Chapter 2
A Brief History and Introduction to GPGPU

Richard Vuduc and Jee Choi

Abstract Graphics processing units (GPU) are increasing the speed and volume of computation possible in scientific computing applications. Much of the rapidly growing interest in GPUs today stems from the numerous reports of 10–100-fold speedups over traditional CPU-based systems. In contrast to traditional CPUs, which are designed to extract as much parallelism and performance as possible from sequential programs, GPUs are designed to efficiently execute explicitly parallel programs. In particular, GPUs excel in the case of programs that have inherent *data-level parallelism*, in which one applies the same operation to many data simultaneously. Applications in scientific computing frequently fit this model. This introductory chapter gives a brief history and overview of modern GPU systems, including the dominant programming model that has made the compute capabilities of GPUs accessible to the programming-literate scientific community. Its examples focus on GPU hardware and programming environment available from a particular vendor, NVIDIA, though the main principles apply to other systems.

Keywords CUDA • GPGPU • High-performance computing • Parallel programming • SIMD • Vector processors

2.1 A Brief History

The broader adoption of GPUs today stems from advances in programming and in the use of power-efficient architectural designs. We briefly recount these advances below.

R. Vuduc (✉) • J. Choi
School of Computational Science & Engineering, College of Computing,
Georgia Institute of Technology, Atlanta, GA 30332, USA
e-mail: richie@cc.gatech.edu; jee@gatech.edu

2.1.1 From Special-Purpose to General-Purpose Programming

GPUs were originally designed to accelerate graphics related tasks such as texture mapping or vertex shading commonly used in computer games and 3D graphics to render images. During the early 2000s, to increase the realism of 3D games and graphics, GPU designs were enhanced to deliver more *polygons per second* of performance. This style of geometric computation is heavily floating-point intensive, and the raw throughput of such operations vastly outstripped the capabilities of conventional, albeit general-purpose, CPU processors. Lured by these capabilities, developers of scientific software began looking for ways to exploit GPUs beyond the graphics rendering tasks for which they were designed.

Early GPUs were hard to program for anything other than graphics applications. The hardware consisted of fixed graphics pipelines; to program them, one had to use specific application programming interfaces (API) such as OpenGL and DirectX, or shader languages such as C for Graphics (Cg). (Regarding terminology, we will refer to languages and/or libraries needed to program a machine as *programming models*.) If one wished to program something other than graphics, it was difficult to do so since every operation had to be mapped to an equivalent graphics operation.

There were numerous efforts in the research community to ease GPU programming through high-level language extensions (Michael et al. 2002; Ian Buck et al. 2004). This research was the basis for NVIDIA's Compute Unified Device Architecture (CUDA), a more general-purpose computing platform and programming model, which is arguably the dominant albeit largely vendor-specific model available today. CUDA allows developers to use C (as well as many other languages, APIs and directives) as a high-level programming language for programming NVIDIA GPUs. Uptake of CUDA within the scientific research community has been reasonably swift, with approximately 37,000 published research papers on the use of CUDA and 1.6 million CUDA downloads as of 03/19/13. Arguably, CUDA and its ilk have turned an otherwise special-purpose platform into one suited to general-purpose use, hence the term *general-purpose GPU* (GPGPU) computing.

2.1.2 Single-Instruction Multiple-Data Designs

Concomitant with the advances in programming models were advances in the hardware design itself. A key idea in GPU architectures is *single instruction multiple data* (SIMD) design, a type of parallel computer architecture in which a single instruction prescribes that the machine perform a computational operation on many words of data simultaneously (Flynn 1972). The first SIMD computer was the ILLIAC-IV, built in the late 1960s (Bouknight et al. 1972). It was later followed by other systems including ICL's Distributed Array Processor (DAP) (Reddaway 1973). However, it was not until the 1980s that interest in SIMD computers peaked and led to the development of Connection Machine CM-1 (Hillis 1982) and MasPar

MP-1 (Blank 1990). Many of these systems had common architectural features: there was typically one or more central processing units or sequencers that fetched and decoded instructions; instructions were then broadcast to an array of simple, interconnected processing elements.

Of these systems, Connection Machine CM-1 perhaps most closely resembles modern GPUs. The CM-1 system consists of four processor arrays, each consisting of 16,000 processors, and from one to four front-end computers depending on how the processor arrays are used; the arrays can be used separately, in pairs, or as a single unit. The front-end computers control the arrays and issue instructions to them. The CM-1 also provides a virtual number of processors to fit application needs; then, the physical processors are time-sliced over multiple data regions that have been assigned to it, enabling effective and flexible programming and utilization of the available processors. As we will see later, this virtual processor abstraction closely resembles modern GPGPU programming.

