CHAPTER

Understanding Parallelism with GPUs

2

INTRODUCTION

This chapter aims to provide a broad introduction to the concepts of parallel programming and how these relate to GPU technology. It's primarily aimed at those people reading this text with a background in serial programming, but a lack of familiarity with parallel processing concepts. We look at these concepts in the primary context of GPUs.

TRADITIONAL SERIAL CODE

A significant number of programmers graduated when serial programs dominated the landscape and parallel programming attracted just a handful of enthusiasts. Most people who go to university get a degree related to IT because they are interested in technology. However, they also appreciate they need to have a job or career that pays a reasonable salary. Thus, in specializing, at least some consideration is given to the likely availability of positions after university. With the exception of research or academic posts, the number of commercial roles in parallel programming has always been, at best, small. Most programmers developed applications in a simple serial fashion based broadly on how universities taught them to program, which in turn was driven by market demand.

The landscape of parallel programming is scattered, with many technologies and languages that never quite made it to the mainstream. There was never really the large-scale market need for parallel hardware and, as a consequence, significant numbers of parallel programmers. Every year or two the various CPU vendors would bring out a new processor generation that executed code faster than the previous generation, thereby perpetuating serial code.

Parallel programs by comparison were often linked closely to the hardware. Their goal was to achieve faster performance and often that was at the cost of portability. Feature X was implemented differently, or was not available in the next generation of parallel hardware. Periodically a revolutionary new architecture would appear that required a complete rewrite of all code. If your knowledge as a programmer was centered around processor X, it was valuable in the marketplace only so long as processor X was in use. Therefore, it made a lot more commercial sense to learn to program x86-type architecture than some exotic parallel architecture that would only be around for a few years.

CUDA Programming. http://dx.doi.org/10.1016/B978-0-12-415933-4.00002-8

However, over this time, a couple of standards did evolve that we still have today. The OpenMP standard addresses parallelism within a single node and is designed for shared memory machines that contain multicore processors. It does not have any concept of anything outside a single node or box. Thus, you are limited to problems that fit within a single box in terms of processing power, memory capacity, and storage space. Programming, however, is relatively easy as most of the low-level threading code (otherwise written using Windows threads or POSIX threads) is taken care of for you by OpenMP.

The MPI (Message Passing Interface) standard addresses parallelism between nodes and is aimed at clusters of machines within well-defined networks. It is often used in supercomputer installations where there may be many thousands of individual nodes. Each node holds a small section of the problem. Thus, common resources (CPU, cache, memory, storage, etc.) are multiplied by the number of nodes in the network. The Achilles' heel of any network is the various interconnects, the parts that connect the networked machines together. Internode communication is usually the dominating factor determining the maximum speed in any cluster-based solution.

Both OpenMP and MPI can be used together to exploit parallelism within nodes as well as across a network of machines. However, the APIs and the approaches used are entirely different, meaning they are often not used together. The OpenMP directives allow the programmer to take a high-level view of parallelism via specifying parallel regions. MPI by contrast uses an explicit interprocess communication model making the programmer do a lot more work.

Having invested the time to become familiar with one API, programmers are often loathe to learn another. Thus, problems that fit within one computer are often implemented with OpenMP solutions, whereas really large problems are implemented with cluster-based solutions such as MPI.

CUDA, the GPU programming language we'll explore in this text, can be used in conjunction with both OpenMP and MPI. There is also an OpenMP-like directive version of CUDA (OpenACC) that may be somewhat easier for those familiar with OpenMP to pick up. OpenMP, MPI, and CUDA are increasingly taught at undergraduate and graduate levels in many university computer courses.

However, the first experience most serial programmers had with parallel programming was the introduction of multicore CPUs. These, like the parallel environments before them, were largely ignored by all but a few enthusiasts. The primary use of multicore CPUs was for OS-based parallelism. This is a model based on *task parallelism* that we'll look at a little later.

As it became obvious that technology was marching toward the multicore route, more and more programmers started to take notice of the multicore era. Almost all desktops ship today with either a dual- or quad-core processor. Thus, programmers started using threads to allow the multiple cores on the CPU to be exploited.

A thread is a separate execution flow within a program that may diverge and converge as and when required with the main execution flow. Typically, CPU programs will have no more than twice the number of threads active than the number of physical processor cores. As with single-core processors, typically each OS task is time-sliced, given a small amount of time in turn, to give the illusion of running more tasks than there are physical CPU cores.

However, as the number of threads grows, this becomes more obvious to the end user. In the background the OS is having to context switch (swap in and out a set of registers) every time it needs to switch between tasks. As context switching is an expensive operation, typically

thousands of cycles, CPU applications tend to have a fairly low number of threads compared with GPUs.

SERIAL/PARALLEL PROBLEMS

Threads brought with them many of the issues of parallel programming, such as sharing resources. Typically, this is done with a semaphore, which is simply a lock or token. Whoever has the token can use the resource and everyone else has to wait for the user of the token to release it. As long as there is only a single token, everything works fine.

Problems occur when there are two or more tokens that must be shared by the same threads. In such situations, thread 0 grabs token 0, while thread 1 grabs token 1. Thread 0 now tries to grab token 1, while thread 1 tries to grab token 0. As the tokens are unavailable, both thread 0 and thread 1 sleep until the token becomes available. As neither thread ever releases the one token they already own, all threads wait forever. This is known as a deadlock, and it is something that can and will happen without proper design.

The opposite also happens—sharing of resources by chance. With any sort of locking system, all parties to a resource must behave correctly. That is, they must request the token, wait if necessary, and, only when they have the token, perform the operation. This relies on the programmer to identify shared resources and specifically put in place mechanisms to coordinate updates by multiple threads. However, there are usually several programmers in any given team. If just one of them doesn't follow this convention, or simply does not know this is a shared resource, you may appear to have a working program, but only by chance.

One of the projects I worked on for a large company had exactly this problem. All threads requested a lock, waited, and updated the shared resource. Everything worked fine and the particular code passed quality assurance and all tests. However, in the field occasionally users would report the value of a certain field being reset to 0, seemingly randomly. Random bugs are always terrible to track down, because being able to consistently reproduce a problem is often the starting point of tracking down the error.

An intern who happened to be working for the company actually found the issue. In a completely unrelated section of the code a pointer was not initialized under certain conditions. Due to the way the program ran, some of the time, depending on the thread execution order, the pointer would point to our protected data. The other code would then initialize "its variable" by writing 0 to the pointer, thus eliminating the contents of our "protected" and thread-shared parameter.

This is one of the unfortunate areas of thread-based operations; they operate with a shared memory space. This can be both an advantage in terms of not having to formally exchange data via messages, and a disadvantage in the lack of protection of shared data.

The alternative to threads is processes. These are somewhat heavier in terms of OS load in that both code *and* data contexts must be maintained by the OS. A thread by contrast needs to only maintain a code context (the program/instruction counter plus a set of registers) and shares the same data space. Both threads and processes may be executing entirely different sections of a program at any point in time.

Processes by default operate in an independent memory area. This usually is enough to ensure one process is unable to affect the data of other processes. Thus, the stray pointer issue should result in an

exception for out-of-bounds memory access, or at the very least localize the bug to the particular process. Data consequently has to be transferred by formally passing messages to or from processes.

In many respects the threading model sits well with OpenMP, while the process model sits well with MPI. In terms of GPUs, they map to a hybrid of both approaches. CUDA uses a grid of blocks. This can be thought of as a queue (or a grid) of processes (blocks) with no interprocess communication. Within each block there are many threads which operate cooperatively in batches called *warps*. We will look at this further in the coming chapters.

CONCURRENCY

The first aspect of concurrency is to think about the particular problem, without regard for any implementation, and consider what aspects of it could run in parallel.

If possible, try to think of a formula that represents each output point as some function of the input data. This may be too cumbersome for some algorithms, for example, those that iterate over a large number of steps. For these, consider each step or iteration individually. Can the data points for the step be represented as a transformation of the input dataset? If so, then you simply have a set of kernels (steps) that run in sequence. These can simply be pushed into a queue (or stream) that the hardware will schedule sequentially.

A significant number of problems are known as "embarrassingly parallel," a term that rather underplays what is being achieved. If you can construct a formula where the output data points can be represented without relation to each other—for example, a matrix multiplication—be very happy. These types of problems can be implemented extremely well on GPUs and are easy to code.

If one or more steps of the algorithm can be represented in this way, but maybe one stage cannot, also be very happy. This single stage may turn out to be a bottleneck and may require a little thought, but the rest of the problem will usually be quite easy to code on a GPU.

If the problem requires every data point to know about the value of its surrounding neighbors then the speedup will ultimately be limited. In such cases, throwing more processors at the problem works up to a point. At this point the computation slows down due to the processors (or threads) spending more time sharing data than doing any useful work. The point at which you hit this will depend largely on the amount and cost of the communication overhead.

CUDA is ideal for an embarrassingly parallel problem, where little or no interthread or interblock communication is required. It supports interthread communication with explicit primitives using on-chip resources. Interblock communication is, however, only supported by invoking multiple kernels in series, communicating between kernel runs using off-chip global memory. It can also be performed in a somewhat restricted way through atomic operations to or from global memory.

