NVIDIA's Fermi: The First Complete GPU Computing Architecture

A white paper by Peter N. Glaskowsky

Prepared under contract with NVIDIA Corporation

Copyright © September 2009, Peter N. Glaskowsky

Peter N. Glaskowsky is a consulting computer architect, technology analyst, and professional blogger in Silicon Valley. Glaskowsky was the principal system architect of chip startup Montalvo Systems. Earlier, he was Editor in Chief of the award-winning industry newsletter Microprocessor Report.

Glaskowsky writes the Speeds and Feeds blog for the CNET Blog Network:

http://www.speedsnfeeds.com/

This document is licensed under the Creative Commons Attribution ShareAlike 3.0 License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license identical to this one.

http://creativecommons.org/licenses/by-sa/3.0/

SOME RIGHTS RESERVED

Company and product names may be trademarks of the respective companies with which they are associated.

Executive Summary

After 38 years of rapid progress, conventional microprocessor technology is beginning to see diminishing returns. The pace of improvement in clock speeds and architectural sophistication is slowing, and while single-threaded performance continues to improve, the focus has shifted to multicore designs.

These too are reaching practical limits for personal computing; a quad-core CPU isn't worth twice the price of a dual-core, and chips with even higher core counts aren't likely to be a major driver of value in future PCs.

CPUs will never go away, but GPUs are assuming a more prominent role in PC system architecture. GPUs deliver more cost-effective and energy-efficient performance for applications that need it.

The rapidly growing popularity of GPUs also makes them a natural choice for high-performance computing (HPC). Gaming and other consumer applications create a demand for millions of high-end GPUs each year, and these high sales volumes make it possible for companies like NVIDIA to provide the HPC market with fast, affordable GPU computing products.

NVIDIA's next-generation CUDA architecture (code named Fermi), is the latest and greatest expression of this trend. With many times the performance of any conventional CPU on parallel software, and new features to make it easier for software developers to realize the full potential of the hardware, Fermi-based GPUs will bring supercomputer performance to more users than ever before.

Fermi is the first architecture of any kind to deliver all of the features required for the most demanding HPC applications: unmatched double-precision floating-point performance, IEEE 754-2008 compliance including fused multiply-add operations, ECC protection from the registers to DRAM, a straightforward linear addressing model with caching at all levels, and support for languages including C, C++, FORTRAN, Java, Matlab, and Python.

With these features, plus many other performance and usability enhancements, Fermi is the first complete architecture for GPU computing.

CPU Computing—the Great Tradition

The history of the microprocessor over the last 38 years describes the greatest period of sustained technical progress the world has ever seen. Moore's Law, which describes the rate of this progress, has no equivalent in transportation, agriculture, or mechanical engineering. Think how different the Industrial Revolution would have been 300 years ago if, for example, the strength of structural materials had doubled every 18 months from 1771 to 1809. Never mind steam; the 19th century could have been powered by pea-sized internal-combustion engines compressing hydrogen to produce nuclear fusion.

CPU performance is the product of many related advances:

- Increased transistor density
- Increased transistor performance
- Wider data paths
- Pipelining
- Superscalar execution
- Speculative execution
- Caching
- Chip- and system-level integration

The first thirty years of the microprocessor focused almost exclusively on serial workloads: compilers, managing serial communication links, user-interface code, and so on. More recently, CPUs have evolved to meet the needs of parallel workloads in markets from financial transaction processing to computational fluid dynamics.

CPUs are great things. They're easy to program, because compilers evolved right along with the hardware they run on. Software developers can ignore most of the complexity in modern CPUs; microarchitecture is almost invisible, and compiler magic hides the rest. Multicore chips have the same software architecture as older multiprocessor systems: a simple coherent memory model and a sea of identical computing engines.

But CPU cores continue to be optimized for single-threaded performance at the expense of parallel execution. This fact is most apparent when one considers that integer and floating-point execution units occupy only a tiny fraction of the die area in a modern CPU.

Figure 1 shows the portion of the die area used by ALUs in the Core i7 processor (the chip code-named Bloomfield) based on Intel's Nehalem microarchitecture.