Modern GPUs have a come a long way since the days of CM-1 and MasPar. For under $500, one can now purchase a desktop GPU that can execute 3.5 trillion floating-point operations per second (teraflop per second, or TFLOP/s) within a power footprint of just 200 Watts. The basic enabling idea is a SIMD execution style.

2.2 Overview of GPGPU Hardware and Programming Today

The CUDA programming model, introduced by NVIDIA in November 2006, simplifies how one may express data parallel programs in a general-purpose programming environment. CUDA originally extended the C++ language, with recent additional support for Fortran and Java. This chapter summarizes the key C++ extensions; we encourage interested readers to the latest CUDA Programming Guide for more details (NVIDIA 2013).

2.2.1 A GPU-Based System

Figure 2.1 shows a typical GPU system. It consists namely of one or more GPU *devices*, which is connected to a *host* system. The host system is a conventional computer consisting of one or more general-purpose CPUs and a communication channel between them and the GPU(s). On desktop and workstation systems, this channel is typically a PCI Express (PCIe) bus.

This host-device design implies an *offload* model of execution. In particular, an application begins running on the host and then offloads computation to the GPU device. Moreover, the programmer must copy data from the host to the GPU device and copy any needed results from the device back to the host as needed. The host and device may otherwise execute simultaneously (and asynchronously), as permitted by the application.

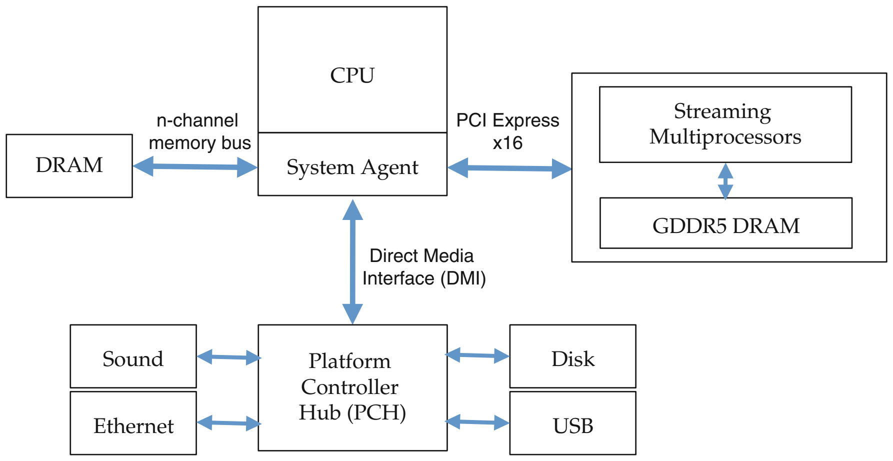


Fig. 2.1 A typical GPU system

This system design has both performance and programming implications. Regarding performance, an application will only see an improvement from use of the GPU if the data transfer time is small relative to the speedup from using the GPU in the first place. Regarding programming, the programmer will write both host code and device code, that is, code that only runs on the host and separate code that may only run on the device; he or she must also coordinate data transfer as the application requires.

Briefly, the typical steps involved in writing a CUDA program may be as follows:

1. Allocate and initialize input and output data structures on host.
2. Allocate input and output data structures on device.
3. Transfer input data from host to device.
4. Execute device code.
5. Transfer output data from device to host.

CUDA provides an API for manipulating the device (e.g., allocating memory on the device) and a compiler (*nvcc*) for compiling the device code.

2.2.2 GPU Architecture

Before we can go into the details of the CUDA programming model, it is necessary to talk about the GPU architecture in more detail. This is because the CUDA programming model (as well as the other models) closely reflects the architectural design. This close matching of programming model to architecture is what enables programs that can effectively use the GPU's capabilities. One implication is that although the programming model abstracts away some details of the hardware, for a program to really execute efficiently, a programmer must understand and exploit a number of hardware details.

