that we cannot load each k-space element into an automatic variable register and access that automatic variable off a register through all the iterations. So, the registers will not help here. However, we should notice that the k-space elements are not modified by the kernel. This means that we might be able to place the k-space elements into the constant memory. Perhaps the cache for the constant memory can eliminate most of the memory accesses.

A simple analysis of the loop in Figure 7.11 reveals that the k-space elements are indeed excellent candidates for constant memory. The index used for accessing $kx$, $ky$, and $kz$ is $m$, which is independent of threadIdx. This means that all threads in a warp will be accessing the same element of $kx$, $ky$, and $kz$. This is the ideal accessing pattern for cached constant memory: every time an element is brought into the cache, it will be used at least by all threads in a warp which is 32 in G80. This means that for every 32 accesses to the constant memory, at least 31 of them will be served by the cache. This allows the cache effectively eliminate about 96% of the accesses to the constant memory. Better yet, each time a constant is accessed from the cache, it can be broadcast to all the threads in a warp. This means that no delays are incurred due to any bank conflicts in the access to the cache. This makes constant memory as efficient as registers when accessing k-space elements.
There is, however, a technical issue involved in placing the k-space elements into the constant memory. Recall that constant memory has a capacity of 64KB. However, the size of the k-space samples can be much larger, in the order of hundreds of thousands or even millions. A typical way of working around the limitation of constant memory capacity is to breakdown a large data set to be placed into the constant memory into 64KB chunks. The developer must re-organize the kernel so that the kernel will be invoked multiple times, with each invocation of the kernel consuming only a 64KB chunk of the large data set. This turns out to be quite easy for the cmpFHD kernel.

```c
__constant__ float kx_c[CHUNK_SIZE],
    ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];

__ void main() {
    int i;

    for (i = 0; i < M/CHUNK_SIZE; i++){
        cudaMemcpy(kx_c, &kx[i*CHUNK_SIZE], 4*CHUNK_SIZE,
                    cudaMemcpyHostToDevice);
        cudaMemcpy(ky_c, &ky[i*CHUNK_SIZE], 4*CHUNK_SIZE,
                    cudaMemcpyHostToDevice);
        cudaMemcpy(kz_c, &kz[i*CHUNK_SIZE], 4*CHUNK_SIZE,
                    cudaMemcpyHostToDevice);

        cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>();
    }
}
```

Figure 7.12 Chunking k-space data to fit into constant memory.

A careful examination of the loop in Figure 7.11 reveals that all threads will be sequentially marching through the k-space arrays. For large data sets, the loop in the kernel simply iterates more times. This means that we can divide up the loop into sections, with each section processing a chunk of the k-space elements that fit into the 64KB capacity of the constant memory. The host code now invokes the kernel multiple times. Each time the host invokes the kernel, it places a new chunk into the constant memory before calling the kernel function. This is illustrated in Figure 7.12.

In Figure 7.12, the cmpFHD kernel is called from a loop. The code assumes that kx, ky, and kz are in the host memory. The dimension of these kx, ky, and kz are given by M. At each iteration, the host code calls the cudaMalloc() function to transfer a chunk of the k-space data into the device constant memory. The kernel is then invoked to process the chunk. Note that when M is not a perfect multiple of CHUNK_SIZE, the host code will need to have an additional round of cudaMemcpy and one more kernel invocation to finish the remaining k-space data.
__global__ void cmpFHd(float* rPhi, iPhi, phiMag, x, y, z, rMu, iMu, int M) {
    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhD_r = rFhD[n]; float iFhD_r = iFhD[n];

    for (m = 0; m < M; m++) {
        float expFHd = 2*PI*(kx[m]*xn_r+ky[m]*yn_r+kz[m]*zn_r);

        float cArg = cos(expFHd);
        float sArg = sin(expFHd);

        rFhD_r += rMu[m]*cArg - iMu[m]*sArg;
        iFhD_r += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}

Figure 7.13 Revised F^4d kernel to use constant memory

Figure 7.13 shows the revised kernel that accesses the k-space data from constant memory. Note that pointers to kx, ky, and kz are no longer in the parameter list of the kernel function. Since we cannot use pointers to access variables in the constant memory, the kx_c, ky_c, and kz_c arrays are accessed as global variables declared under __constant__ keyword as shown Figure 7.12. By accessing these elements from the constant cache, the kernel now has effectively only 4 global memory accesses to the rMu and iMu arrays. The compiler will typically recognize that the four array accesses are made to only two locations. It will only perform two global accesses, one to rMu[m] and one to iMu[m]. The values will be stored in temporary register variables for use in the other two. This makes the final number of memory accesses to 2. The computer to memory access ratio is down to 14:2, or 7:1. This is still not quite the desired 10:1 ratio but is sufficiently high that the memory bandwidth limitation is no longer the only factor that limits performance. As we will see, we can perform a few other optimizations that make computation more efficient and further improve performance.