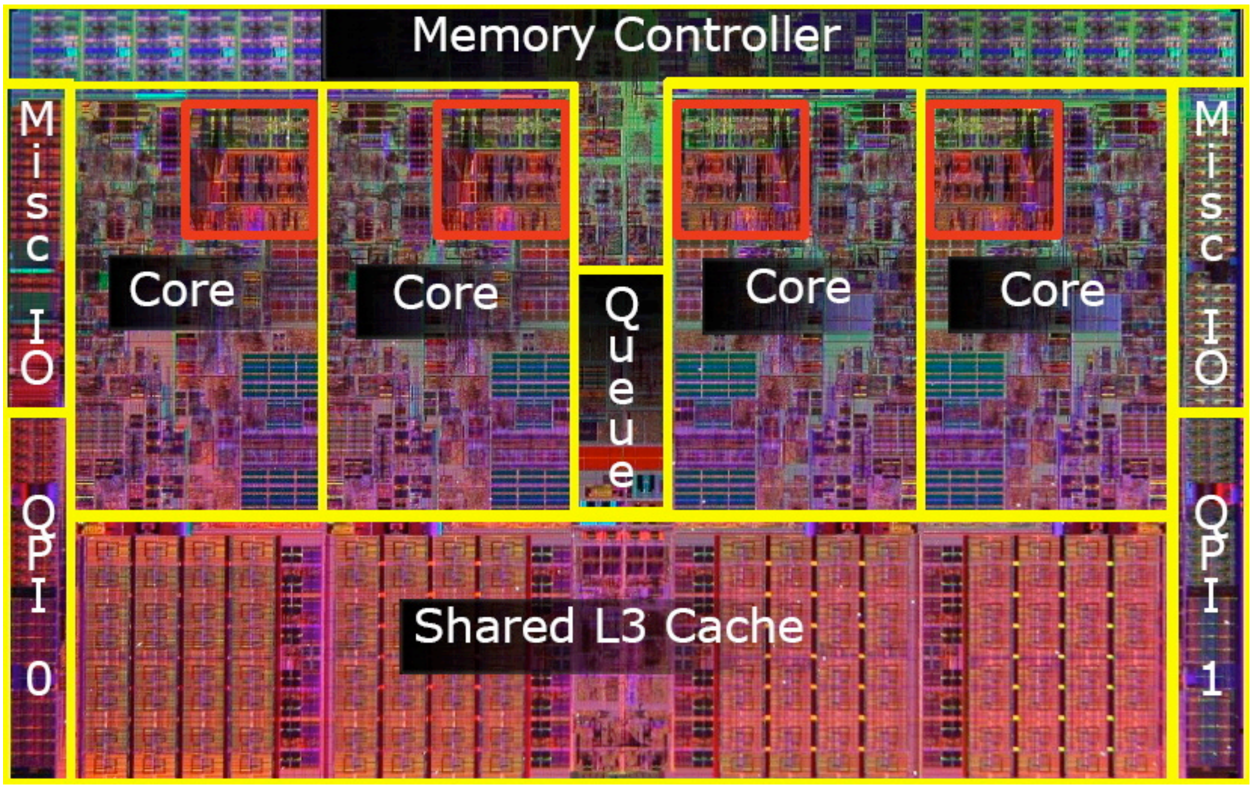


Figure 1. Intel's Core i7 processor (the chip code-named Bloomfield, based on the Nehalem microarchitecture) includes four CPU cores with simultaneous multithreading, 8MB of L3 cache, and on-chip DRAM controllers. Made with 45nm process technology, each chip has 731 million transistors and consumes up to 130W of thermal design power. Red outlines highlight the portion of each core occupied by execution units. (Source: Intel Corporation except red highlighting)

With such a small part of the chip devoted to performing direct calculations, it's no surprise that CPUs are relatively inefficient for high-performance computing applications. Most of the circuitry on a CPU, and therefore most of the heat it generates, is devoted to invisible complexity: those caches, instruction decoders, branch predictors, and other features that are not architecturally visible but which enhance single-threaded performance.

Speculation

At the heart of this focus on single-threaded performance is a concept known as speculation. At a high level, speculation encompasses not only speculative execution (in which instructions begin executing even before it is possible to know their results will be needed), but many other elements of CPU design.

Caches, for example, are fundamentally speculative: storing data in a cache represents a bet that the data will be needed again soon. Caches consume die area and power that could otherwise be used to implement and operate more execution units. Whether the bet pays off depends on the nature of each workload.

Similarly, multiple execution units, out of order processing, and branch prediction also represent speculative optimizations. All of these choices tend to pay off for code with high data locality (where the same data items, or those nearby in memory, are frequently accessed), a mix of different operations, and a high percentage of conditional branches.

But when executing code consisting of many sequential operations of the same type—like scientific workloads—these speculative elements can sit unused, consuming die area and power.

The effect of process technology

The need for CPU designers to maximize single-threaded performance is also behind the use of aggressive process technology to achieve the highest possible clock rates. But this decision also comes with significant costs. Faster transistors run hotter, leak more power even when they aren't switching, and cost more to manufacture.

Companies that make high-end CPUs spend staggering amounts of money on process technology just to improve single-threaded performance. Between them, IBM and Intel have invested tens of billions of dollars on R&D for process technology and transistor design. The results are impressive when measured in gigahertz, but less so from the perspective of GFLOPS per dollar or per watt.

Processor microarchitecture also contributes to performance. Within the PC and server markets, the extremes of microarchitectural optimization are represented by two classes of CPU design: relatively simple dual-issue cores and more complex multi-issue cores.

Dual-issue CPUs

The simplest CPU microarchitecture used in the PC market today is the dual-issue superscalar core. Such designs can execute up to two operations in each clock cycle, sometimes with special "pairing rules" that define which instructions can be executed together. For example, some early dual-issue CPUs could issue two simple integer operations at the same time, or one integer and one floating-point operation, but not two floating-point operations.

Dual-issue cores generally process instructions in program order. They deliver improved performance by exploiting the natural instruction-level parallelism (ILP) in most programs. The amount of available ILP varies from one program to another, but there's almost always enough to take advantage of a second pipeline.

Intel's Atom processor is a good example of a fully evolved dual-issue processor. Like other advanced x86 chips, Atom translates x86 instructions into internal "micro-ops" that are more like the instructions in old RISC (reduced instruction set computing) processors. In Atom, each micro-op can typically perform one ALU operation plus one or more supporting operation such as a memory load or store.

Dual-issue processors like Atom usually occupy the low end of the market where cost-efficiency is paramount. For this reason, Atom has fewer performance-oriented optimizations than more expensive Intel chips. Atom executes in order, with no speculative execution. Much of the new engineering work in Atom went into improving its power efficiency when not operating at full speed.

Atom has six execution pipelines (two for floating point operations, two for integer operations, and two for address calculations; the latter are common in the x86 architecture because instruction operands can specify memory locations). Only two instructions, however, can be issued to these pipelines in a single clock period. This low utilization means that some execution units will always go unused in each cycle.

Like any x86 processor, a large part of Atom is dedicated to instruction caching, decoding (in this case, translating to micro-ops), and a microcode store to implement the more complex x86 instructions. It also supports Atom's two-way simultaneous multithreading (SMT) feature. This circuitry, which Intel calls the "front end cluster," occupies more die area than the chip's floating-point unit.

SMT is basically a way to work around cases that further limit utilization of the execution units. Sometimes a single thread is stalled waiting for data from the cache, or has multiple instructions pending for a single pipeline. In these cases, the second thread may be able to issue an instruction or two. The net performance benefit is usually low, only 10%–20% on some applications, but SMT adds only a few percent to the size of the chip.

As a result, the Atom core is suitable for low-end consumer systems, but provides very low net performance, well below what is available from other Intel processors.

Intel's Larrabee

Larrabee is Intel's code name for a future graphics processing architecture based on the x86 architecture. The first Larrabee chip is said to use dual-issue cores derived from the original Pentium design, but modified to include support for 64-bit x86 operations and a new 512-bit vector-processing unit.

Apart from the vector unit, the Larrabee core is simpler than Atom's. It doesn't support Intel's MMX or SSE extensions, instead relying solely on the new vector unit, which has its own new instructions. The vector unit is wide enough to perform 16 single-precision FP operations per clock, and also provides double-precision FP support at a lower rate.

Several features in Larrabee's vector unit are new to the x86 architecture, including scatter-gather loads and stores (forming a vector from 16 different locations in memory—a convenient feature, though one that must be used judiciously), fused multiply-add, predicated execution, and three-operand floating-point instructions.