CUDA splits problems into grids of blocks, each containing multiple threads. The blocks may run in any order. Only a subset of the blocks will ever execute at any one point in time. A block must execute from start to completion and may be run on one of *N* SMs (symmetrical multiprocessors). Blocks are allocated from the grid of blocks to any SM that has free slots. Initially this is done on a round-robin basis so each SM gets an equal distribution of blocks. For most kernels, the number of blocks needs to be in the order of eight or more times the number of physical SMs on the GPU.

To use a military analogy, we have an army (a grid) of soldiers (threads). The army is split into a number of units (blocks), each commanded by a lieutenant. The unit is split into squads of 32 soldiers (a warp), each commanded by a sergeant (See Figure 2.1).

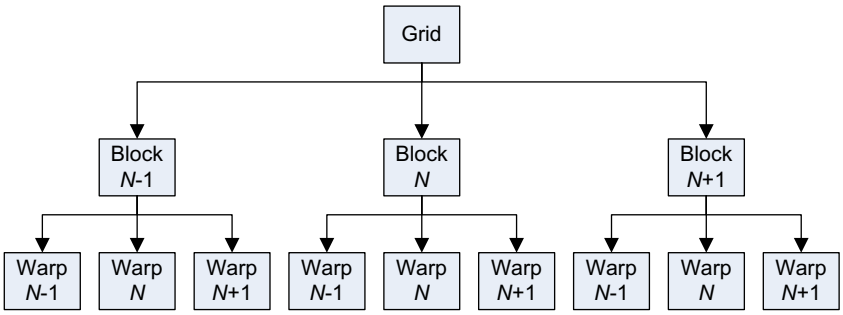

FIGURE 2.1

GPU-based view of threads.

To perform some action, central command (the kernel/host program) must provide some action plus some data. Each soldier (thread) works on his or her individual part of the problem. Threads may from time to time swap data with one another under the coordination of either the sergeant (the warp) or the lieutenant (the block). However, any coordination with other units (blocks) has to be performed by central command (the kernel/host program).

Thus, it's necessary to think of orchestrating thousands of threads in this very hierarchical manner when you think about how a CUDA program will implement concurrency. This may sound quite complex at first. However, for most embarrassingly parallel programs it's just a case of thinking of one thread generating a single output data point. A typical GPU has on the order of 24 K *active* threads. On Fermi GPUs you can define 65,535 $\times$ 65,535 $\times$ 1536 threads in total, 24 K of which are active at any time. This is usually enough to cover most problems within a single node.

Locality

Computing has, over the last decade or so, moved from one limited by computational throughput of the processor, to one where moving the data is the primary limiting factor. When designing a processor in terms of processor real estate, compute units (or ALUs—algorithmic logic units) are cheap. They can run at high speed, and consume little power and physical die space. However, ALUs are of little use without operands. Considerable amounts of power and time are consumed in moving the operands to and from these functional units.

In modern computer designs this is addressed by the use of multilevel caches. Caches work on the principle of either spatial (close in the address space) or temporal (close in time) locality. Thus, data that has been accessed before, will likely be accessed again (temporal locality), and data that is close to the last accessed data will likely be accessed in the future (spatial locality).

Caches work well where the task is repeated many times. Consider for the moment a tradesperson, a plumber with a toolbox (a cache) that can hold four tools. A number of the jobs he will attend are similar, so the same four tools are repeatedly used (a cache hit).

However, a significant number of jobs require additional tools. If the tradesperson does not know in advance what the job will entail, he arrives and starts work. Partway through the job he needs an additional tool. As it's not in his toolbox (L1 cache), he retrieves the item from the van (L2 cache).

Occasionally he needs a special tool or part and must leave the job, drive down to the local hardware store (global memory), fetch the needed item, and return. Neither the tradesperson nor the client knows how long (the latency) this operation will actually take. There may be congestion on the freeway and/or queues at the hardware store (other processes competing for main memory access).

Clearly, this is not a very efficient use of the tradesperson's time. Each time a different tool or part is needed, it needs to be fetched by the tradesperson from either the van or the hardware store. While fetching new tools the tradesperson is not working on the problem at hand.

While this might seem bad, fetching data from a hard drive or SSD (solid-state drive) is akin to ordering an item at the hardware store. In comparative form, data from a hard drive arrives by regular courier several days later. Data from the SSD may arrive by overnight courier, but it's still very slow compared to accessing data in global memory.

In some more modern processor designs we have hardware threads. Some Intel processors feature hyperthreading, with two hardware threads per CPU core. To keep with the same analogy, this is equivalent to the tradesperson having an assistant and starting two jobs. Every time a new tool/part is required, the assistant is sent to fetch the new tool/part and the tradesperson switches to the alternate job. Providing the assistant is able to return with the necessary tool/part before the alternate job also needs an additional tool/part, the tradesperson continues to work.

Although an improvement, this has not solved the latency issue—how long it takes to fetch new tools/parts from the hardware store (global memory). Typical latencies to global memory are in the order of hundreds of clocks. Increasingly, the answer to this problem from traditional processor design has been to increase the size of the cache. In effect, arrive with a bigger van so fewer trips to the hardware store are necessary.

There is, however, an increasing cost to this approach, both in terms of capital outlay for a larger van and the time it takes to search a bigger van for the tool/part. Thus, the approach taken by most designs today is to arrive with a van (L2 cache) and a truck (L3 cache). In the extreme case of the server processors, a huge 18-wheeler is brought in to try to ensure the tradesperson is kept busy for just that little bit longer.

All of this work is necessary because of one fundamental reason. The CPUs are designed to run software where the programmer does not have to care about locality. Locality is an issue, regardless of whether the processor tries to hide it from the programmer or not. The denial that this is an issue is what leads to the huge amount of hardware necessary to deal with memory latency.

The design of GPUs takes a different approach. It places the GPU programmer in charge of dealing with locality and instead of an 18-wheeler truck gives him or her a number of small vans and a very large number of tradespeople.

Thus, in the first instance the programmer must deal with locality. He or she needs to think in advance about what tools/parts (memory locations/data structures) will be needed for a given job. These then need to be collected in a single trip to the hardware store (global memory) and placed in the correct van (on chip memory) for a given job at the outset. Given that this data has been collected, as much work as possible needs to be performed with the data to avoid having to fetch and return it only to fetch it again later for another purpose.

Thus, the continual cycle of work-stall-fetch from global memory, work-stall-fetch from global memory, etc. is broken. We can see the same analogy on a production line. Workers are supplied with baskets of parts to process, rather than each worker individually fetching widgets one at a time from the store manager's desk. To do otherwise is simply a hugely inefficient use of the available workers' time.

This simple process of planning ahead allows the programmer to schedule memory loads into the on-chip memory before they are needed. This works well with both an explicit local memory model such as the GPU's shared memory as well as a CPU-based cache. In the shared memory case you tell the memory management unit to request this data and then go off and perform useful work on another piece of data. In the cache case you can use special cache instructions that allow prefilling of the cache with data you expect the program to use later.

The downside of the cache approach over the shared memory approach is eviction and dirty data. Data in a cache is said to be dirty if it has been written by the program. To free up the space in the cache for new useful data, the dirty data has to be written back to global memory before the cache space can be used again. This means instead of one trip to global memory of an unknown latency, we now have two—one to write the old data and one to get the new data.

The big advantage of the programmer-controlled on-chip memory is that the programmer is in control of when the writes happen. If you are performing some local transformation of the data, there may be no need to write the intermediate transformation back to global memory. With a cache, the cache controller does not know what needs to be written and what can be discarded. Thus, it writes everything, potentially creating lots of useless memory traffic that may in turn cause unnecessary congestion on the memory interface.

Although many do, not every algorithm lends itself to this type of "known in advance" memory pattern that the programmer can optimize for. At the same time, not every programmer wants to deal with locality issues, either initially or sometimes at all. It's a perfectly valid approach to develop a program, prove the concept, and then deal with locality issues.

To facilitate such an approach and to deal with the issues of algorithms that did not have a well-defined data/execution pattern, later generations of GPUs (compute 2.x onward) have both L1 and L2 caches. These can be configured with a preference toward cache or shared memory, allowing the programmer flexibility to configure the hardware for a given problem.

TYPES OF PARALLELISM

Task-based parallelism

If we look at a typical operating system, we see it exploit a type of parallelism called *task parallelism*. The processes are diverse and unrelated. A user might be reading an article on a website while playing music from his or her music library in the background. More than one CPU core can be exploited by running each application on a different core.

In terms of parallel programming, this can be exploited by writing a program as a number of sections that "pipe" (send via messages) the information from one application to another. The Linux pipe operator (the | symbol) does just this, via the operating system. The output of one program, such as `grep`, is the input of the next, such as `sort`. Thus, a set of input files can be easily scanned for a certain set of characters (the `grep` program) and that output set then sorted (the `sort` program). Each program can be scheduled to a separate CPU core.

This pattern of parallelism is known as *pipeline parallelism*. The output on one program provides the input for the next. With a diverse set of components, such as the various text-based tools in Linux, a huge variety of useful functions can be performed by the user. As the programmer cannot know at the outset everyone's needs, by providing components that operate together and can be connected easily, the programmer can target a very wide and diverse user base.

This type of parallelism is very much geared toward *coarse-grained parallelism*. That is, there are a number of powerful processors, each of which can perform a significant chunk of work.