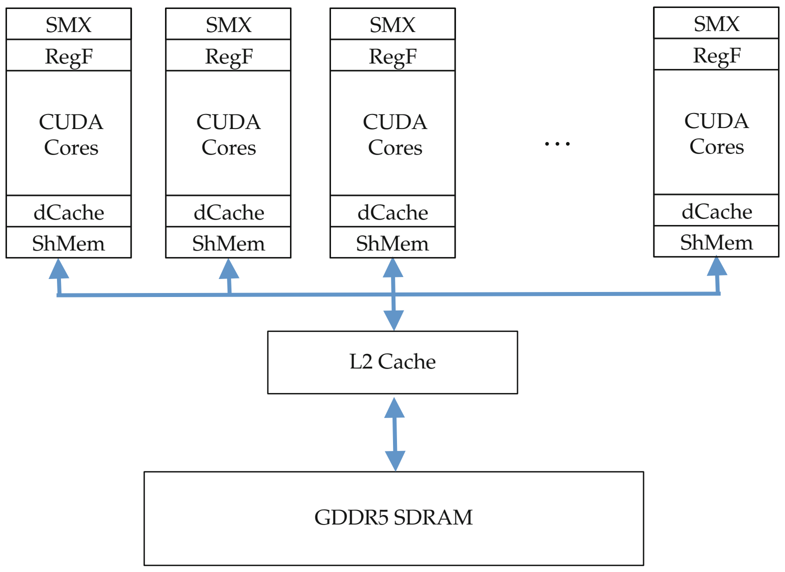


Fig. 2.2 Diagram of a basic GPU architecture

Figure 2.2 shows a basic GPU architecture. First, the GPU has both processing units, which perform actual computation, and a main memory, which store all of the data on which the GPU may operate. The processing units are labeled *Streaming Multiprocessors* (SMX) in the figure, and the memory is labeled *SDRAM*. (The *GDDR5* designation refers to the particular type of memory and protocols used when transferring data).

The SMX units are further divided into several components. At the heart of an SMX is a number of small *functional units* called "CUDA cores." One may regard these CUDA cores as the basic hardware for performing primitive operations, such as floating-point operations.

These CUDA cores share a number of other hardware resources, the most important of which is the *register file* (RegF). A CUDA core may only perform primitive operations on data stored in registers. Roughly speaking, variables (i.e., "local variables") that appear in a CUDA program are stored in registers. The compiler, when it translates the CUDA program into machine code, manages the use of these registers and the mapping of program variables to registers. As such, they are not really exposed to the programmer. However, to achieve good performance, a programmer should keep in mind that since registers are directly connected to the cores, any data stored in registers will be much faster to access than data stored elsewhere in the system.

Indeed, this basic principle underlying register use applies more generally to the entire *memory hierarchy* of the GPU. The memory hierarchy refers to the collection of memories including the main memory (SDRAM), the registers, and a number of intermediate memories. The relative speed of accessing data in registers may be 100× or 1000× faster than doing so from main memory. However, the capacity of all the

GPU's registers compared to main memory may also differ by that same factor. For instance, the aggregate register capacity across all SMX units typically numbers in the millions of bytes, or megabytes, MB; by contrast, the capacity of the main memory is much larger, numbering typically in the billions of bytes, or gigabytes (GB). Therefore, using the memory hierarchy effectively is critical to achieving speedups in practice.

The hierarchy refers to additional intermediate staging levels of progressively smaller but faster memories between main memory and the register file. In Fig. 2.2, these are the *caches* (i.e., the so-called level-1 or *dCache* and the so-called level-2 or *L2 Cache*) and the *shared memory* (*ShMem*). Caches are managed automatically by the hardware: the caching hardware "observes" the data requests from the processing units and try to keep frequently accessed data in the cache. By contrast, the shared memory is programmer-managed: the programmer controls exactly what data is in the shared memory at any point in time during the computation.

There are some additional useful details to know about the memory hierarchy. The CUDA cores on a given SMX share the dCache; all SMX units on the GPU share the L2 Cache. Regarding their relative capacities, the CUDA cores of a given SMX share the register file (typical capacity is 256 KB). The total ShMem and dCache capacity is 64 KB per SMX, where the ShMem and dCache may actually be re-configured by the programmer to be either 16 KB of programmer-managed memory with 48 KB of dCache, or 48 KB of programmer-managed memory and 16 KB of dCache, or split equally between the two on current systems.

More concretely, Table 2.1 shows the exact specifications for the latest NVIDIA GPU, the GTX Titan (released in late 2012). The sheer number of CUDA cores is what makes GPUs so compute-rich. Exploiting this relatively large amount of parallelism—thousands, compared to tens on conventional processors—requires a "non-traditional" programming model.

2.2.3 SIMT and Hardware Multithreading

To fully utilize the available CUDA cores on a GPU, CUDA adopts a variation of SIMD, which NVIDIA refers to as the *single instruction multiple thread* (SIMT) style.