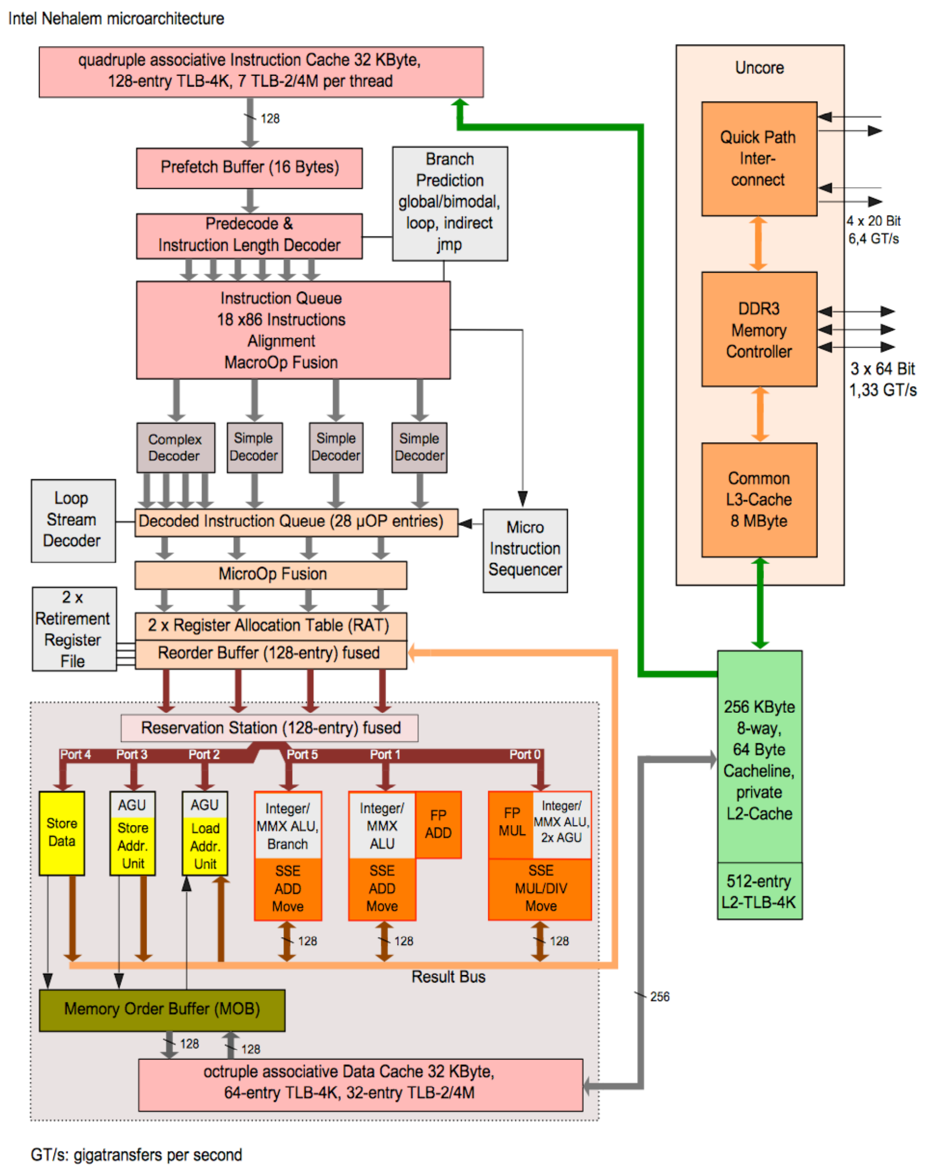
Larrabee also supports four-way multithreading, but not in the same way as Atom. Where Atom can simultaneously execute instructions from two threads (hence the SMT name), Larrabee simply maintains the state of multiple threads to speed the process of switching to a new thread when the current thread stalls.

Larrabee's x86 compatibility reduces its performance and efficiency without delivering much benefit for graphics. As with Atom, a significant (if not huge) part of the Larrabee die area and power budget will be consumed by instruction decoders. As a graphics chip, Larrabee will be impaired by its lack of optimized fixed-function logic for rasterization, interpolating, and alpha blending. Lacking cost-effective performance for 3D games, it will be difficult for Larrabee to achieve the kind of sales volumes and profit margins Intel expects of its major product lines.

Larrabee will be Intel's second attempt to enter the PC graphics-chip market, after the i740 program of 1998, which was commercially unsuccessful but laid the foundation for Intel's later integrated-graphics chipsets. (Intel made an even earlier run at the video controller business with the i750, and before that, the company's i860 RISC processor was used as a graphics accelerator in some workstations.)

Intel’s Nehalem microarchitecture

Nehalem is the most sophisticated microarchitecture in any x86 processor. Its features are like a laundry list of high-performance CPU design: four-wide superscalar, out of order, speculative execution, simultaneous multithreading, multiple branch predictors, on-die power gating, on-die memory controllers, large caches, and multiple interprocessor interconnects. Figure 2 shows the Nehalem microarchitecture.



Intel Nehalem microarchitecture
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU Store Addr. Unit
AGU Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octruple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K
GT/s: gigatransfers per second

Figure 2. The Nehalem core includes multiple x86 instruction decoders, queues, reordering buffers, and six execution pipelines to support speculative out-of-order multithreaded execution. (Source: "File:Intel Nehalem arch.svg." Wikimedia Commons)

Four instruction decoders are provided in each Nehalem core; these run in parallel wherever possible, though only one can decode complex x86 instructions, which are relatively infrequent. The micro-ops generated by the decoders are queued and dispatched out of order through six ports to 12 computational execution units. There is also one load unit and two store units for data and address values.

Nehalem's 128-bit SIMD floating-point units are similar to those found on previous generation Intel processors: one for FMUL and FDIV (floating-point multiply and divide), one for FADD, and one for FP shuffle operations. The "shuffle" unit is used to rearrange data values within the SIMD registers, and does not contribute to the performance of multiply-add intensive algorithms.

