

6.6 Introduction to graphics processing units

The original justification for adding SIMD instructions to existing architectures was that many microprocessors were connected to graphics displays in PCs and workstations, so an increasing fraction of processing time was used for graphics. As **Moore's Law** increased the number of transistors available to microprocessors, it therefore made sense to improve graphics processing.



A major driving force for improving graphics processing was the computer game industry, both on PCs and in dedicated game consoles such as the Sony PlayStation. The rapidly growing game market encouraged many companies to make increasing investments in developing faster graphics hardware, and this positive feedback loop led graphics processing to improve at a quicker rate than general-purpose processing in mainstream microprocessors.

Given that the graphics and game community had different goals than the microprocessor development community, it evolved its own style of processing and terminology. As the graphics processors increased in power, they earned the name *Graphics Processing Units* or *GPUs* to distinguish themselves from CPUs.

For a few hundred dollars, anyone can buy a GPU today with hundreds of parallel floating-point units, which makes high-performance computing more accessible. The interest in GPU computing blossomed when this potential was combined with a programming language that made GPUs easier to program. Hence, many programmers of scientific and multimedia applications today are pondering whether to use GPUs or CPUs.

(This section concentrates on using GPUs for computing. To see how GPU computing combines with the traditional role of graphics acceleration, see COD Appendix B (Graphics and Computing GPUs).)

Here are some of the key characteristics as to how GPUs vary from CPUs:

- GPUs are accelerators that supplement a CPU, so they do not need to be able to perform all the tasks of a CPU. This role allows them to dedicate all their resources to graphics. It's fine for GPUs to perform some tasks poorly or not at all, given that in a system with both a CPU and a GPU, the CPU can do them if needed.
- The GPU problem sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.

These differences led to different styles of architecture:

- Perhaps the biggest difference is that GPUs do not rely on multilevel caches to overcome the long latency to memory, as do CPUs. Instead, GPUs rely on hardware multithreading (COD Section 6.4 (Hardware multithreading)) to hide the latency to memory. That is, between the time of a memory request and the time that data arrive, the GPU executes hundreds or thousands of threads that are independent of that request.
- The GPU memory is thus oriented toward bandwidth rather than latency. There are even special graphics DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs. In addition, GPU memories have traditionally had smaller main memories than conventional microprocessors. In 2013, GPUs typically have 4 to 6 GiB or less, while CPUs have 32 to 256 GiB. Finally, keep in mind that for general-purpose computation, you must include the time to transfer the data between CPU memory and GPU memory, since the GPU is a coprocessor.
- Given the reliance on many threads to deliver good memory bandwidth, GPUs can accommodate many parallel processors (MIMD) as well as many threads. Hence, each GPU processor is more highly multithreaded than a typical CPU, plus they have more processors.

Hardware/Software Interface

Although GPUs were designed for a narrower set of applications, some programmers wondered if they could specify their applications in a form that would let them tap the high potential performance of GPUs. After tiring of trying to specify their problems using the graphics APIs and languages, they developed C-inspired programming languages to allow them to write programs directly for the GPUs. An example is NVIDIA's CUDA (Compute Unified Device Architecture), which enables the programmer to write C programs to execute on GPUs, albeit with some restrictions. COD Appendix B (Graphics and Computing GPUs) gives examples of CUDA code. (OpenCL is a multi-company initiative to develop a portable programming language that provides many of the benefits of CUDA.)

NVIDIA decided that the unifying theme of all these forms of parallelism is the *CUDA Thread*. Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can gang thousands of CUDA threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism. These threads are blocked together and executed in groups of 32 at a time. A multithreaded processor inside a GPU executes these blocks of threads, and a GPU consists of 8 to 32 of these multithreaded processors.

An introduction to the NVIDIA GPU architecture

We use NVIDIA systems as our example as they are representative of GPU architectures. Specifically, we follow the terminology of the CUDA parallel programming language and use the Fermi architecture as the example.

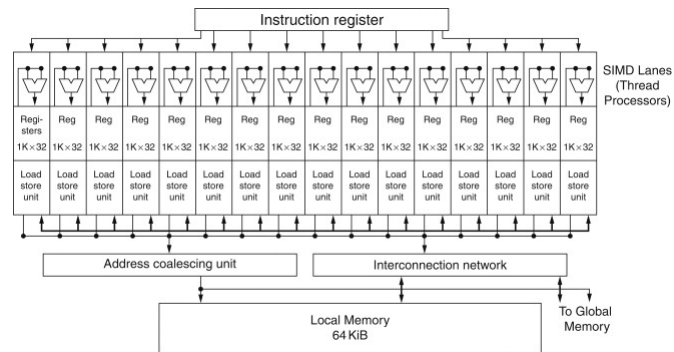
Like vector architectures, GPUs work well only with data-level parallel problems. Both styles have gather-scatter data transfers, and GPU processors have even more registers than do vector processors. Unlike most vector architectures, GPUs also rely on hardware multithreading within a single multithreaded SIMD processor to hide memory latency (see COD Section 6.4 (Hardware multithreading)).