In terms of GPUs we see coarse-grained parallelism only in terms of a GPU card and the execution of GPU kernels. GPUs support the pipeline parallelism pattern in two ways. First, kernels can be pushed into a single stream and separate streams executed concurrently. Second, multiple GPUs can work together directly through either passing data via the host or passing data via messages directly to one another over the PCI-E bus. This latter approach, the peer-to-peer (P2P) mechanism, was introduced in the CUDA 4.x SDK and requires certain OS/hardware/driver-level support.

One of the issues with a pipeline-based pattern is, like any production line, it can only run as fast as the slowest component. Thus, if the pipeline consists of five elements, each of which takes one second, we can produce one output per second. However, if just one of these elements takes two seconds, the throughput of the entire pipeline is reduced to one output every two seconds.

The approach to solving this is twofold. Let's consider the production line analogy for a moment. Fred's station takes two seconds because his task is complex. If we provide Fred with an assistant, Tim, and split his task in half with Tim, we're back to one second per stage. We now have six stages instead of five, but the throughput of the pipeline is now again one widget per second.

You can put up to four GPUs into a desktop PC with some thought and care about the design (see Chapter 11 on designing GPU systems). Thus, if we have a single GPU and it's taking too long to process a particular workflow, we can simply add another one and increase the overall processing power of the node. However, we then have to think about the division of work between the two GPUs. There may not be an easy 50/50 split. If we can only extract a 70/30 split, clearly the maximum benefit will be 7/10 (70%) of the existing runtime. If we could introduce another GPU and then maybe move another task, which occupied say 20% of the time, we'd end up with a 50/30/20 split. Again the speedup compared to one GPU would be 1/2 or 50% of the original time. We're still left with the worst-case time dominating the overall execution time.

The same issue applies to providing a speedup when using a single CPU/GPU combination. If we move 80% of the work off the CPU and onto the GPU, with the GPU computing this in just 10% of the time, what is the speedup? Well the CPU now takes 20% of the original time and the GPU 10% of the original time, but in parallel. Thus, the dominating factor is still the CPU. As the GPU is running in parallel and consumes less time than the CPU fraction, we can discount this time entirely. Thus, the maximum speedup is one divided by the fraction of the program that takes the longest time to execute.

This is known as Amdahl's law and is often quoted as the limiting factor in any speedup. It allows you to know at the outset what the maximum speedup achievable is, without writing a single line of code. Ultimately, you will have serial operations. Even if you move everything onto the GPU, you will still have to use the CPU to load and store data to and from storage devices. You will also have to transfer data to and from the GPU to facilitate input and output (I/O). Thus, maximum theoretical speedup is determined by the fraction of the program that performs the computation/algorithmic part, plus the remaining serial fraction.

Data-based parallelism

Computation power has been greatly increasing over the past couple of decades. We now have teraflop-capable GPUs. However, what has not kept pace with this evolution of compute power is the access time for data. The idea of data-based parallelism is that instead of concentrating on what tasks have to be performed, we look first to the data and how it needs to be transformed.

Task-based parallelism tends to fit more with coarse-grained parallelism approaches. Let's use an example of performing four different transformations on four separate, unrelated, and similarly sized arrays. We have four CPU cores, and a GPU with four SMs. In a task-based decomposition of the problem, we would assign one array to each of the CPU cores or SMs in the GPU. The parallel decomposition of the problem is driven by thinking about the tasks or transformations, not the data.

On the CPU side we could create four threads or processes to achieve this. On the GPU side we would need to use four blocks and pass the address of every array to every block. On the newer Fermi and Kepler devices, we could also create four separate kernels, one to process each array and run it concurrently.

A data-based decomposition would instead split the first array into four blocks and assign one CPU core or one GPU SM to each section of the array. Once completed, the remaining three arrays would be processed in a similar way. In terms of the GPU implementation, this would be four kernels, each of which contained four or more blocks. The parallel decomposition here is driven by thinking about the data first and the transformations second.

As our CPU has only four cores, it makes a lot of sense to decompose the data into four blocks. We could have thread 0 process element 0, thread 1 process element 1, thread 2 process element 2, thread 3 process element 3, and so on. Alternatively, the array could be split into four parts and each thread could start processing its section of the array.

In the first case, thread 0 fetches element 0. As CPUs contain multiple levels of cache, this brings the data into the device. Typically the L3 cache is shared by all cores. Thus, the memory access from the first fetch is distributed to all cores in the CPU. By contrast in the second case, four separate memory fetches are needed and four separate L3 cache lines are utilized. The latter approach is often better where the CPU cores need to write data back to memory. Interleaving the data elements by core means the cache has to coordinate and combine the writes from different cores, which is usually a bad idea.

If the algorithm permits, we can exploit a certain type of data parallelism, the SIMD (single instruction, multiple data) model. This would make use of special SIMD instructions such as MMX, SSE, AVX, etc. present in many x86-based CPUs. Thus, thread 0 could actually fetch multiple adjacent elements and process them with a single SIMD instruction.

If we consider the same problem on the GPU, each array needs to have a separate transformation performed on it. This naturally maps such that one transformation equates to a single GPU kernel (or program). Each SM, unlike a CPU core, is designed to run multiple blocks of data with each block split into multiple threads. Thus, we need a further level of decomposition to use the GPU efficiently. We'd typically allocate, at least initially, a combination of blocks and threads such that a single thread processed a single element of data. As with the CPU, there are benefits from processing multiple elements per thread. This is somewhat limited on GPUs as only load/store/move explicit SIMD primitives are supported, but this in turn allows for enhanced levels of *instruction-level parallelism* (ILP), which we'll see later is actually quite beneficial.

With a Fermi and Kepler GPUs, we have a shared L2 cache that replicates the L3 cache function on the CPU. Thus, as with the CPU, a memory fetch from one thread can be distributed to other threads directly from the cache. On older hardware, there is no cache. However, on GPUs adjacent memory locations are coalesced (combined) together by the hardware, resulting in a single and more efficient memory fetch. We look at this in detail in Chapter 6 on memory.

One important distinction between the caches found in GPUs and CPUs is cache coherency. In a cache-coherent system a write to a memory location needs to be communicated to all levels of cache

in all cores. Thus, all processor cores see the same view of memory at any point in time. This is one of the key factors that limits the number of cores in a processor. Communication becomes increasingly more expensive in terms of time as the processor core count increases. The worst case in a cache-coherent system is where each core writes adjacent memory locations as each write forces a global update to every core's cache.

A non cache-coherent system by comparison does not automatically update the other core's caches. It relies on the programmer to write the output of each processor core to separate areas/addresses. This supports the view of a program where a single core is responsible for a single or small set of outputs. CPUs follow the cache-coherent approach whereas the GPU does not and thus is able to scale to a far larger number of cores (SMs) per device.

Let's assume for simplicity that we implement a kernel as four blocks. Thus, we have four kernels on the GPU and four processes or threads on the CPU. The CPU may support mechanisms such as hyperthreading to enable processing of additional threads/processes due to a stall event, a cache miss, for example. Thus, we could increase this number to eight and we might see an increase in performance. However, at some point, sometimes even at less than the number of cores, the CPU hits a point where there are just too many threads.

At this point the memory bandwidth becomes flooded and cache utilization drops off, resulting in less performance, not more.

On the GPU side, four blocks is nowhere near enough to satisfy four SMs. Each SM can actually schedule up to eight blocks (16 on Kepler). Thus, we'd need $8 \times 4 = 32$ blocks to load the four SMs correctly. As we have four independent operations, we can launch four simultaneous kernels on Fermi hardware via the streams feature (see Chapter 8 on using multiple GPUs). Consequently, we can launch 16 blocks in total and work on the four arrays in parallel. As with the CPU, however, it would be more efficient to work on one array at a time as this would likely result in better cache utilization. Thus, on the GPU we need to ensure we always have enough blocks (typically a minimum of 8 to 16 times the number of SMs on the GPU device).

FLYNN'S TAXONOMY

We mentioned the term SIMD earlier. This classification comes from Flynn's taxonomy, a classification of different computer architectures. The various types are as follows:

- SIMD—single instruction, multiple data
- MIMD—multiple instructions, multiple data
- SISD—single instruction, single data
- MISD—multiple instructions, single data

The standard serial programming most people will be familiar with follows the SISD model. That is, there is a single instruction stream working on a single data item at any one point in time. This equates to a single-core CPU able to perform one task at a time. Of course it's quite possible to provide the illusion of being able to perform more than a single task by simply switching between tasks very quickly, so-called time-slicing.

MIMD systems are what we see today in dual- or quad-core desktop machines. They have a work pool of threads/processes that the OS will allocate to one of *N* CPU cores. Each thread/process has an

independent stream of instructions, and thus the hardware contains all the control logic for decoding many separate instruction streams.

SIMD systems try to simplify this approach, in particular with the data parallelism model. They follow a single instruction stream at any one point in time. Thus, they require a single set of logic inside the device to decode and execute the instruction stream, rather than multiple-instruction decode paths. By removing this silicon real estate from the device, they can be smaller, cheaper, consume less power, and run at higher clock rates than their MIMD cousins.