The peak single-precision floating-point performance of a four-core Nehalem processor (not counting the shuffle unit) can be calculated as:

```
4 cores * 2 SIMD ops/clock * 4 values/op * clock rate
```

Also, while Nehalem processors provide 32 GB/s of peak DRAM bandwidth—a commendable figure for a PC processor—this figure represents a little less than one byte of DRAM I/O for each three floating-point operations. As a result, many high-performance computing applications will be bottlenecked by DRAM performance before they saturate the chip's floating-point ALUs.

The Xeon W5590 is Intel's high-end quad-core workstation processor based on the Nehalem-EP "Gainestown" chip. The W5590 is priced at $1,600 each when purchased in 1,000-unit quantities (as of August 2009).

At its 3.33 GHz clock, the W5590 delivers a peak single-precision floating-point rate of 106.56 GFLOPS. The W5590 has a 130W thermal design power (TDP) rating, or 1.22 watts/GFLOPS—not including the necessary core-logic chipset.

Nehalem has been optimized for single-threaded performance and clock speed at the expense of sustained throughput. This is a desirable tradeoff for a chip intended to be a market-leading PC desktop and server processor, but it makes Nehalem an expensive, power-hungry choice for high-performance computing.

"The Wall"

The market demands general-purpose processors that deliver high single-threaded performance as well as multi-core throughput for a wide variety of workloads on client, server, and high-performance computing (HPC) systems. This pressure has given us almost three decades of progress toward higher complexity and higher clock rates.

This progress hasn't always been steady. Intel cancelled its "Tejas" processor, which was rumored to have a 40-stage pipeline, and later killed off the entire Pentium 4 "NetBurst" product family because of its relative inefficiency. The Pentium 4 ultimately reached a clock rate of 3.8 GHz in the 2004 "Prescott" model, a speed that Intel has been unable to match since.

In the more recent Core 2 (Conroe/Penryn) and Core i7 (Nehalem) processors, Intel uses increased complexity to deliver substantial performance improvements over the Pentium 4 line, but the pace of these improvements is slowing. Each new generation of process technology requires ever more heroic measures to improve transistor characteristics; each new core microarchitecture must work disproportionately harder to find and exploit instruction-level parallelism (ILP).

As these challenges became more apparent in the 1990s, CPU architects began referring to the "power wall," the "memory wall," and the "ILP wall" as obstacles to the kind of rapid progress seen up until that time. It may be better to think of these issues as mountains rather than walls—mountains that begin as mild slopes and become steeper with each step, making further progress increasingly difficult.

Nevertheless, the inexorable advance of process technology provided CPU designers with more transistors in each generation. By 2005, the competitive pressure to use these additional transistors to deliver improved performance (at the chip level, if not at the core level) drove AMD and Intel to introduce dual-core processors. Since then, the primary focus of PC processor design has been continuing to increase the core count on these chips.

That approach, however, has reached a point of diminishing returns. Dual-core CPUs provide noticeable benefits for most PC users, but are rarely fully utilized except when working with multimedia content or multiple performance-hungry applications. Quad-core CPUs are only a slight improvement, most of the time. By 2010, there will be eight-core CPUs in desktops, but it will likely be difficult to sell most customers on the value of the additional cores. Selling further increases will be even more problematic.

Once the increase in core count stalls, the focus will return to single-threaded performance, but with all the low-hanging fruit long gone, further improvements will be hard to find. In the near term, AMD and Intel are expected to emphasize vector floating-point improvements with the forthcoming Advanced Vector Extensions (AVX). Like SSE, AVX's primary value will be for applications where vectorizable floating-point computations need to be closely coupled with the kind of control-flow code for which modern x86 processors have been optimized.

CPU core design will continue to progress. There will continue to be further improvements in process technology, faster memory interfaces, and wider superscalar cores. But about ten years ago, NVIDIA's processor architects realized that CPUs were no longer the preferred solution for certain problems, and started from a clean sheet of paper to create a better answer.

The History of the GPU

It's one thing to recognize the future potential of a new processing architecture. It's another to build a market before that potential can be achieved. There were attempts to build chip-scale parallel processors in the 1990s, but the limited transistor budgets in those days favored more sophisticated single-core designs.

The real path toward GPU computing began, not with GPUs, but with non-programmable 3G-graphics accelerators. Multi-chip 3D rendering engines were developed by multiple companies starting in the 1980s, but by the mid-1990s it became possible to integrate all the essential elements onto a single chip. From 1994 to 2001, these chips progressed from the simplest pixel-drawing functions to implementing the full 3D pipeline: transforms, lighting, rasterization, texturing, depth testing, and display.

NVIDIA's GeForce 3 in 2001 introduced programmable pixel shading to the consumer market. The programmability of this chip was very limited, but later GeForce products became more flexible and faster, adding separate programmable engines for vertex and geometry shading. This evolution culminated in the GeForce 7800, shown in Figure 3.

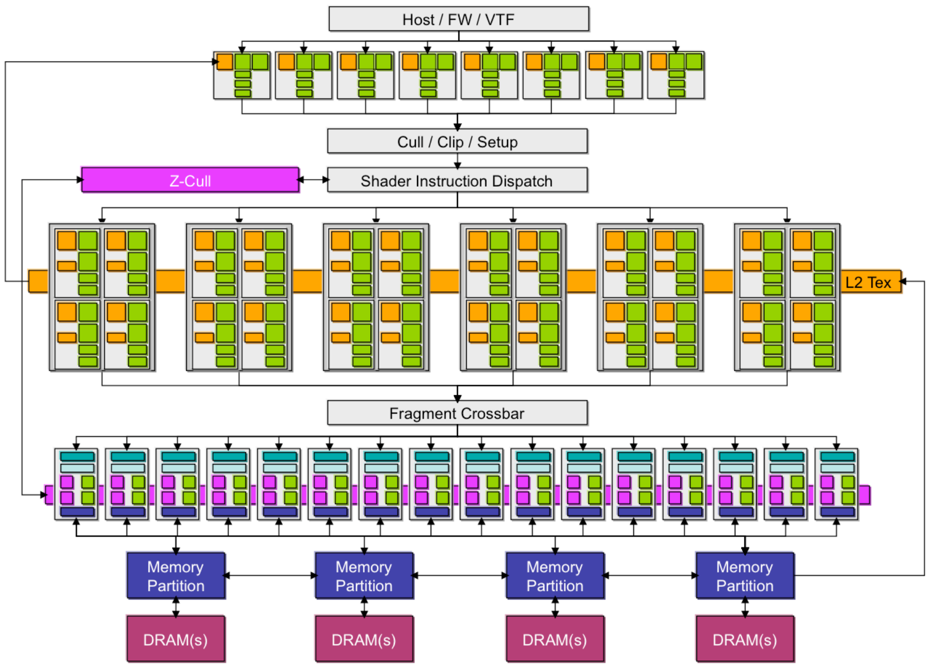


Figure 3. The GeForce 7800 had three kinds of programmable engines for different stages of the 3D pipeline plus several additional stages of configurable and fixed-function logic. (Source: NVIDIA)

So-called general-purpose GPU (GPGPU) programming evolved as a way to perform non-graphics processing on these graphics-optimized architectures, typically by running carefully crafted shader code against data presented as vertex or texture information and retrieving the results from a later stage in the pipeline. Though sometimes awkward, GPGPU programming showed great promise.