If we ran the code in Figures 12 and 13, we would have found out that the performance enhancement was not as high as we expected. As it turns out, the code shown in these
figures does not result in as much memory bandwidth reduction as we expected. The reason is that the constant cache does not perform very well for the code. This has to do with the design of the constant cache and the memory layout of the k-space data. As shown in Figure 14, each constant cache entry is designed to store multiple consecutive words. This design reduces the overhead of bookkeeping in the hardware design. If multiple data elements that are used by each thread are not in consecutive words, as illustrated in Figure 14(a), they will end up taking up multiple cache entries. Due to cost constraints, the constant cache has only a very small number of entries. As shown in Figure 12 and 13, the k-space data is stored in three arrays: \texttt{kx\_c, ky\_c}, and \texttt{kz\_c}. During each iteration of the loop, three entries of the constant cache is needed to hold the three k-space element being processed. Since different warps can be at very different iterations, they may require many entries altogether. As it turns out, the G80 cache capacity was not sufficient to provide sufficient number of entries for all the warps active in an SM.

```
struct kdata {
    float x, float y, float z;
} k;

__constant__ struct kdata k_c[CHUNK_SIZE];

__ void main() {
    int i;
    for (i = 0; i < M/CHUNK_SIZE; i++) {
        cudaMemcpy(k_c,k,12*CHUNK_SIZE, cudaMemcpyHostToDevice);
        cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>> ();
    }
}
```

Figure 7.15 adjusting k-space data layout to improve cache efficiency

The problem of inefficient use of cache entries has been well studied in the literature and can be solved by adjusting the memory layout of the k-space data. The solution is illustrated in Figure 14(b) and the code based on this solution in Figures 15. Rather than having the \( \text{x, y, and z} \) components of the k-space data to be stored in three separate arrays, the solution stores these components in an array whose elements are structs. The declaration of the array is shown on top of Figure 15. By storing the \( \text{x, y, and z} \) components in the three fields of an array element, the developer forces these components to be stored in consecutive locations of the constant memory. Therefore, all three components used by an iteration can now fit into one cache entry, reducing the number of entries needed to support the execution of all the active warps. Note that since we have only one array to hold all k-space data, we can just use one cudaMemcpy to copy the entire chunk to the device constant memory. The size of the transfer is adjusted from 4*CHUNK\_SIZE to 12*CHUNK\_SIZE to reflect the transfer of all the three components in one cudaMemcpy call.
Figure 7.16 Adjusting the k-space data memory layout in the FHD kernel

With the new data structure layout, we also need to revise the kernel so that the access is done according to the new layout. The new kernel is shown in Figure 7.16. Note that kx[m] has become k[m].x, ky[m] has become k[m].y, and so on.
__global__ void cmpFHd(float* rPhi, iPhi, phiMag, 
x, y, z, rMu, iMu, int M) {
    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

    for (m = 0; m < M; m++) {
        float expFhD = 2*PI*(k[m]*x*n_r+k[m]*y*n_r+k[m]*z*n_r);
        float cArg = __cos(expFhD);
        float sArg = __sin(expFhD);

        rFhDn_r += rMu[m]*cArg - iMu[m]*sArg;
        iFhDn_r += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}

Figure 7.17 using hardware __sin() and __cos() functions.

Step 3: Using hardware trigonometry functions
As we discussed in Chapter 5, CUDA offers hardware mathematic functions that offer much higher throughput than their software counter parts. These functions are implemented in the SFU (Super Function Units) in G80. The procedure for using these functions is quite easy. In the case of the cmpFHd kernel, what we need to do is to change the calls to sin and cos functions into their hardware versions: __sin and __cos. Because, these functions are called in a heavily executed loop body, we expect that the change will result in very significant performance improvement. The resulting cmpFHd kernel is shown in Figure 7.17.


\[
MSE = \frac{1}{mn} \sum_{i} \sum_{j} (I(i, j) - I_0(i, j))^2 \quad A_j = \frac{1}{mn} \sum_{i} \sum_{j} I_0(i, j)^2
\]

\[
PSNR = 20 \log_{10} \left( \frac{\max(I_0(i, j))}{\sqrt{MSE}} \right) \quad SNR = 20 \log_{10} \left( \frac{\sqrt{A_j}}{\sqrt{MSE}} \right)
\]

Figure 7.18 metrics used to validate the accuracy of hardware functions
I0 is perfect image. I is reconstructed image.
PSNR is Peak signal-to-noise ratio, SNR is Signal-to-noise ratio.