A multithreaded SIMD processor is similar to a vector processor, but the former has many parallel functional units instead of just a few that are deeply pipelined, as does the latter.

As mentioned above, a GPU contains a collection of multithreaded SIMD processors; that is, a GPU is a MIMD composed of multithreaded SIMD processors. For example, NVIDIA has four implementations of the Fermi architecture at different price points with 7, 11, 14, or 15 multithreaded SIMD processors. To provide transparent scalability across models of GPUs with differing number of multithreaded SIMD processors, the Thread Block Scheduler hardware assigns blocks of threads to multithreaded SIMD processors. The figure below shows a simplified block diagram of a multithreaded SIMD processor.

Figure 6.6.1: Simplified block diagram of the datapath of a multithreaded SIMD Processor (COD Figure 6.9).

It has 16 SIMD lanes. The SIMD Thread Scheduler has many independent SIMD threads that it chooses from to run on this processor.



Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*, which we will also call a *SIMD thread*. It is a traditional thread, but it contains exclusively SIMD instructions. These SIMD threads have their own program counters, and they run on a multithreaded SIMD processor. The *SIMD Thread Scheduler* includes a controller that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded SIMD processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor (see COD Section 6.4 (Hardware multithreading)), except that it is scheduling threads of SIMD instructions. Thus, GPU hardware has two levels of hardware schedulers:

1. The *Thread Block Scheduler* that assigns blocks of threads to multithreaded SIMD processors, and
2. The *SIMD Thread Scheduler* *within* a SIMD processor, which schedules when SIMD threads should run.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions would compute 32 of the elements of the computation. Since the thread consists of SIMD instructions, the SIMD processor must have parallel functional units to perform the operation. We call them *SIMD Lanes*, and they are quite similar to the Vector Lanes in COD Section 6.3 (SISD, MIMD, SIMD, SPMD, and vector).

Elaboration

The number of lanes per SIMD processor varies across GPU generations. With Fermi, each 32-wide thread of SIMD instructions is mapped to 16 SIMD lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete. Each thread of SIMD instructions is executed in lock step. Staying with the analogy of a SIMD processor as a vector processor, you could say that it has 16 lanes, and the vector length would be 32. This wide but shallow nature is why we use the term SIMD processor instead of vector processor, as it is more intuitive.

Since by definition the threads of SIMD instructions are independent, the SIMD Thread Scheduler can pick whatever thread of SIMD instructions is ready, and need not stick with the next SIMD instruction in the sequence within a single thread. Thus, using the terminology of COD Section 6.4 (Hardware multithreading), it uses fine-grained multithreading.

To hold these memory elements, a Fermi SIMD processor has an impressive 32,768 32-bit registers. Just like a vector processor, these registers are divided logically across the vector lanes or, in this case, SIMD lanes. Each SIMD thread is limited to no more than 64 registers, so you might think of a SIMD thread as having up to 64 vector registers, with each vector register having 32 elements and each element being 32 bits wide.

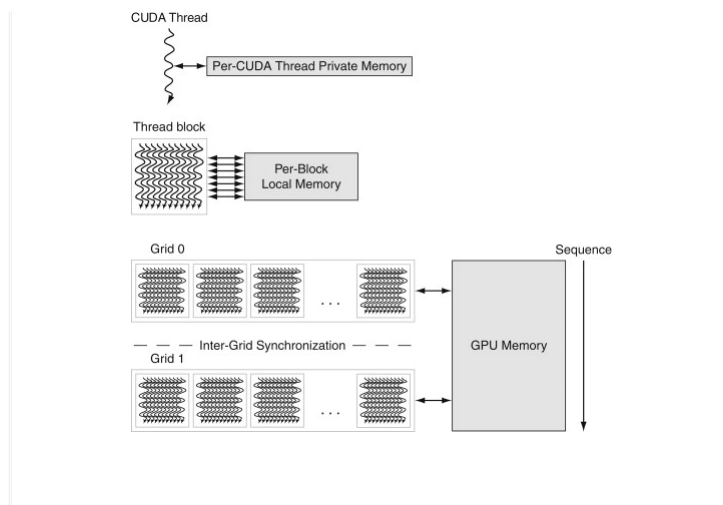
Since Fermi has 16 SIMD lanes, each contains 2048 registers. Each CUDA thread gets one element of each of the vector registers. Note that a CUDA thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by one SIMD lane. Beware that CUDA threads are very different from POSIX threads; you can't make arbitrary system calls or synchronize arbitrarily in a CUDA thread.