Many algorithms make use of a small number of data points in one way or another. The data points can often be arranged as a SIMD instruction. Thus, all data points may have some fixed offset added, followed by a multiplication, a gain factor for example. This can be easily implemented as SIMD instructions. In effect, you are programming "for this range of data, perform this operation" instead of "for this data point, perform this operation." As the data operation or transformation is constant for all elements in the range, it can be fetched and decoded from the program memory only once. As the range is defined and contiguous, the data can be loaded en masse from the memory, rather than one word at a time.

However, algorithms where one element has transformation A applied while another element has transformation B applied, and all others have transformation C applied, are difficult to implement using SIMD. The exception is where this algorithm is hard-coded into the hardware because it's very common. Such examples include AES (Advanced Encryption Standard) and H.264 (a video compression standard).

The GPU takes a slightly different approach to SIMD. It implements a model NVIDIA calls SIMT (single instruction, multiple thread). In this model the instruction side of the SIMD instruction is not a fixed function as it is within the CPU hardware. The programmer instead defines, through a kernel, what each thread will do. Thus, the kernel will read the data uniformly and the kernel code will execute transformation A, B, or C as necessary. In practice, what happens is that A, B, and C are executed in sequence by repeating the instruction stream and masking out the nonparticipating threads. However, conceptually this is a much easier model to work with than one that only supports SIMD.

SOME COMMON PARALLEL PATTERNS

A number of parallel problems can be thought of as patterns. We see patterns in many software programs, although not everyone is aware of them. Thinking in terms of patterns allows us to broadly deconstruct or abstract a problem, and therefore more easily think about how to solve it.

Loop-based patterns

Almost anyone who has done any programming is familiar with loops. They vary primarily in terms of entry and exit conditions (`for`, `do...while`, `while`), and whether they create dependencies between loop iterations or not.

A loop-based iteration dependency is where one iteration of the loop depends on one or more previous iterations. We want to remove these if at all possible as they make implementing parallel algorithms more difficult. If in fact this can't be done, the loop is typically broken into a number of blocks that are executed in parallel. The result from block 0 is then retrospectively applied to block 1, then to block 2, and so on. There is an example later in this text where we adopt just such an approach when handling the prefix-sum algorithm.

Loop-based iteration is one of the easiest patterns to parallelize. With inter-loop dependencies removed, it's then simply a matter of deciding how to split, or partition, the work between the available processors. This should be done with a view to minimizing communication between processors and maximizing the use of on-chip resources (registers and shared memory on a GPU; L1/L2/L3 cache on a CPU). Communication overhead typically scales badly and is often the bottleneck in poorly designed systems.

The macro-level decomposition should be based on the number of logical processing units available. For the CPU, this is simply the number of logical hardware threads available. For the GPU, this is the number of SMs multiplied by the maximum load we can give to each SM, 1 to 16 blocks depending on resource usage and GPU model. Notice we use the term *logical* and not physical hardware thread. Some Intel CPUs in particular support more than one logical thread per physical CPU core, so-called hyperthreading. GPUs run multiple blocks on a single SM, so we have to at least multiply the number of SMs by the maximum number of blocks each SM can support.

Using more than one thread per physical device maximizes the throughput of such devices, in terms of giving them something to do while they may be waiting on either a memory fetch or I/O-type operation. Selecting some multiple of this minimum number can also be useful in terms of load balancing on the GPU and allows for improvements when new GPUs are released. This is particularly the case when the partition of the data would generate an uneven workload, where some blocks take much longer than others. In this case, using many times the number of SMs as the basis of the partitioning of the data allows slack SMs to take work from a pool of available blocks.

However, on the CPU side, over subscribing the number of threads tends to lead to poor performance. This is largely due to context switching being performed in software by the OS. Increased contention for the cache and memory bandwidth also contributes significantly should you try to run too many threads. Thus, an existing multicore CPU solution, taken as is, typically has far too large a granularity for a GPU. You will almost always have to repartition the data into many smaller blocks to solve the same problem on the GPU.

When considering loop parallelism and porting an existing serial implementation, be critically aware of hidden dependencies. Look carefully at the loop to ensure one iteration does not calculate a value used later. Be wary of loops that count down as opposed to the standard zero to max value construct, which is the most common type of loop found. Why did the original programmer count backwards? It is likely this may be because there is some dependency in the loop and parallelizing it without understanding the dependencies will likely break it.

We also have to consider loops where we have an inner loop and one or more outer loops. How should these be parallelized? On a CPU the approach would be to parallelize only the outer loop as you have only a limited number of threads. This works well, but as before it depends on there being no loop iteration dependencies.

On the GPU the inner loop, provided it is small, is typically implemented by threads within a single block. As the loop iterations are grouped, adjacent threads usually access adjacent memory locations. This often allows us to exploit locality, something very important in CUDA programming. Any outer loop(s) are then implemented as blocks of the threads. These are concepts we cover in detail in Chapter 5.

Consider also that most loops can be flattened, thus reducing an inner and outer loop to a single loop. Think about an image processing algorithm that iterates along the *X* pixel axis in the inner loop and the *Y* pixel axis in the outer loop. It's possible to flatten this loop by considering all pixels as

a single-dimensional array and iterating over pixels as opposed to image coordinates. This requires a little more thought on the programming side, but it may be useful if one or more loops contain a very small number of iterations. Such small loops present considerable loop overhead compared to the work done per iteration. They are, thus, typically not efficient.

Fork/join pattern

The fork/join pattern is a common pattern in serial programming where there are synchronization points and only certain aspects of the program are parallel. The serial code runs and at some point hits a section where the work can be distributed to *P* processors in some manner. It then "forks" or spawns *N* threads/processes that perform the calculation in parallel. These then execute independently and finally converge or join once *all* the calculations are complete. This is typically the approach found in OpenMP, where you define a parallel region with pragma statements. The code then splits into *N* threads and later converges to a single thread again.

In Figure 2.2, we see a queue of data items. As we have three processing elements (e.g., CPU cores), these are split into three queues of data, one per processing element. Each is processed independently and then written to the appropriate place in the destination queue.

The fork/join pattern is typically implemented with static partitioning of the data. That is, the serial code will launch *N* threads and divide the dataset equally between the *N* threads. If each packet of data takes the same time to process, then this works well. However, as the overall time to execute is the time of the slowest thread, giving one thread too much work means it becomes the single factor determining the total time.

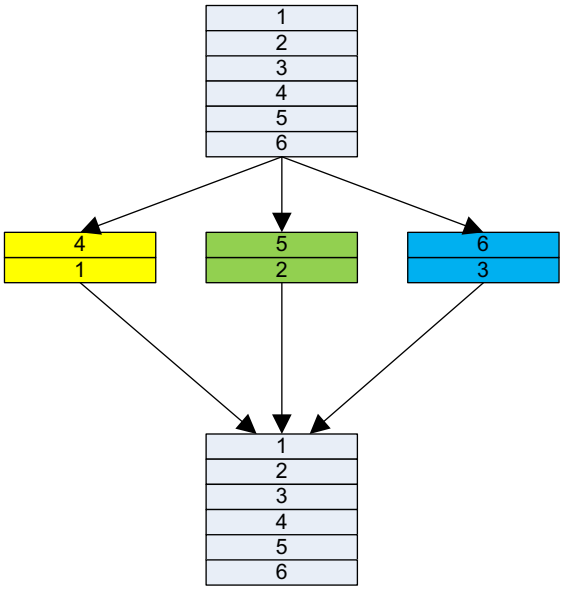

FIGURE 2.2

A queue of data processed by *N* threads.

Systems such as OpenMP also have dynamic scheduling allocation, which mirrors the approach taken by GPUs. Here a thread pool is created (a block pool for GPUs) and only once one task is completed is more work allocated. Thus, if 1 task takes 10x time and 20 tasks take just 1x time each, they are allocated only to free cores. With a dual-core CPU, core 1 gets the big 10x task and five of the smaller 1x tasks. Core 2 gets 15 of the smaller 1x tasks, and therefore both CPU core 1 and 2 complete around the same time.

In this particular example, we've chosen to fork three threads, yet there are six data items in the queue. Why not fork six threads? The reality is that in most problems there can actually be millions of data items and attempting to fork a million threads will cause almost all OSs to fail in one way or another.

Typically an OS will apply a "fair" scheduling policy. Thus, each of the million threads would need to be processed in turn by one of perhaps four available processor cores. Each thread also requires its own memory space. In Windows a thread can come with a 1 MB stack allocation, meaning we'd rapidly run out of memory prior to being able to fork enough threads.

Therefore on CPUs, typically programmers and many multithreaded libraries will use the number of logical processor threads available as the number of processes to fork. As CPU threads are typically also expensive to create and destroy, and also to limit maximum utilization, often a thread pool of workers is used who then fetch work from a queue of possible tasks.

On GPUs we have the opposite problem, in that we in fact need thousands or tens of thousands of threads. We have exactly the thread pool concept we find on more advanced CPU schedulers, except it's more like a block pool than a thread pool. The GPU has an upper limit on the number of *concurrent* blocks it can execute. Each block contains a number of threads. Both the number of threads per block and the overall number of concurrently running blocks vary by GPU generation.

The fork/join pattern is often used when there is an unknown amount of concurrency in a problem. Traversing a tree structure or a path exploration type algorithm may spawn (fork) additional threads when it encounters another node or path. When the path has been fully explored these threads may then join back into the pool of threads or simply complete to be respawned later.