Managing three different programmable engines in a single 3D pipeline led to unpredictable bottlenecks; too much effort went into balancing the throughput of each stage. In 2006, NVIDIA introduced the GeForce 8800, as Figure 4 shows. This design featured a "unified shader architecture" with 128 processing elements distributed among eight shader cores. Each shader core could be assigned to any shader task, eliminating the need for stage-by-stage balancing and greatly improving overall performance.

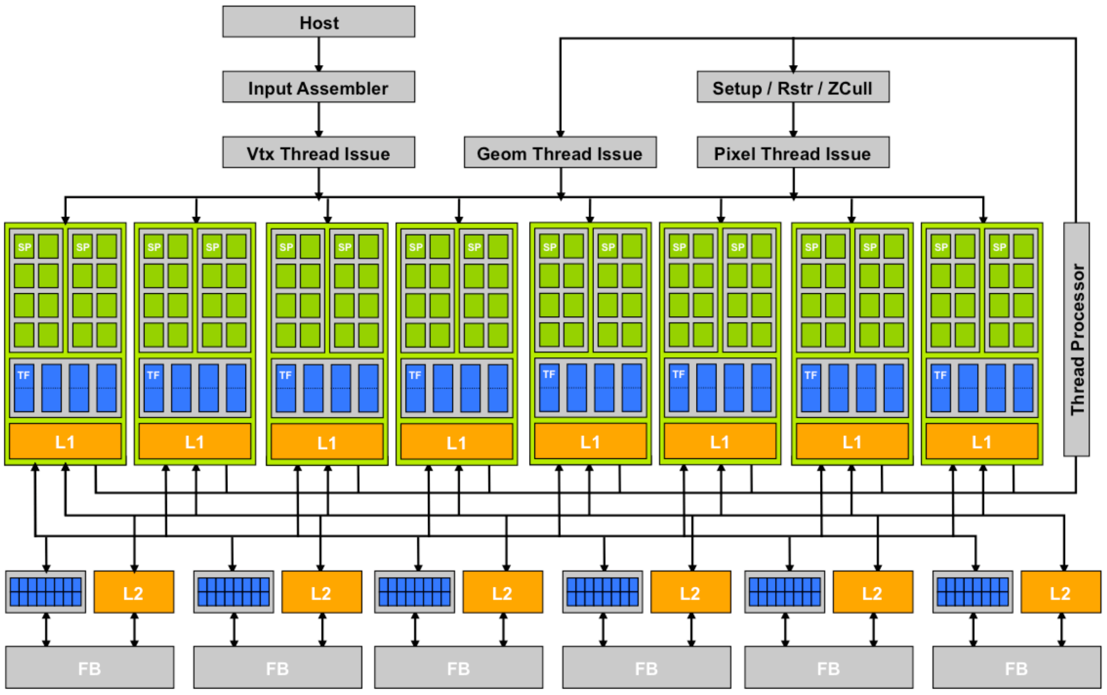


Figure 4. The GeForce 8800 introduced a unified shader architecture with just one kind of programmable processing element that could be used for multiple purposes. Some simple graphics operations still used special-purpose logic. (Source: NVIDIA)

The 8800 also introduced CUDA, the industry's first C-based development environment for GPUs. (CUDA originally stood for "Compute Unified Device Architecture," but the longer name is no longer spelled out.) CUDA delivered an easier and more effective programming model than earlier GPGPU approaches.

To bring the advantages of the 8800 architecture and CUDA to new markets such as HPC, NVIDIA introduced the Tesla product line. Current Tesla products use the more recent GT200 architecture.

The Tesla line begins with PCI Express add-in boards—essentially graphics cards without display outputs—and with drivers optimized for GPU computing instead of 3D rendering. With Tesla, programmers don't have to worry about making tasks look like graphics operations; the GPU can be treated like a many-core processor.

Unlike the early attempts at chip-scale multiprocessing back in the '90s, Tesla was a high-volume hardware platform right from the beginning. This is due in part to NVIDIA's strategy of supporting the CUDA software development platform on the company's GeForce and Quadro products, making it available to a much wider audience of developers. NVIDIA says it has shipped over 100 million CUDA-capable chips.

At the time of this writing, the price for the entry-level Tesla C1060 add-in board is under $1,500 from some Internet mail-order vendors. That's lower than the price of a single Intel Xeon W5590 processor—and the Tesla card has a peak GFLOPS rating more than eight times higher than the Xeon processor.

The Tesla line also includes the S1070, a 1U-height rackmount server that includes four GT200-series GPUs running at a higher speed than that in the C1060 (up to 1.5 GHz core clock vs. 1.3 GHz), so the S1070's peak performance is over 4.6 times higher than a single C1060 card. The S1070 connects to a separate host computer via a PCI Express add-in card.

This widespread availability of high-performance hardware provides a natural draw for software developers. Just as the high-volume x86 architecture attracts more developers than the IA-64 architecture of Intel's Itanium processors, the high sales volumes of GPUs—although driven primarily by the gaming market—makes GPUs more attractive for developers of high-performance computing applications than dedicated supercomputers from companies like Cray, Fujitsu, IBM, NEC, and SGI.

Although GPU computing is only a few years old now, it's likely there are already more programmers with direct GPU computing experience than have ever used a Cray. Academic support for GPU computing is also growing quickly. NVIDIA says over 200 colleges and universities are teaching classes in CUDA programming; the availability of OpenCL (such as in the new "Snow Leopard" version of Apple's Mac OS X) will drive that number even higher.