In SIMT, a program consists of a number of threads and all threads execute the same sequence of instructions. Therefore, if all threads execute the same instruction at the same time, just on different data per thread, a CUDA program would simply be a sequence of SIMD instructions.

However, SIMT generalizes SIMD in that it allows individual threads to execute different instructions. This situation occurs when, for instance, threads simultaneously execute a conditional (e.g., "if" statement) but execute different branches. When this occurs, threads are said to *diverge*. Although this behavior is allowed by the model, it has an important performance implication. When threads diverge, their execution is serialized—that is, the threads that take one branch may execute first while the other threads idle until the first threads complete. In other words, SIMT allows flexibility in programming at the cost of performance. For example, the only way to achieve the peak performance as calculated in Table 2.1 is to have absolutely no divergent threads in the code.

Table 2.1 Specifications for the NVIDIA GTX Titan

Parameters	Values
SMX	14
32-bit (Single precision) CUDA Cores	2,688 (192 cores/SMX)
64-bit (Double precision) CUDA Cores	896 (64 cores/SMX)
Clock	837 MHz
Boost clock	876 MHz
Memory interface width	384 bits
Memory clock	6.008 GHz [a]
Single precision peak performance	4709.38 GFLOP/s [b]
Double precision peak performance	1569.79 GFLOP/s
Peak bandwidth	288.38 GB/s [c]

[a] Effective clock rate since DDR memory reads from both rising and falling edges of the signal
[b] Peak Performance = (number of cores) × (boost clock) × 2 FLOP/cycle (fused multiply-add)
[c] Peak Bandwidth = (memory interface width) × (memory clock)

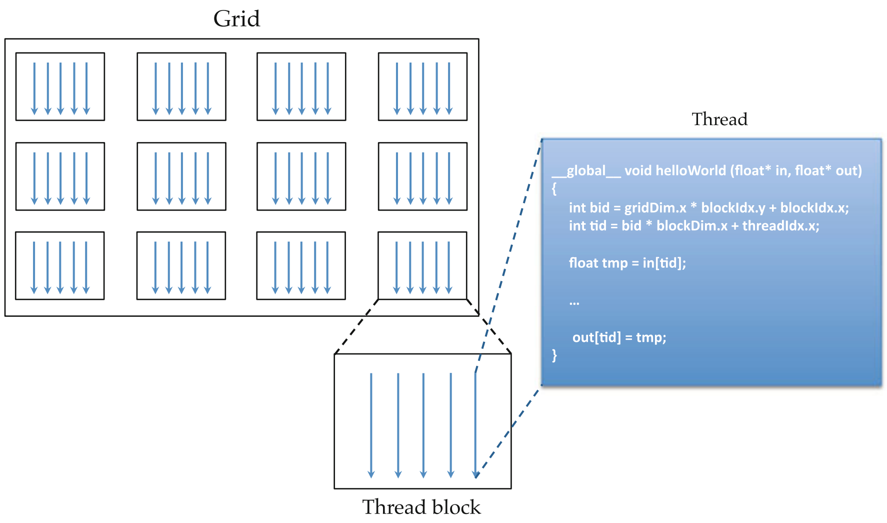
In order to have many CUDA cores, a key design decision in GPUs is to also use simpler cores, compared to their equivalents in CPU systems. Simpler cores tend to be smaller and more energy-efficient, but also slower *per instruction*. However, the long latencies associated with these slower instructions—as well as the relatively slower memory operations—may be mitigated by allowing many simultaneous instructions to be in-flight at any time, with the processors juggling among available instructions. That is, at any given time, a GPU keeps the context of a large number of threads on the chip so that it can switch from one set of threads to another quickly whenever threads are ready to execute. This approach is referred to as *hardware multithreading*.[1]

Indeed, hardware multithreading is feasible because of the SIMT approach, which enables (or rather, *requires*) the programmer to express a large amount of thread-level parallelism. On the hardware side, generous amounts of on-chip resources, such as the large register file and shared memory, are also necessary. Although conventional CPUs also employ hardware multithreading (e.g. Intel's hyperthreading technology), the number of in-flight threads is much smaller—compare 2 in-flight threads per core using Intel's hyperthreading technology, compared to, say, 64 threads per SMX on a representative GPU.

2.3 CUDA

Here we go into the specifics of the CUDA programming model. We will first cover three key concepts: the *thread hierarchy*, the *memory hierarchy*, and *synchronization*. Then, we will discuss commonly used strategies for optimizing CUDA programs for performance. All hardware specifications mentioned in this section will be that of the GTX Titan which is based on the latest generation of NVIDIA's GPU architecture, the *GK110*.