We need to, however, be careful about the reduced accuracy when switching from software functions to hardware functions. As we discussed in Chapter 6, Hardware implementation currently have slightly less accuracy than software libraries. In the case of MRI, we need to make sure that the hardware implementation passes test cases that measure the Signal to Noise Ration (SNR) of the resulting image, as shown in Figure 7.18. The testing process involves a “perfect” image (I0). We then use a reverse process to generate a corresponding
“scanned” k-space data that is actually synthesized. The synthesized scanned data is then processed by the proposed reconstruction system to generate a reconstructed image (I). The value of the voxels in the images are then plugged into the MSE, PSNR, and SNR formula in Figure 7.18.

The criteria for passing the test depend on the application that the image is intended for. In our case, we worked with the MRI clinical experts to ensure that the SNR changes due to hardware functions is well within the accepted limits for their applications. In applications where the images are used by physicians to form impression of injury or disease evaluation, one also needs to have visual inspection of the image quality. Figure 7.19 shows the visual comparison of the original “true” image. It then shows that the SNR achieved CPU double precision and single precision implementation both achieve 27.6 dB, an well acceptable level for the application. A visual inspection also shows that the reconstructed image indeed correspond well with the original image.

The advantage of iterative reconstruction compared to a simple bilinear interpolation gridding/FFT is also obvious in Figure 7.19. The image reconstructed with the simple gridding/FFT has an SNR of only 16.8 dB, substantially lower than that of the iterative reconstruction method. A visual inspection of the gridding/FFT image shows that there is severe artifact that can significantly impact the usability of the image for diagnosis purposes.

When we move from double precision to single precision arithmetic on the CPU, there was no measurable degradation of SNR, which remains at 27.6 dB. When we move the trig function from software library to the hardware units, we observed a negligible degradation.
of SNR, from 27.6 dB to 27.5 dB. A visual inspection shows that the reconstructed image does not have significant artifact compared to the original image.

**Step 4: Experimental Performance Tuning**

Up to this point, we have not determined the appropriate values for the configuration parameters for the kernel. For example, we need to determine the best number of threads for each block. On one hand, using a large number of threads in a block is needed to reach thread capacity of each SM given that up to eight blocks can be assigned to each SM. On the other hand, having more threads in each block increases the register usage of each block and can reduce the number of blocks that can fit into an SM. Some possible values of number of threads per block are 32, 64, 128, 256, and 512. One can also consider non-power-of-two numbers.

One also needs to determine the number of scan points per grid. All the scan point data consumed by a grid must fit into the 64KB constant memory. This is 16K single precision. Since each scan point requires three single-precision floating point data, we can have up to 4K scan points if we want to use power-of-two scan points in each grid for convenient loop control. Some possible numbers are 32, 64, 128, 256, 1024, 2048, and 4096.

Another kernel configuration parameter is the number of times one should unroll the body of the `for` loop. On one hand, unrolling the loop can reduce the number of overhead instructions, and potentially reduce the number of clock cycles to process each k-space sample data. On the other hand, too much unrolling can increase the usage of registers and reduce the number of blocks that can fit into an SM.

Note that these configurations are not independent of each other. Increasing one parameter value can potentially use the resource that could be used to increase another parameter value. As a result, one needs to evaluate these parameters jointly in an experimental manner. That is, one may need to change the source code for each joint configuration and measure the run time. There can be a large number of source code versions to try. In the case of F3D, the performance improves about 20% by systematically searching all the combinations and choosing the one with the best measured runtime, as compared to a heuristic tuning search effort that explore some promising trends.

**7.4. Final Evaluation**

To obtain a reasonable baseline, we implemented two versions of FHd on the CPU. Version CPU.DP uses double-precision for all floating-point values and operations, while version CPU.SP uses single-precision. Both CPU versions are compiled with Intel’s icpc (version 10.1) using flags -O3 -msse3 -axT -vec-report3 -fp-model fast = 2, which (1) vectorizes the algorithm’s dominant loops using instructions tuned for the Core 2 architecture, and (2) links the trigonometric operations to fast, approximate functions in the math library. Based on experimental tuning with a smaller data set, the inner loops are unrolled by a factor of four and the scan data is tiled to improve locality in the L1 cache.
Each GPU version of FHd is compiled using nvcc -O3 (CUDA version 1.1) and executed on a 1.35 GHz Quadro FX 5600. The Quadro card is housed in a system with a 2.4 GHz dual-socket, dual-core Opteron 2216 CPU. Each core has a 1 MB L2 cache. The CPU versions use pthreads to execute on all four cores of 2.66 GHz Core 2 Extreme quad-core CPU, which has peak theoretical capacity of 21.2 GFLOPS per core and a 4 MB L2 cache. The CPU versions perform substantially better on the Core 2 Extreme quad-core than on the dual-socket, dual-core Opteron.