Introducing Fermi

GPU computing isn't meant to replace CPU computing. Each approach has advantages for certain kinds of software. As explained earlier, CPUs are optimized for applications where most of the work is being done by a limited number of threads, especially where the threads exhibit high data locality, a mix of different operations, and a high percentage of conditional branches.

GPU design aims at the other end of the spectrum: applications with multiple threads that are dominated by longer sequences of computational instructions. Over the last few years, GPUs have become much better at thread handling, data caching, virtual memory management, flow control, and other CPU-like features, but the distinction between computationally intensive software and control-flow intensive software is fundamental.

The state of the art in GPU design is represented by NVIDIA's next-generation CUDA architecture, code named Fermi. Figure 5 shows a high-level block diagram of the first Fermi chip.

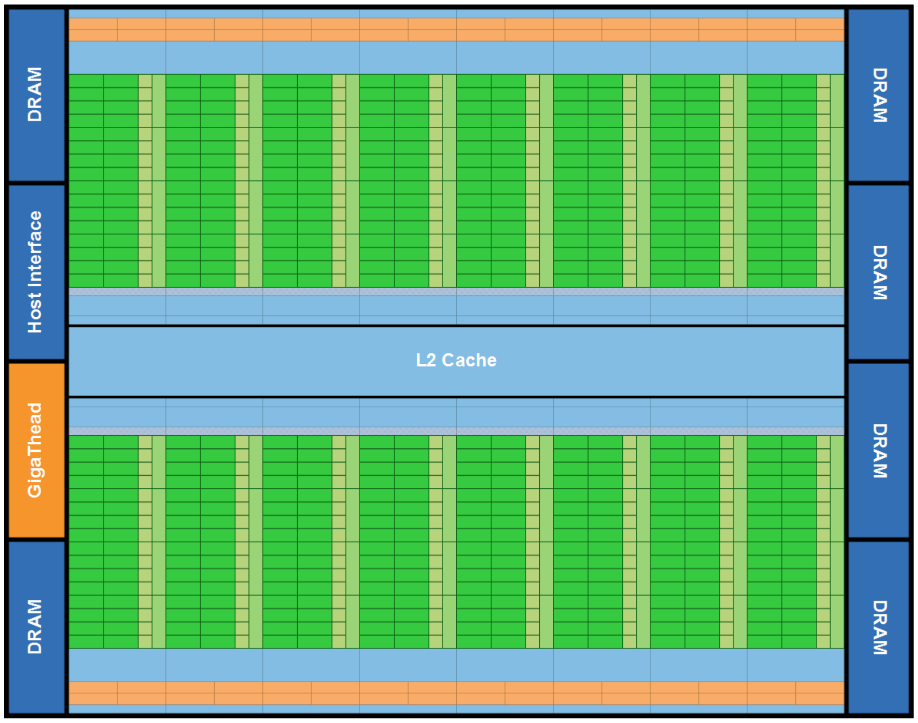


Figure 5. NVIDIA's Fermi GPU architecture consists of multiple streaming multiprocessors (SMs), each consisting of 32 cores, each of which can execute one floating-point or integer instruction per clock. The SMs are supported by a second-level cache, host interface, GigaThread scheduler, and multiple DRAM interfaces. (Source: NVIDIA)

At this level of abstraction, the GPU looks like sea of computational units with only a few support elements—an illustration of the key GPU design goal, which is to maximize floating-point throughput.

Since most of the circuitry within each core is dedicated to computation, rather than speculative features meant to enhance single-threaded performance, most of the die area and power consumed by Fermi goes into the application's actual algorithmic work.

The Programming Model

The complexity of the Fermi architecture is managed by a multi-level programming model that allows software developers to focus on algorithm design rather than the details of how to map the algorithm to the hardware, thus improving productivity. This is a concern that conventional CPUs have yet to address because their structures are simple and regular: a small number of cores presented as logical peers on a virtual bus.

In NVIDIA's CUDA software platform, as well as in the industry-standard OpenCL framework, the computational elements of algorithms are known as *kernels* (a term here adapted from its use in signal processing rather than from operating systems). An application or library function may consist of one or more kernels.

Kernels can be written in the C language (specifically, the ANSI-standard C99 dialect) extended with additional keywords to express parallelism directly rather than through the usual looping constructs.

Once compiled, kernels consist of many *threads* that execute the same program in parallel: one thread is like one iteration of a loop. In an image-processing algorithm, for example, one thread may operate on one pixel, while all the threads together—the kernel—may operate on a whole image.

Multiple threads are grouped into *thread blocks* containing up to 1,536 threads. All of the threads in a thread block will run on a single SM, so within the thread block, threads can cooperate and share memory. Thread blocks can coordinate the use of global shared memory among themselves but may execute in any order, concurrently or sequentially.

Thread blocks are divided into warps of 32 threads. The warp is the fundamental unit of dispatch within a single SM. In Fermi, two warps from different thread blocks (even different kernels) can be issued and executed concurrently, increasing hardware utilization and energy efficiency.

Thread blocks are grouped into *grids*, each of which executes a unique kernel.

Thread blocks and threads each have identifiers (IDs) that specify their relationship to the kernel. These IDs are used within each thread as indexes to their respective input and output data, shared memory locations, and so on.

At any one time, the entire Fermi device is dedicated to a single application. As mentioned above, an application may include multiple kernels. Fermi supports simultaneous execution of multiple kernels from the same application, each kernel being distributed to one or more SMs on the device. This capability avoids the situation where a kernel is only able to use part of the device and the rest goes unused.

Switching from one application to another is about 20 times faster on Fermi (just 25 microseconds) than on previous-generation GPUs. This time is short enough that a Fermi GPU can still maintain high utilization even when running multiple applications, like a mix of compute code and graphics code. Efficient multitasking is important for consumers (e.g., for video games using physics-based effects) and professional users (who often need to run computationally intensive simulations and simultaneously visualize the results).

This switching is managed by the chip-level GigaThread hardware thread scheduler, which manages 1,536 simultaneously active threads for each streaming multiprocessor across 16 kernels.