This pattern is not natively supported on a GPU, as it uses a fixed number of blocks/threads at kernel launch time. Additional blocks cannot be launched by the kernel, only the host program. Thus, such algorithms on the GPU side are typically implemented as a series of GPU kernel launches, each of which needs to generate the next state. An alternative is to coordinate or signal the host and have it launch additional, concurrent kernels. Neither solution works particularly well, as GPUs are designed for a static amount of concurrency. Kepler introduces a concept, dynamic parallelism, which addresses this issue. See chapter 12 for more information on this.

Within a block of threads on a GPU there are a number of methods to communication between threads and to coordinate a certain amount of problem growth or varying levels of concurrency within a kernel. For example, if you have an 8×8 matrix you may have many places where just 64 threads are active. However, there may be others where 256 threads can be used. You can launch 256 threads and leave most of them idle until such time as needed. Such idle threads occupy resources and may limit the overall throughput, but do not consume any execution time on the GPU whilst idle. This allows the use of shared memory, fast memory close to the processor, rather than creating a number of distinct steps that need to be synchronized by using the much slower global memory and multiple kernel launches. We look at memory types in Chapter 6.

Finally, the later-generation GPUs support fast atomic operations and synchronization primitives that communicate data between threads in addition to simply synchronizing. We look at some examples of this later in the text.

Tiling/grids

The approach CUDA uses with all problems is to require the programmer to break the problem into smaller parts. Most parallel approaches make use of this concept in one way or another. Even in huge supercomputers problems such climate models must be broken down into hundreds of thousands of blocks, each of which is then allocated to one of the thousands of processing elements present in the machine. This type of parallel decomposition has the huge advantage that it scales really well.

A GPU is in many ways similar to a symmetrical multiprocessor system on a single processor. Each SM is a processor in its own right, capable of running up multiple blocks of threads, typically 256 or 512 threads per block. A number of SMs exist on a single GPU and share a common global memory space. Together as a single GPU they can operate at peak speeds of up to 3 teraflops/s (GTX680).

While peak performance may be impressive, achieving anything like this is not possible without specially crafted programs, as this peak performance does not include things such as memory access, which is somewhat key to any real program. To achieve good performance on any platform requires a good knowledge of the hardware and the understanding of two key concepts—concurrency and locality.

There is concurrency in many problems. It's just that as someone who may come from a serial background, you may not immediately see the concurrency in a problem. The tiling model is thus an easy model to conceptualize. Imagine the problem in two dimensions—a flat arrangement of data—and simply overlay a grid onto the problem space. For a three-dimensional problem imagine the problem as a Rubik's Cube—a set of blocks that map onto the problem space.

CUDA provides the simple two-dimensional grid model. For a significant number of problems this is entirely sufficient. If you have a linear distribution of work within a single block, you have an ideal decomposition into CUDA blocks. As we can assign up to sixteen blocks per SM and we can have up to 16 SMs (32 on some GPUs), any number of blocks of 256 or larger is fine. In practice, we'd like to limit the number of elements within the block to 128, 256, or 512, so this in itself may drive much larger numbers of blocks with a typical dataset.

When considering concurrency, consider also if there is ILP that can be exploited. Conceptually it's easier to think about a single thread being associated with a single output data item. If, however, we can fill the GPU with threads on this basis and there is still more data that could be processed, can we still improve the throughput? The answer is yes, but only through the use of ILP.

ILP exploits the fact that instruction streams can be pipelined within the processor. Thus, it is more efficient to push four add operations into the queue, wait, and then collect them one at a time (push-push-push-push-wait), rather than perform them one at a time (push-wait-push-wait-push-wait-push-wait). For most GPUs, you'll find an ILP level of four operations per thread works best. There are some detailed studies and examples of this in Chapter 9. Thus, if possible we'd like to process *N* elements per thread, but not to the extent that it reduces the overall number of active threads.

Divide and conquer

The divide-and-conquer pattern is also a pattern for breaking down large problems into smaller sections, each of which can be conquered. Taken together these individual computations allow a much larger problem to be solved.

Typically you see divide-and-conquer algorithms used with recursion. Quick sort is a classic example of this. It recursively partitions the data into two sets, those above a pivot point and those below the pivot point. When the partition finally consists of just two items, they are compared and swapped.

Most recursive algorithms can also be represented as an iterative model, which is usually somewhat easier to map onto the GPU as it fits better into the primary tile-based decomposition model of the GPU.

Recursive algorithms are also supported on Fermi-class GPUs, although as with the CPU you have to be aware of the maximum call depth and translate this into stack usage. The available stack can be queried with API call `cudaDeviceGetLimit()`. It can also be set with the API call `cudaDeviceSetLimit()`. Failure to allocate enough stack space, as with CPUs, will result in the program failing. Some debugging tools such as Parallel Nsight and CUDA-GDB can detect such stack overflow issues.

In selecting a recursive algorithm be aware that you are making a tradeoff of development time versus performance. It may be easier to conceptualize and therefore code a recursive algorithm than to try to convert such an approach to an iterative one. However, each recursive call causes any formal parameters to be pushed onto the stack along with any local variables. GPUs and CPUs implement a stack in the same way, simply an area of memory from the global memory space. Although CPUs and the Fermi-class GPUs cache this area, compared to passing values using registers, this is slow. Use iterative solutions where possible as they will generally perform much better and run on a wider range of GPU hardware.

CONCLUSION

We've looked here at a broad overview of some parallel processing concepts and how these are applied to the GPU industry in particular. It's not the purpose of this text to write a volume on parallel processing, for there are entire books devoted to just this subject. We want readers to have some feeling for the issues that parallel programming bring to the table that would not otherwise be thought about in a serial programming environment.

In subsequent chapters we cover some of these concepts in detail in terms of practical examples. We also look at parallel prefix-sum, an algorithm that allows multiple writers of data to share a common array without writing over one another's data. Such algorithms are never needed for serial based programming.

With parallelism comes a certain amount of complexity and the need for a programmer to think and plan ahead to consider the key issues of concurrency and locality. Always keep these two key concepts in mind when designing any software for the GPU.

CHAPTER

CUDA Hardware Overview 3

PC ARCHITECTURE

Let's start by looking at the typical Core 2 architecture we still find today in many PCs and how it impacts our usage of GPU accelerators (Figure 3.1).

Notice that all GPU devices are connected to the processor via the PCI-E bus. In this case we've assumed a PCI-E 2.0 specification bus, which is currently the fastest bus available, giving a transfer rate of 5 GB/s. PCI-E 3.0 has become available at the time of this writing and should significantly improve the bandwidth available.

However, to get data from the processor, we need to go through the Northbridge device over the slow FSB (front-side bus). The FSB can run anything up to 1600 MHz clock rate, although in many designs it is much slower. This is typically only one-third of the clock rate of a fast processor.

Memory is also accessed through the Northbridge, and peripherals through the Northbridge and Southbridge chipset. The Northbridge deals with all the high-speed components like memory, CPU, PCI-E bus connections, etc. The Southbridge chip deals with the slower devices such as hard disks, USB, keyboard, network connections, etc. Of course, it's quite possible to connect a hard-disk controller to the PCI-E connection, and in practice, this is the only true way of getting RAID high-speed data access on such a system.

PCI-E (Peripheral Communications Interconnect Express) is an interesting bus as, unlike its predecessor, PCI (Peripheral Component Interconnect), it's based on guaranteed bandwidth. In the old PCI system each component could use the full bandwidth of the bus, but only one device at a time. Thus, the more cards you added, the less available bandwidth each card would receive. PCI-E solved this problem by the introduction of PCI-E lanes. These are high-speed serial links that can be combined together to form X1, X2, X4, X8, or X16 links. Most GPUs now use at least the PCI-E 2.0, X16 specification, as shown in Figure 3.1. With this setup, we have a 5 GB/s full-duplex bus, meaning we get the same upload and download speed, at the same time. Thus, we can transfer 5 GB/s to the card, while at the same time receiving 5 GB/s from the card. However, this does not mean we can transfer 10 GB/s *to* the card if we're not receiving any data (i.e., the bandwidth is not cumulative).

In a typical supercomputer environment, or even in a desktop application, we are dealing with a large dataset. A supercomputer may deal with petabytes of data. A desktop PC may be dealing with as little as a several GB high-definition video. In both cases, there is considerable data to fetch from the attached peripherals. A single 100 MB/s hard disk will load 6 GB of data in one minute. At this rate it takes over two and a half hours to read the entire contents of a standard 1 TB disk.