[1] It also explains why advocates of GPU design refer to GPUs as being especially suited to *throughput-oriented* execution, rather than *latency-oriented execution* as in CPUs.



```
__global__ void helloWorld (float* in, float* out)
{
    int bid = gridDim.x * blockIdx.y + blockIdx.x;
    int tid = bid * blockDim.x + threadIdx.x;

    float tmp = in[tid];

    ...

    out[tid] = tmp;
}
```

Fig. 2.3 CUDA thread hierarchy

2.3.1 Thread Hierarchy

There are three layers of hierarchy for CUDA threads; threads, thread blocks, and grids. The relationship between these three layers are shown in Fig. 2.3.

Recall that the smallest granularity of computation on a GPU is a *thread*, which simply executes a sequence of instructions, or *kernel*. A programmer may think of the kernel as the "program" that all threads execute simultaneously. While executing the kernel code, each thread has a unique integer identifier that it can use to determine whether to perform thread-specific operations, such as loading a particular data element. A typical CUDA program consists of anywhere from thousands to millions of these threads.

Threads are grouped into *thread blocks*. A thread block may contain any number of threads up to some limit, which on current hardware is 1,024 threads per block. Importantly, thread blocks execute completely *independently*. That is, once a thread block begins execution, it runs to completion; moreover, there is no guarantee on what order thread blocks will execute, so a programmer should not rely on this fact. Additionally, the thread block may be logically configured as a one-, two-, or three-dimensional array of threads. The purpose of a multi-dimensional configuration of threads is to allow for easier mapping of threads to work.[2]

[2] For instance, if a GPU kernel operates on a 2-D image, it is very likely most "natural" to assign one thread to perform a computation on each pixel; in this case, a 2-D logical organization of threads is likely to be a sensible way of mapping threads to work.

The significance of a thread block is scalability. When threads are assigned to a multiprocessor, they are done so in the granularity of thread blocks. Threads within a thread block may coordinate their work, but thread blocks—being executed independently as noted previously—may not. Additionally, a thread block completes only when every thread in the block has finished its work. This behavior permits scaling a workload with sufficiently many thread blocks onto GPUs that may have differing numbers of SMXs.

Thread blocks may be further logically organized into *grids*. That is, a grid is a collection of thread blocks, also configurable as a one-, two-, or three-dimensional array. Grid size and dimensions are dictated typically by either the work-thread block mapping, or the total number of thread blocks. A grid can be seen as the entire collection of CUDA threads that will execute a given kernel.

2.3.2 Memory Hierarchy

As noted in the architectural overview above, GPUs have multi-level memory hierarchy. This hierarchy is similar to traditional CPUs; on current generation GPUs, there is a level-2 (L2) cache that is hardware-managed and shared between all multiprocessor cores on the GPU and a level-1 (L1) cache that is local to each core. On current GPUs, the L2 cache is 1,536 KB in size, and all data accesses through the SDRAM, whether read or write, is stored in the L2 cache. The L1 cache is reserved for local memory usage such as register spills or local arrays that are either too large or cannot be indexed with constants.

Additionally, there is also the programmer-managed shared memory. The hardware used to implement shared memory is identical to that of the L1 cache and together they make up 64 KB in size. As needed, shared memory can be configured to be 16, 32, or 48 KB, with the L1 cache taking up the rest. Shared memory is local to a thread block, and is a limited resource that can constrain the number of thread blocks that can be resident on the multiprocessor at the same time.

There is also a 48 KB read-only data cache called constant memory. This is part of the *texture* unit that is necessary for graphics performance, but is also available for general purpose use. The availability of constant memory relieves pressure on the shared and L1 cache by reducing conflict misses.

Lastly, GPUs have large register files to support hardware multithreading. The size of the register file is 256 KB on the GTX Titan and each thread can use as many as 255 32-bit registers.

One interesting point to note is that unlike traditional CPUs where closer you get to the core, the smaller the cache becomes, GPUs show an opposite trend. That is, when going from L2 to L1 to register, the size of the cache goes from 1,536 KB to 1,568 KB to 3,584 KB.

From a programming perspective, CUDA exposes the shared memory to the programmer. That is, the programmer controls completely how data is placed in shared memory. By contrast, the compiler controls registers (though the programmer may influence their use) and the hardware manages the caches.