This centralized scheduler is another point of departure from conventional CPU design. In a multicore or multiprocessor server, no one CPU is "in charge". All tasks, including the operating system's kernel itself, may be run on any available CPU. This approach allows each operating system to follow a different philosophy in kernel design, from large monolithic kernels like Linux's to the microkernel design of QNX and hybrid designs like Windows 7. But the generality of this approach is also its weakness, because it requires complex CPUs to spend time and energy performing functions that could also be handled by much simpler hardware.

With Fermi, the intended applications, principles of stream processing, and the kernel and thread model, were all known in advance so that a more efficient scheduling method could be implemented in the GigaThread engine.

In addition to C-language support, Fermi can also accelerate all the same languages as the GT200, including FORTRAN (with independent solutions from The Portland Group and NOAA, the National Oceanic and Atmospheric Administration), Java, Matlab, and Python. Supported software platforms include NVIDIA's own CUDA development environment, the OpenCL standard managed by the Khronos Group, and Microsoft's Direct Compute API.

The Portland Group (PGI) supports two ways to use GPUs to accelerate FORTRAN programs: the PGI Accelerator programming model in which regions of code within a FORTRAN program can be offloaded to a GPU, and CUDA FORTRAN, which allows the programmer direct control over the operation of attached GPUs including managing local and shared memory, thread synchronization, and so on. NOAA provides a language translator that converts FORTRAN code into CUDA C.

Fermi brings an important new capability to the market with new instruction-level support for C++, including instructions for C++ virtual functions, function pointers, dynamic object allocation, and the C++ exception handling operations "try" and "catch". The popularity of the C++ language, previously unsupported on GPUs, will make GPU computing more widely available than ever.

The Streaming Multiprocessor

Fermi's streaming multiprocessors, shown in Figure 6, comprise 32 cores, each of which can perform floating-point and integer operations, along with 16 load-store units for memory operations, four special-function units, and 64K of local SRAM split between cache and local memory.

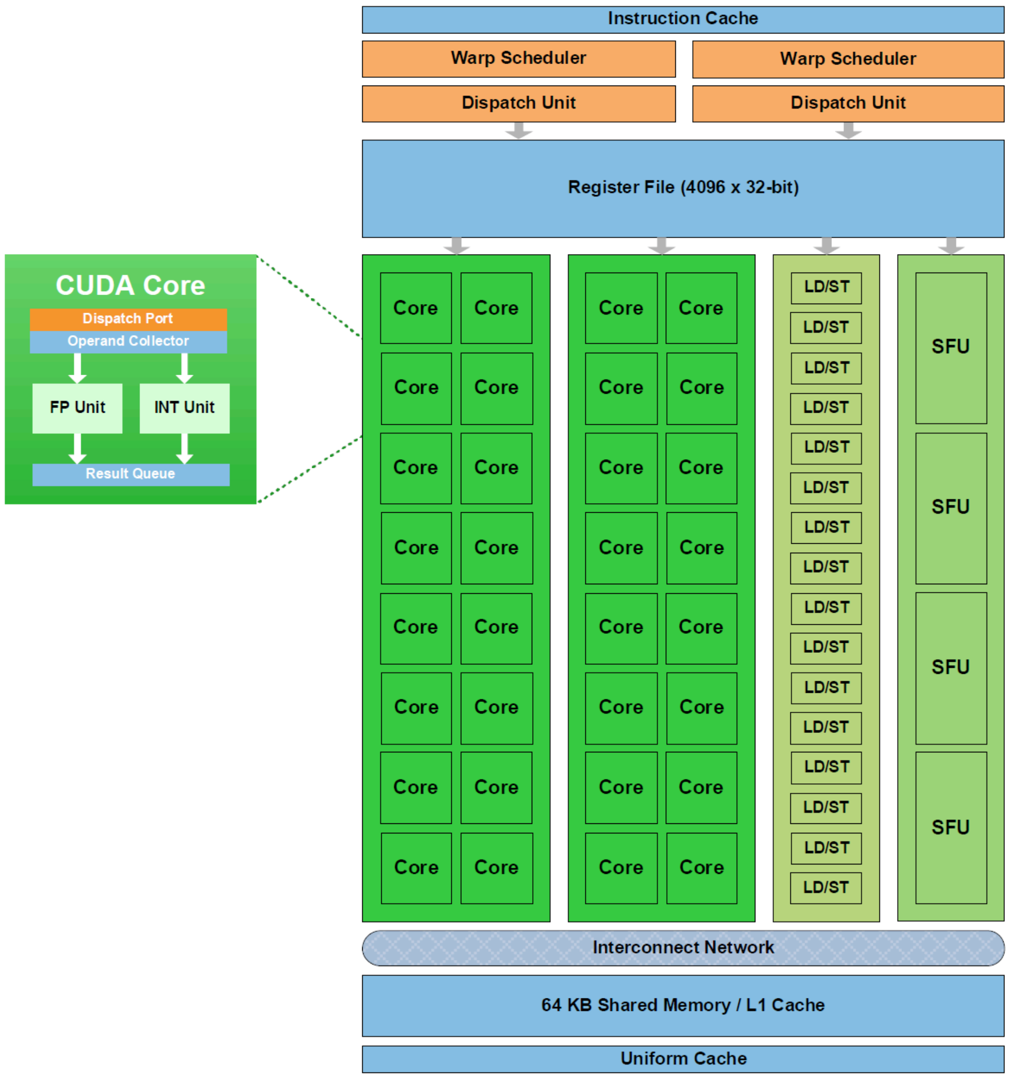


Figure 6. Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 4K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units. (Source: NVIDIA)

Floating-point operations follow the IEEE 754-2008 floating-point standard. Each core can perform one single-precision fused multiply-add operation in each clock period and one double-precision FMA in two clock periods. At the chip level, Fermi performs more than 8× as many double-precision operations per clock than the previous GT200 generation, where double-precision processing was handled by a dedicated unit per SM with much lower throughput.

IEEE floating-point compliance includes all four rounding modes, and subnormal numbers (numbers closer to zero than a normalized format can represent) are handled correctly by the Fermi hardware rather than being flushed to zero or requiring additional processing in a software exception handler.

Fermi's support for fused multiply-add (FMA) also follows the IEEE 754-2008 standard, improving the accuracy of the commonly used multiply-add sequence by not rounding off the intermediate result, as otherwise happens between the multiply and add operations. In Fermi, this intermediate result carries a full 106-bit mantissa; in fact, 161 bits of precision are maintained during the add operation to handle worst-case denormalized numbers before the final double-precision result is computed. The GT200 supported FMA for double-precision operations only; Fermi brings the benefits of FMA to single-precision as well.