CUDA Programming. http://dx.doi.org/10.1016/B978-0-12-415933-4.00003-X

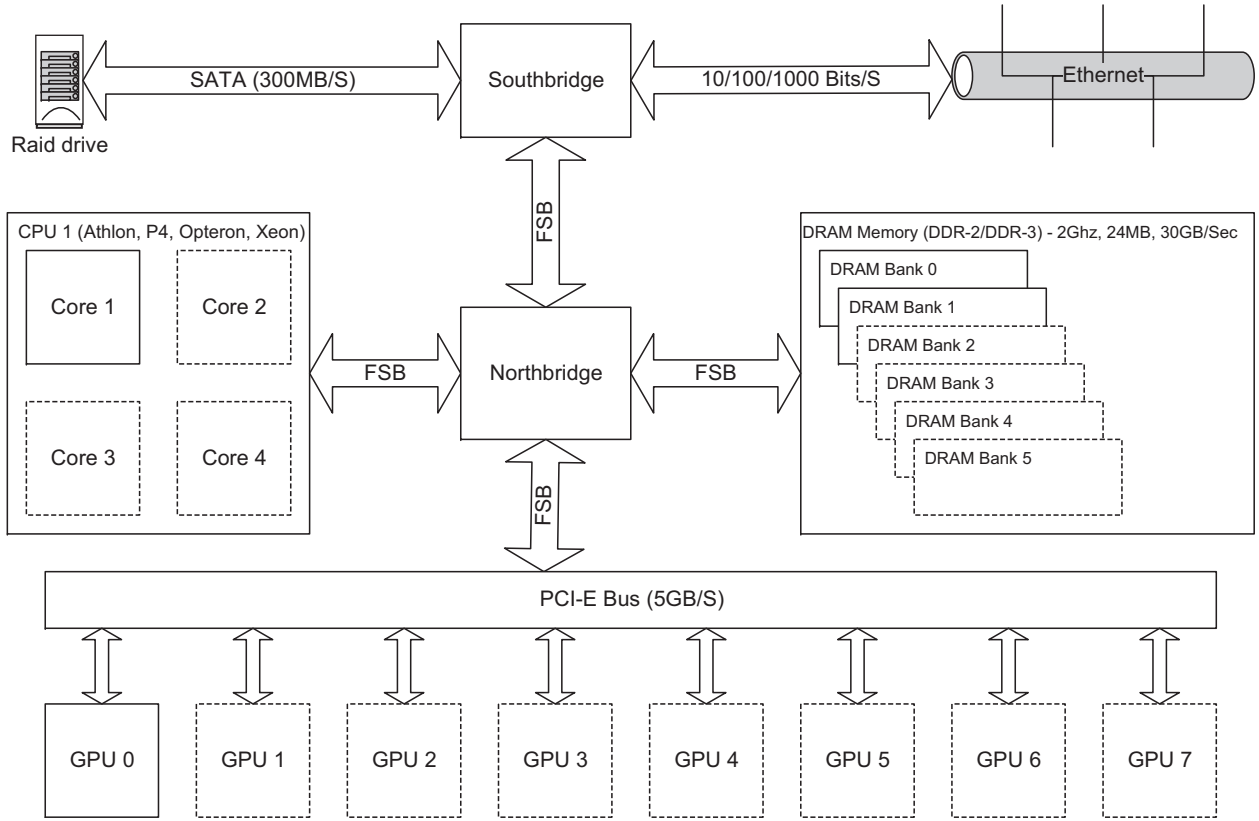

FIGURE 3.1

Typical Core 2 series layout.

If using MPI (Message Passing Interface), commonly used in clusters, the latency for this arrangement can be considerable if the Ethernet connections are attached to the Southbridge instead of the PCI-E bus. Consequently, dedicated high-speed interconnects like InfiniBand or 10 Gigabit Ethernet cards are often used on the PCI-E bus. This removes slots otherwise available for GPUs. Previously, as there was no direct GPU MPI interface, all communications in such a system are routed over the PCI-E bus to the CPU and back again. The GPU-Direct technology, available in the CUDA 4.0 SDK, solved this issue and it's now possible for certain InfiniBand cards to talk directly to the GPU without having to go through the CPU first. This update to the SDK also allows direct GPU to GPU communication.

We saw a number of major changes with the advent of the Nehalem architecture. The main change was to replace the Northbridge and the Southbridge chipset with the X58 chipset. The Nehalem architecture brought us QPI (Quick Path Interconnect), which was actually a huge advance over the FSB (Front Side Bus) approach and is similar to AMD's HyperTransport. QPI is a high-speed interconnect that can be used to talk to other devices or CPUs. In a typical Nehalem system it will connect to the memory subsystem, and through an X58 chipset, the PCI-E subsystem (Figure 3.2). The QPI runs at either 4.8 GT/s or 6.4 GT/s in the Extreme/Xeon processor versions.

With the X58 and 1366 processor socket, a total of 36 PCI-E lanes are available, which means up to two cards are supported at X16, or four cards at X8. Prior to the introduction of the LGA2011 socket, this provided the best bandwidth solution for a GPU machine to date.

The X58 design is also available in a lesser P55 chipset where you get only 16 lanes. This means one GPU card at X16, or two cards at X8.

From the I7/X58 chipset design, Intel moved onto the Sandybridge design, shown in Figure 3.3. One of the most noticeable improvements was the support for the SATA-3 standard, which supports 600 MB/s transfer rates. This, combined with SSDs, allows for considerable input/output (I/O) performance with loading and saving data.

The other major advance with the Sandybridge design was the introduction of the AVX (Advanced Vector Extensions) instruction set, also supported by AMD processors. AVX allows for vector instructions that provide up to four double-precision (256 bit/32 byte) wide vector operations. It's a very interesting development and something that can be used to considerably speed up compute-bound applications on the CPU.

Notice, however, the big downside of socket 1155 Sandybridge design: It supports only 16 PCI-E lanes, limiting the PCI-E bandwidth to 16 GB/s theoretical, 10 GB/s actual bandwidth. Intel has gone down the route of integrating more and more into the CPU with their desktop processors. Only the socket 2011 Sandybridge-E, the server offering, has a reasonable number of PCI-E lanes (40).

So how does AMD compare with the Intel designs? Unlike Intel, which has gradually moved away from large numbers of PCI-E lanes, in all but their server line, AMD have remained fairly constant. Their FX chipset, provides for either two X16 devices or four X8 PCI-E devices. The AMD3+ socket paired with the 990FX chipset makes for a good workhorse, as it provides SATA 6 GB/s ports paired with up to four X16 PCI-E slots (usually running at X8 speed).

One major difference between Intel and AMD is the price point for the number of cores. If you count only real processor cores and ignore logical (hyperthreaded) ones, for the same price point, you typically get more cores on the AMD device. However, the cores on the Intel device tend to perform better. Therefore, it depends a lot on the number of GPUs you need to support and the level of loading of the given cores.

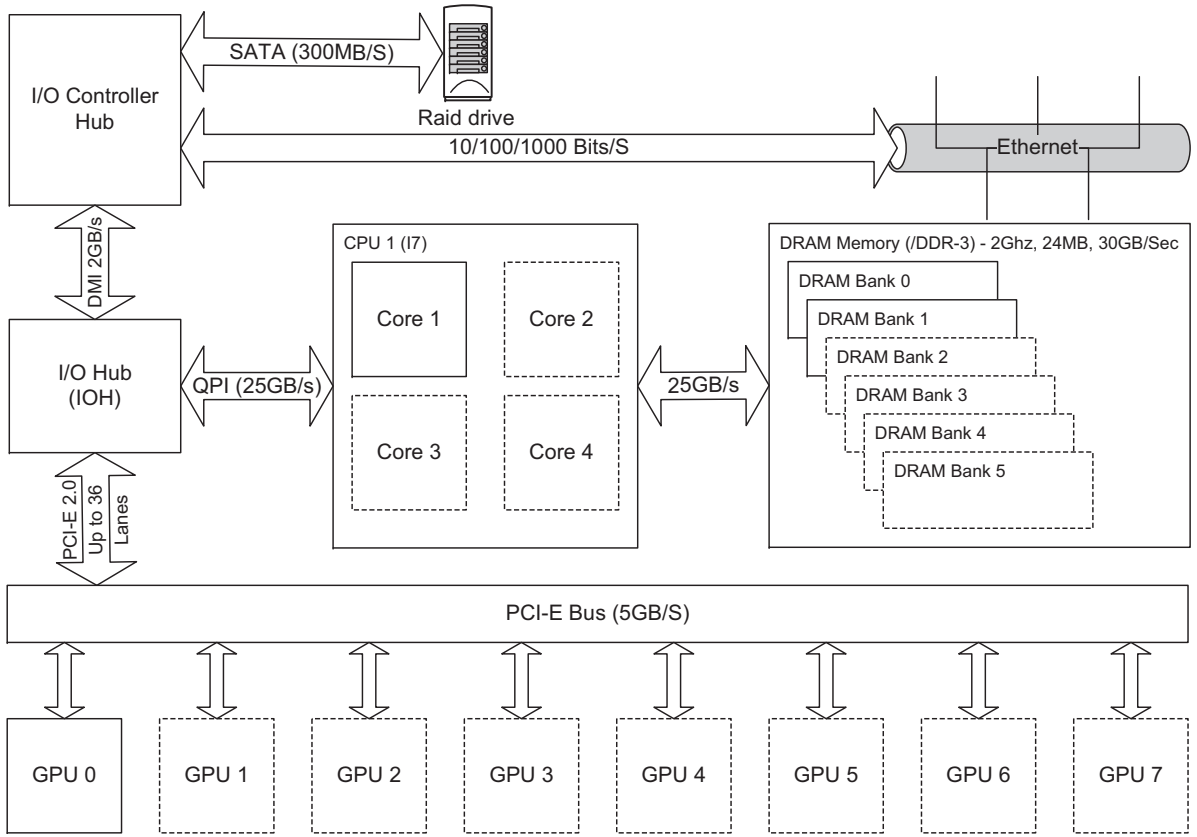

FIGURE 3.2

Nehalem/X58 system.