NVIDIA GPU memory structures

The figure below shows the memory structures of an NVIDIA GPU. We call the on-chip memory that is local to each multithreaded SIMD processor *Local Memory*. It is shared by the SIMD lanes within a multithreaded SIMD processor, but this memory is not shared between multithreaded SIMD processors. We call the off-chip DRAM shared by the whole GPU and all thread blocks *GPU Memory*.

Figure 6.6.2: GPU Memory structures (COD Figure 6.10).

GPU Memory is shared by the vectorized loops. All threads of SIMD instructions within a thread block share Local Memory.



Rather than rely on large caches to contain the entire working sets of an application, GPUs traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM, since their working sets can be hundreds of megabytes. Thus, they will not fit in the last-level cache of a multicore microprocessor. Given the use of hardware multithreading to hide DRAM latency, the chip area used for caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of the many threads of SIMD instructions.

Elaboration

While hiding memory latency is the underlying philosophy, note that the latest GPUs and vector processors have added caches. For example, the recent Fermi architecture has added caches, but they are thought of as either bandwidth filters to reduce demands on GPU Memory or as accelerators for the few variables whose latency cannot be hidden by multithreading. Local memory for stack frames, function calls, and register spilling is a good match to caches, since latency matters when calling a function. Caches can also save energy, since on-chip cache accesses take much less energy than accesses to multiple, external DRAM chips.

Putting GPUs into perspective

At a high level, multicore computers with SIMD instruction extensions do share similarities with GPUs. The figure below summarizes the similarities and differences. Both are MIMDs whose processors use multiple SIMD lanes, although GPUs have more processors and many more lanes. Both use hardware multithreading to improve processor utilization, although GPUs have hardware support for many more threads. Both use caches, although GPUs use smaller streaming caches and multicore computers use large multilevel caches that try to contain whole working sets completely. Both use a 64-bit address space, although the physical main memory is much smaller in GPUs. While GPUs support memory protection at the page level, they do not yet support demand paging.

Figure 6.6.3: Similarities and differences between multicore with Multimedia SIMD extensions and recent GPUs (COD Figure 6.11).

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GiB to 256 GiB	4 GiB to 6 GiB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Cache coherent	Yes	No

SIMD processors are also similar to vector processors. The multiple SIMD processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors. This view would consider the Fermi GTX 580 as a 16-core machine with hardware support for multithreading, where each core has 16 lanes. The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

GPUs and CPUs do not go back in computer architecture genealogy to a shared ancestor; there is no Missing Link that explains both. As a result of this uncommon heritage, GPUs have not used the terms common in the computer architecture community, which has led to confusion about what GPUs are and how they work. To help resolve the confusion, the figure below (from left to right) lists the more descriptive term used in this section, the closest term from mainstream computing, the official NVIDIA GPU term in case you are interested, and then a short description of the term. This "GPU Rosetta Stone" may help relate this section and ideas to more conventional GPU descriptions, such as those found in COD Appendix B (Graphics and Computing GPUs).

Figure 6.6.4: Quick guide to GPU terms (COD Figure 6.12).

We use the first column for hardware terms. Four groups cluster these 12 terms. From top to bottom: Program Abstractions, Machine Objects, Processing Hardware, and Memory Hardware.

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

While GPUs are moving toward mainstream computing, they can't abandon their responsibility to continue to excel at graphics. Thus, the design of GPUs may make more sense when architects ask, given the hardware invested to do graphics well, how can we supplement it to improve the performance of a wider range of applications?

Having covered two different styles of MIMD that have a shared address space, we next introduce parallel processors where each processor has its own private address space, which makes it considerably easier to build much larger systems. The Internet services that you use every day depend on these large-scale systems.

Elaboration

While the GPU was introduced as having a separate memory from the CPU, both AMD and Intel have announced "fused" products that combine GPUs and CPUs to share a single memory. The challenge will be to maintain the high bandwidth memory in a fused architecture that has been a foundation of GPUs.

PARTICIPATION ACTIVITY 6.6.1: Check yourself: GPUs.

1) GPUs rely on graphics DRAM chips to reduce memory latency and thereby increase performance on graphics applications.

- ☐ True
☐ False

 Provide feedback on this section