All reconstructions use the GPU version of the linear solver, which executes 60 iterations on the Quadro FX 5600. Two versions of Q were computed on the Core 2 Extreme, one using double-precision and the other using single-precision. The singleprecision Q was used for all GPU-based reconstructions and for the reconstruction involving CPU.SP, while the double-precision Q was used only for the reconstruction involving CPU.DP. As the computation of Q is not on the reconstruction’s critical path, we give Q no further consideration.

To facilitate comparison of the iterative reconstruction with a conventional reconstruction, we also evaluated a reconstruction based on bilinear interpolation gridding and inverse FFT. Our version of the gridded reconstruction is not optimized for performance, but it is already quite fast.

All reconstructions are performed on sample data obtained from a simulated, three-dimensional, non-Cartesian scan of a phantom image. There are 284,592 sample points in the scan data set, and the image is reconstructed at 1,283 resolution, for a total of 221 voxels. In the first set of experiments, the simulated data contains no noise. In the second set of experiments, we added complex white Gaussian noise to the simulated data. When determining the quality of the reconstructed images, the percent error and peak signal-to-noise ratio metrics are used. The percent error is the root-mean-square (RMS) of the voxel error divided by the RMS voxel value in the true image (after the true image has been sampled at 1283 resolution).

The data (runtime, GFLOPS, and images) were obtained by reconstructing each image once with each of the implementations of the FHd algorithm described above. There are two exceptions to this policy. For GPU.Tune and GPU.Multi, the time required to compute FHd is so small that run-
time variations in performance may become non-negligible. Therefore, for these configurations we computed $F_{hd}$ three times and reported the average performance.

As shown in Figure 7.20, the total reconstruction time for the test image using bilinear interpolation gridding followed by inverse FFT takes only less than one minute on a high-end sequential CPU. This confirms that there is little value in parallelizing this traditional reconstruction strategy. It is, however, obvious from Figure 7.19(2) that the resulting image exhibits an unacceptable level of artifact.

The LS (CPU, DP) row shows the execution timing of reconstructing the test image using double-precision floating-point arithmetic on the CPU. The timing shows that the core step (Q) of calculating $F_{hd}^*F + \lambda W_{hd}^H W$. The first observation is that the Q computation for a moderate resolution image based on a moderate sized data sample takes an unacceptable amount of time (more than 65 hours) on the CPU for setting up the system for a patient. Note that this time is eventually reduced to 6.5 minutes on the GPU with all the optimizations described in Section 7.3. The second observation is that the total reconstruction time of each image requires more than 8 hours, with only 1.59 minutes spent in the linear solver. This validates our decision to focus our parallelization effort on $F_{hd}$.

The LS(CPU, SP) row shows that we can reduce the execution time significantly when we covert the computation from double-precision floating-point arithmetic to single-precision on the CPU. This is because the SSE instructions have higher throughput, i.e., calculate more data elements per clock cycle when executing in single-precision mode. The execution times, however, are still unacceptable for practical use.

The LS(GPU, naïve) row shows that a straightforward CUDA implementation can achieve a speedup about 10 times for Q and 8 times for the reconstruction of each image. This is a good speedup but the resulting execution times are still unacceptable for practical use.

The LS(GPU, Cmem) row shows that significant further speedup is achieved by using registers and constant cache to get around the global memory bandwidth limitations. These enhancements achieve about 4X speedup over the naïve CUDA code! This shows the importance of achieving good compute to memory ratios in CUDA kernels. These enhancements bring the CUDA code to about 40X speedup over the single precision CPU code.

The LS(GPU, CMem, SPU, exp) row shows the use of hardware trigonometry functions and experimental tuning together results in dramatic further speedup. A separate experiment, not shown here, shows that most of the speedup comes from hardware trigonometry functions. The total speedup over CPU single-precision code is very impressive: 357X for Q and 108X for the reconstruction of each image.

An interesting observation is that in the end, the linear solver actually takes more time that $F_{hd}^*F$. This is because we have accelerated $F_{hd}^*F$ dramatically (228X). What used to be close to 100% of the per image reconstruction time now accounts for less than 50%. Any further
acceleration will now require acceleration of the linear solver, a much more difficult type of computation for massively parallel execution.