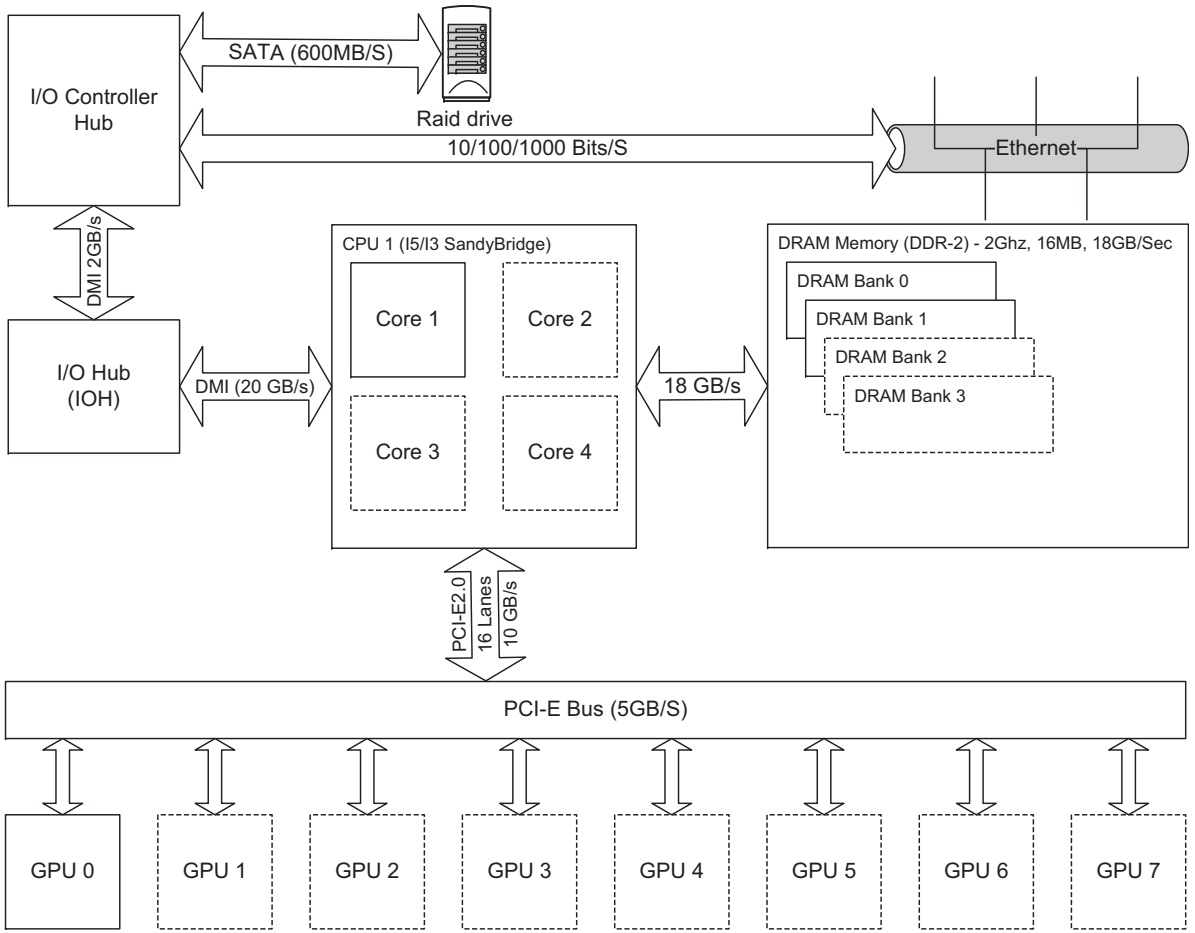

FIGURE 3.3

Sandybridge design.

As with the Intel design, you see similar levels of bandwidth around the system, with the exception of bandwidth to main memory. Intel uses triple or quad channel memory on their top-end systems and dual-channel memory on the lower-end systems. AMD uses only dual-channel memory, leading to significantly less CPU host-memory bandwidth being available (Figure 3.4).

One significant advantage of the AMD chipsets over the Intel ones is the support for up to six SATA (Serial ATA) 6 GB/s ports. If you consider that the slowest component in any system usually limits the overall throughput, this is something that needs some consideration. However, SATA3 can very quickly overload the bandwidth of Southbridge when using multiple SSDs (solid state drives). A PCI-E bus solution may be a better one, but it obviously requires additional costs.

GPU HARDWARE

GPU hardware is radically different than CPU hardware. Figure 3.5 shows how a multi-GPU system looks conceptually from the other side of the PCI-E bus.

Notice the GPU hardware consists of a number of key blocks:

- Memory (global, constant, shared)
- Streaming multiprocessors (SMs)
- Streaming processors (SPs)

The main thing to notice here is that a GPU is really an array of SMs, each of which has *N* cores (8 in G80 and GT200, 32–48 in Fermi, 8 plus in Kepler; see Figure 3.6). This is the key aspect that allows scaling of the processor. A GPU device consists of one or more SMs. Add more SMs to the device and you make the GPU able to process more tasks at the same time, or the same task quicker, if you have enough parallelism in the task.

Like CPUs, if the programmer writes code that limits the processor usage to *N* cores, let's say dual-core, when the CPU manufacturers bring out a quad-core device, the user sees no benefit. This is exactly what happened in the transition from dual- to quad-core CPUs, and lots of software then had to be rewritten to take advantage of the additional cores. NVIDIA hardware will increase in performance by growing a combination of the number of SMs and number of cores per SM. When designing software, be aware that the next generation may double the number of either.

Now let's take a closer look at the SMs themselves. There are number of key components making up each SM, however, not all are shown here for reasons of simplicity. The most significant part is that there are multiple SPs in each SM. There are 8 SPs shown here; in Fermi this grows to 32–48 SPs and in Kepler to 192. There is no reason to think the next hardware revision will not continue to increase the number of SPs/SMs.

Each SM has access to something called a register file, which is much like a chunk of memory that runs at the same speed as the SP units, so there is effectively zero wait time on this memory. The size of this memory varies from generation to generation. It is used for storing the registers in use within the threads running on an SP. There is also a shared memory block accessible only to the individual SM; this can be used as a program-managed cache. Unlike a CPU cache, there is no hardware evicting cache data behind your back—it's entirely under programmer control.

Each SM has a separate bus into the texture memory, constant memory, and global memory spaces. Texture memory is a special view onto the global memory, which is useful for data where

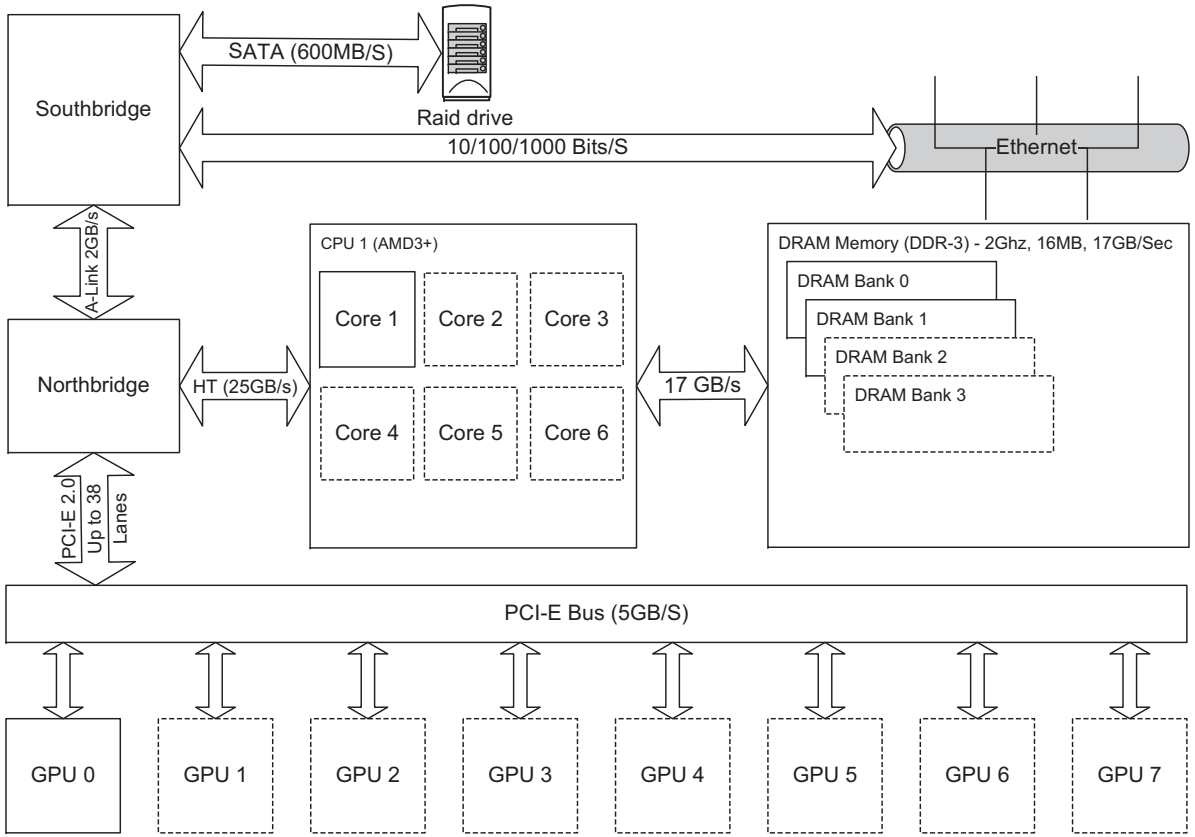

FIGURE 3.4

AMD.