2.3.3 Synchronization

We noted previously that threads within a thread block may coordinate their activity. The mechanism for doing so is shared data access and synchronization.

With respect to shared data access, threads within a thread block all "see" the same shared memory data. Therefore, they may cooperatively manage this data. For instance, if the thread block needs to perform computation on some chunk of data, each thread may be assigned by the programmer to load a designated piece of that data.

With respect to synchronization, threads may "meet" at a particular program point within the kernel. This operation is called a *barrier*—when a thread executes a barrier, it waits until all other threads have also reached the barrier before continuing. For example, a common pattern is for threads to cooperatively load a chunk of data from main memory to shared memory as noted above, then issue a barrier. By doing so, the threads ensure that all the data is loaded before any thread proceeds with computation.

Importantly, this type of synchronization is really only possible within a thread block, but not between thread blocks, since there is no guarantee on how thread blocks may be scheduled. The only way to do global synchronization across thread blocks is to do so on the host: the host is what launches the GPU kernel, and the CUDA API makes a host function available to check whether all thread blocks have completed.

2.3.4 Performance Considerations

2.3.4.1 Bandwidth Utilization

GTX Titan boasts 288.4 GB/s of bandwidth, an order of magnitude higher than most CPUs, due to its high memory clock frequency and wide interface width of its GDDR5 SDRAM. However, in order to fully utilize the available bandwidth, certain precautions must be taken on how the data is loaded from the memory.

By a principle known as *Little's Law* (from queuing theory), in order to saturate the memory system, the number of in-flight memory requests must be approximately equal to the memory bandwidth multiplied by the latency between the SDRAM and the multiprocessor core. Latency today is typically in the range of 400–800 clock cycles. Consequently, a GPU kernel in practice needs to have several tens of thousands of bytes of data being requested simultaneously. A computation that cannot have that level of memory-level parallelism will not make the most efficient use of the GPU's memory transfer capability.

Another consideration is to ensure that loads from GPU memory (either to shared memory or to registers) are *coalesced*. Conceptually, the GPU memory system is designed to load chunks of consecutive data at a time. Furthermore, the hardware typically schedules consecutively numbered threads at the same time.

Therefore, a good CUDA programming practice is to ensure that consecutively numbered threads that are performing load or store operations do so to consecutive locations in memory. An easy way to make sure data is coalesced is to *stream* the data where threads accesses one or more words of data consecutively.

2.3.4.2 Core Utilization

To understand how to maximize the use of CUDA cores on a GPU, we must briefly sketch how threads are executed on a GPU.

Although threads begin and complete on the multiprocessor cores of a GPU at the granularity of thread blocks, execution of threads occurs at a smaller granularity. This minimum unit of execution is called a *warp*. (On more classical vector processors, a warp is analogous to the vector or SIMD width.) Although the size of a warp can vary across different GPUs, the warp size has always been 32. This value will most likely increase in the future as the number of cores continues to increase with each successive generation.

On a current generation GTX Titan, there are 4 *warp schedulers* on each multiprocessor. Each warp scheduler can take a warp of threads from the same or different thread blocks currently residing on the multiprocessor and issue one or two instructions, depending on the availability of CUDA cores and the number of independent instructions in each warp, to 32 or 64 CUDA cores respectively. Since there are 192 CUDA cores in each multiprocessor, at least 6 sets of instructions need to be issued *every* cycle in order to fully occupy the CUDA cores. This is equivalent to 2 warp schedulers issuing 1 instruction each, and 2 warp schedulers issuing 2 instructions each. This means that at least 2 of the warps need to have 2 independent instructions that can be scheduled simultaneously. Having only data parallelism is therefore not enough to achieve peak performance on the GTX Titan; instruction level parallelism is also required.

2.3.4.3 Special Functional Units

GPUs also have special functional units (SFU) that implement fast approximate calculation of transcendental operations (e.g., trigonometric functions, exponential and log functions, square root) on single precision data. There are 32 of these units in each multiprocessor of a GTX Titan. They can significantly improve performance when needed.

2.4 Advanced Features of CUDA

In this section, we list some of the newer and more advanced features that have been added to the latest versions of CUDA and GPU hardware, version 5.x and compute capabiilty 3.x respectively at the time of this writing (NVIDIA).