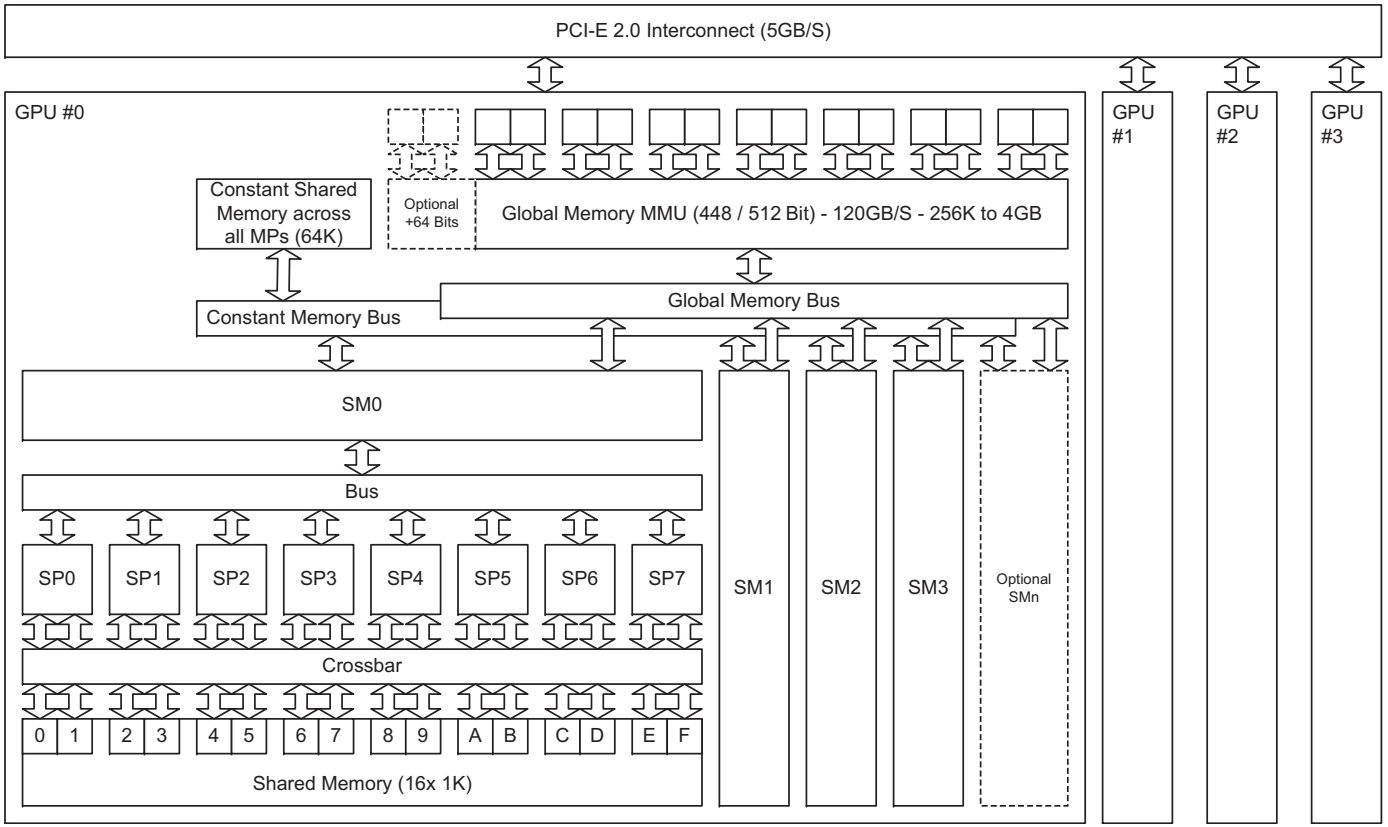

FIGURE 3.5

Block diagram of a GPU (G80/GT200) card.

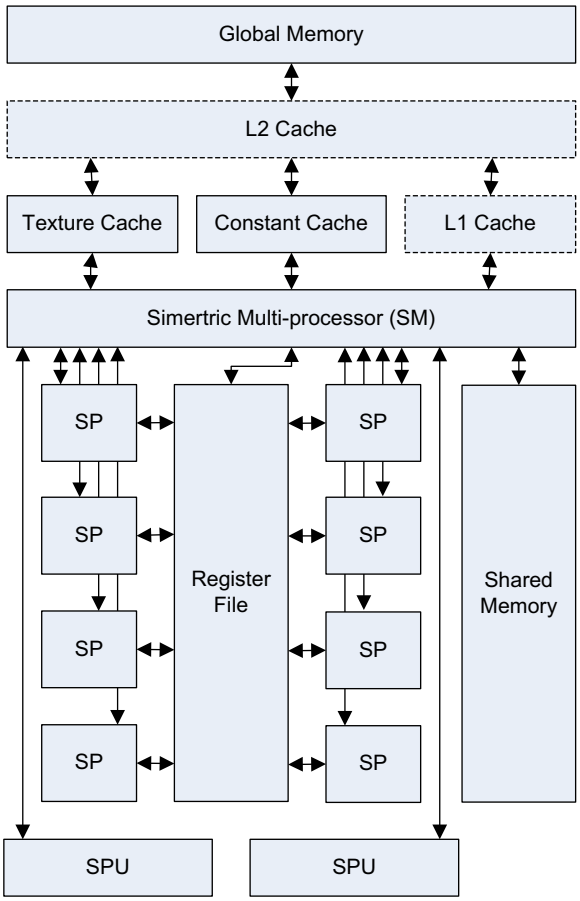

FIGURE 3.6

Inside an SM.

there is interpolation, for example, with 2D or 3D lookup tables. It has a special feature of hardware-based interpolation. Constant memory is used for read-only data and is cached on all hardware revisions. Like texture memory, constant memory is simply a view into the main global memory.

Global memory is supplied via GDDR (Graphic Double Data Rate) on the graphics card. This is a high-performance version of DDR (Double Data Rate) memory. Memory bus width can be up to 512 bits wide, giving a bandwidth of 5 to 10 times more than found on CPUs, up to 190 GB/s with the Fermi hardware.

Each SM also has two or more special-purpose units (SPUs), which perform special hardware instructions, such as the high-speed 24-bit sin/cosine/exponent operations. Double-precision units are also present on GT200 and Fermi hardware.

CPUS AND GPUS

Now that you have some idea what the GPU hardware looks like, you might say that this is all very interesting, but what does it mean for us in terms of programming?

Anyone who has ever worked on a large project will know it's typically partitioned into sections and allocated to specific groups. There may be a specification group, a design group, a coding group, and a testing group. There are absolutely huge benefits to having people in each team who understand completely the job of the person before and after them in the chain of development.

Take, for example, testing. If the designer did not consider testing, he or she would not have included any means to test in software-specific hardware failures. If the test team could only test hardware failure by having the hardware fail, it would have to physically modify hardware to cause such failures. This is hard. It's much easier for the software people to design a flag that inverts the hardware-based error flag in software, thus allowing the failure functionality to be tested easily. Working on the testing team you might see how hard it is to do it any other way, but with a blinkered view of your discipline, you might say that testing is not your role.

Some of the best engineers are those with a view of the processes before and after them. As software people, it's always good to know how the hardware actually works. For serial code execution, it may be interesting to know how things work, but usually not essential. The vast majority of developers have never taken a computer architecture course or read a book on it, which is a great shame. It's one of the main reasons we see such inefficient software written these days. I grew up learning BASIC at age 11, and was programming Z80 assembly language at 14, but it was only during my university days that I really started to understand computer architecture to any great depth.

Working in an embedded field gives you a very hands-on approach to hardware. There is no nice Windows operating system to set up the processor for you. Programming is a very low-level affair. With embedded applications, there are typically millions of boxes shipped. Sloppy code means poor use of the CPU and available memory, which could translate into needing a faster CPU or more memory. An additional 50 cent cost on a million boxes is half a million dollars. This translates into a lot of design and programming hours, so clearly it's more cost effective to write better code than buy additional hardware.

Parallel programming, even today, is very much tied to the hardware. If you just want to write code and don't care about performance, parallel programming is actually quite easy. To really get performance out of the hardware, you need to understand how it works. Most people can drive a car safely and slowly in first gear, but if you are unaware that there are other gears, or do not have the knowledge to engage them, you will never get from point A to point B very quickly. Learning about the hardware is a little like learning to change gear in a car with a manual gearbox—a little tricky at first, but something that comes naturally after awhile. By the same analogy, you can also buy a car with an automatic gearbox, akin to using a library already coded by someone who understands the low-level mechanics of the hardware. However, doing this without understanding the basics of how it works will often lead to a suboptimal implementation.

COMPUTE LEVELS

CUDA supports a number of compute levels. The original G80 series graphics cards shipped with the first version of CUDA. The compute capability is fixed into the hardware. To upgrade to a newer