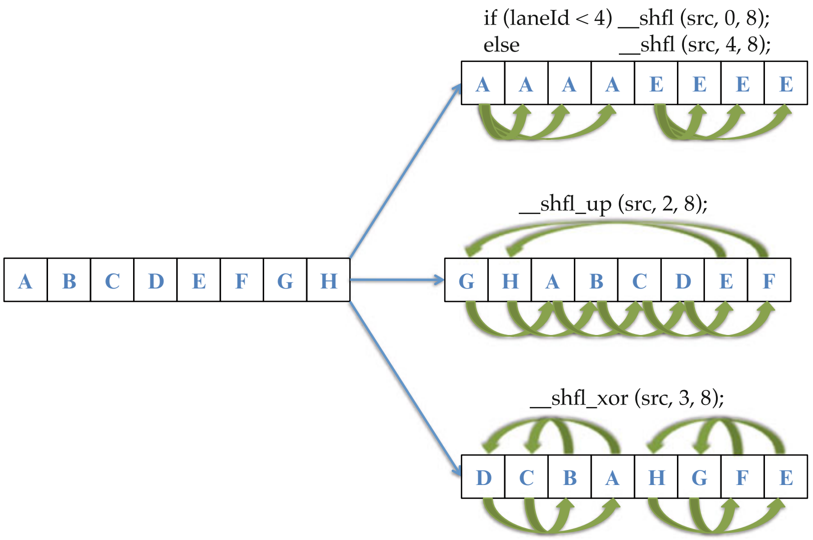


Fig. 2.4 Examples of the shuffle instruction

2.4.1 Shuffle

Shuffle instructions have been added in the latest generation of GPUs to enable data exchange between threads in a warp. The purpose of the shuffle instructions is similar to that of shared memory. However, shuffle instructions have certain performance advantages.

First, sharing data over shared memory requires two steps, a store and a load, whereas a shuffle instruction requires only one, reducing the total number of operations. Secondly, using shuffle instructions will reduce the shared memory footprint for each thread block, which may result in more thread blocks fitting into each multiprocessor and a subsequent increase in performance. On the downside, shuffle instructions are limited to moving 4 byte data types (8 byte data types require 2 shuffle instructions) and cannot be used for inter-warp communication even within the same thread block.

GPUs support three types of shuffle instructions: arbitrarily indexed, shift left/right by *n*, and XOR. Examples of these three types of shuffle instructions are illustrated in Fig. 2.4.

Readers should keep in mind that although in theory each thread can read from any variable of any other thread in the same warp, doing so using conditionals will create divergence, as exemplified in the topmost shuffle instruction in Fig. 2.4. Therefore, users are recommended to read from the same variable and use computable offsets rather than conditionals whenever possible.

2.4.2 Dynamic Parallelism

Traditionally, all CUDA programs were initiated from the host. After each kernel was launched, the host had to wait for it to complete before launching the next kernel, effectively creating a global synchronization between kernels. This can be expensive as well as being redundant.

In the latest generation of GPUs, CUDA programs can initiate work by themselves without involving the host. This feature, called *dynamic parallelism*, allows programmers to create and optimize recursive and data-dependent execution patterns, opening the door to more varieties of parallel algorithms running on GPUs.

Using dynamic parallelism has other benefits. The ability to create and launch new kernels inside another kernel allows the programmer to change the granularity of the computation on the fly depending on the results of the current kernel execution, possibly increasing the accuracy or the performance of the program as a whole. Finally, it also has the benefit that the CPU resource can now be left uninterrupted for other computation, allowing for better resource utilization and heterogeneous computing.

2.4.3 Hyper Q

One potential problem of having so many cores on a single device is utilization. Unless there is enough work, utilizing the GPU to its full potential can often prove to be difficult or impossible. GPUs provide *streams*, where a single stream is a series of dependent kernel executions, to allow concurrent execution of different kernels to better utilize the available resources. However, in older GPUs it suffered from limitations such as false dependencies due to its *single* hardware work queue which limited the concurrency that could be exploited.

With the introduction of Hyper-Q in the latest generation of GPUs, streams can now run concurrently with minimal or no false dependencies. Figure 2.5 shows an example of concurrent streams with and without false dependencies.

2.4.4 Grid Management Unit

The grid management unit (GMU) is a new feature that allows both CPUs and GPU kernels to launch a grid of thread blocks for execution. It uses *multiple* hardware work queues to paralyze different threads of execution to run concurrently on the same GPU, allowing true concurrency with little or no false dependencies. The use of GMU is what allows both dynamic parallelism and Hyper-Q to work on the latest generation of GPUs.

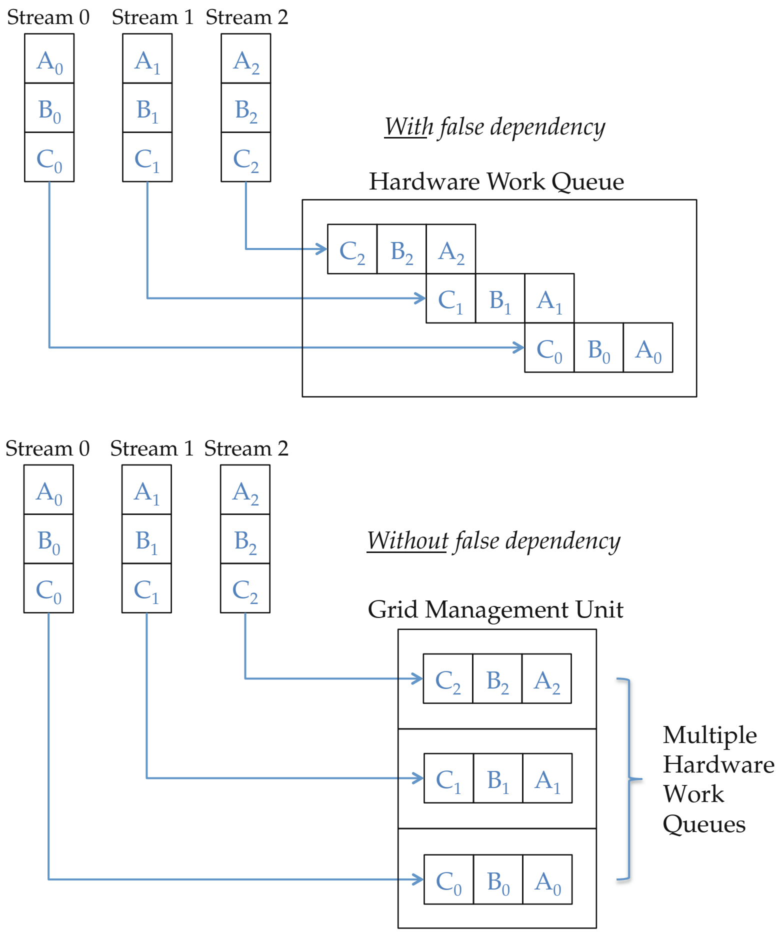


Fig. 2.5 CUDA streams with and without false dependency

2.4.5 GPUDirect

When working with large amounts of data distributed over many GPUs and nodes, inter-GPU or inter-node communication can become a bottleneck for performance. NVIDIA's GPUDirect technology introduces two features that improve performance.

First, GPUDirect allows third-party devices such as Infiniband routers and network interface cards (NIC) to directly access the GPU memory without involving the CPU. This is achieved by having the GPU and the third-party device share the same memory space on the host which removes the need for the CPU to copy the same data from one memory location to another. Secondly, GPUDirect allows point-to-point (PTP) communication between GPUs on the same PCIe bus by communicating directly over the PCIe instead of the host memory. GPUDirect also allows the use of direct memory access (DMA) to communicate, effectively eliminating CPU overheads such as latency and bandwidth bottlenecks.

2.5 Conclusion

In this chapter, we covered the basics of the GPU architecture and its programming model Although this chapter will allow beginners to understand the basic concepts behind CUDA and how it differs from traditional programming, it is far from being a comprehensive guide for GPGPU programming. CUDA and GPU technology continue to evolve even now, requiring constant study and practice in order to effectively utilize the latest GPUs' capabilities and features. Readers are recommended to study the most recent CUDA programming manual and other optimization guides, as well as the latest research papers and various other resources available on the internet in order to have a complete and full understanding of how to use CUDA to leverage the full potential of GPUs.

References

T. Blank. The maspar mp-1 architecture. In *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 20–24, 1990.

W. J. Bouknight, S.A. Denenberg, D.E. McIntyre, J. M. Randall, A.H. Sameh, and D.L. Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4):369–388, 1972.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

W. Daniel Hillis. New computer architectures and their relationship to physics or why computer science is no good. *International Journal of Theoretical Physics*, 21(3–4):255–262, 1982.

M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, 1972.

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 Whitepaper*. NVIDIA, Santa Clara, CA, USA.

NVIDIA. *CUDA Toolkit Documentation*. NVIDIA, Santa Clara, CA, USA, May 2013.

S. F. Reddaway. Dap–a distributed array processor. In *Proceedings of the 1st annual symposium on Computer architecture*, ISCA '73, pages 61–65, New York, NY, USA, 1973. ACM.