FMA support also increases the accuracy and performance of other mathematical operations such as division and square root, and more complex functions such as extended-precision arithmetic, interval arithmetic, and linear algebra.

The integer ALU supports the usual mathematical and logical operations, including multiplication, on both 32-bit and 64-bit values.

Memory operations are handled by a set of 16 load-store units in each SM. The load/store instructions can now refer to memory in terms of two-dimensional arrays, providing addresses in terms of x and y values. Data can be converted from one format to another (for example, from integer to floating point or vice-versa) as it passes between DRAM and the core registers at the full rate. These formatting and converting features are further examples of optimizations unique to GPUs—not worthwhile in general-purpose CPUs, but here they will be used sufficiently often to justify their inclusion.

A set of four Special Function Units (SFUs) is also available to handle transcendental and other special operations such as `sin`, `cos`, `exp`, and `rcp` (reciprocal). Four of these operations can be issued per cycle in each SM.

Within the SM, cores are divided into two *execution blocks* of 16 cores each. Along with the group of 16 load-store units and the four SFUs, there are four execution blocks per SM. In each cycle, a total of 32 instructions can be dispatched from one or two warps to these blocks. It takes two cycles for the 32 instructions in each warp to execute on the cores or load/store units. A warp of 32 special-function instructions is issued in a single cycle but takes eight cycles to complete on the four SFUs. Figure 7 shows a sequence of instructions being distributed among the available execution blocks.

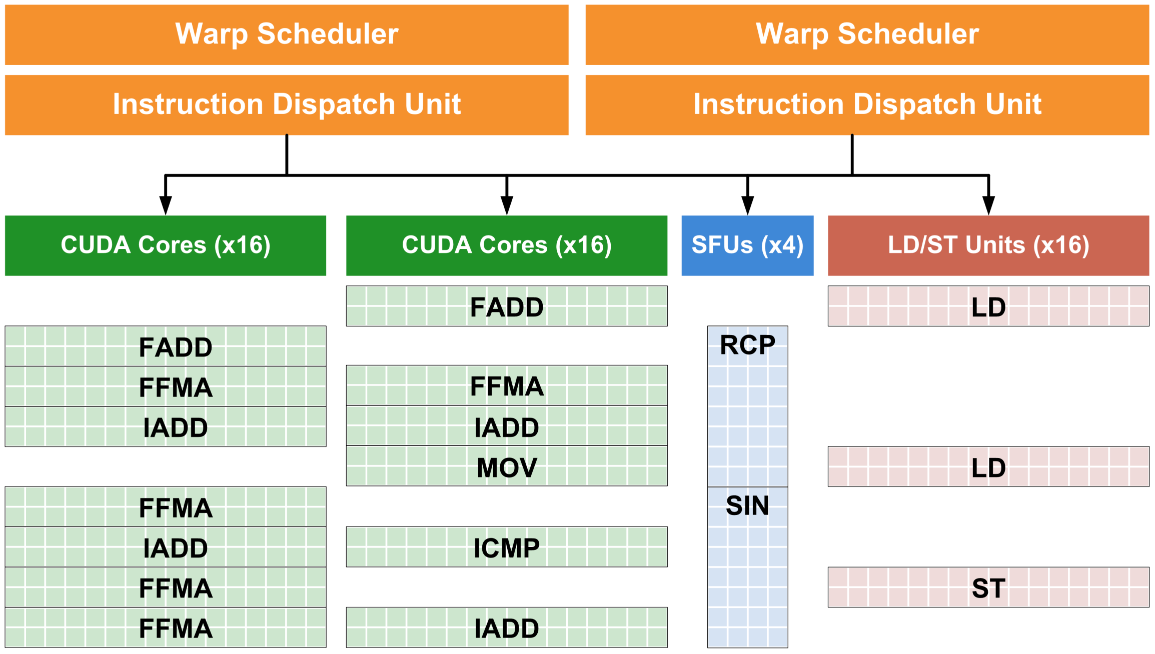


Figure 7. A total of 32 instructions from one or two warps can be dispatched in each cycle to any two of the four execution blocks within a Fermi SM: two blocks of 16 cores each, one block of four Special Function Units, and one block of 16 load/store units. This figure shows how instructions are issued to the execution blocks. (Source: NVIDIA)

ISA improvements

Fermi debuts the Parallel Thread eXecution (PTX) 2.0 instruction-set architecture (ISA). PTX 2.0 defines an instruction set and a new virtual machine architecture that amounts to an idealized processor designed for parallel thread operation.

Because this virtual machine model doesn't literally model the Fermi hardware, it can be portable from one generation to the next. NVIDIA intends PTX 2.0 to span multiple generations of GPU hardware and multiple GPU sizes within each generation, just as PTX 1.0 did.

Compilers supporting NVIDIA GPUs provide PTX-compliant binaries that act as a hardware-neutral distribution format for GPU computing applications and middleware. When applications are installed on a target machine, the GPU driver translates the PTX binaries into the low-level machine instructions that are directly executed by the hardware. (PTX 1.0 binaries can also be translated by Fermi GPU drivers into native instructions.)

This final translation step imposes no further performance penalties. Kernels and libraries for even the most performance-sensitive applications can be hand-coded to the PTX 2.0 ISA, making them portable across GPU generations and implementations.

All of the architecturally visible improvements in Fermi are represented in PTX 2.0. Predication is one of the more significant enhancements in the new ISA.

All instructions support predication. Each instruction can be executed or skipped based on condition codes. Predication allows each thread—each core—to perform different operations as needed while execution continues at full speed. Where predication isn't sufficient, Fermi also supports the usual if-then-else structure with branch statements.

Most CPUs rely exclusively on conditional branches and incorporate branch-prediction hardware to allow speculation along the likely path. That's a reasonable solution for branch-intensive serial code, but less efficient than predication for streaming applications.

Another major improvement in Fermi and PTX 2.0 is a new unified addressing model. All addresses in the GPU are allocated from a continuous 40-bit (one terabyte) address space. Global, shared, and local addresses are defined as ranges within this address space and can be accessed by common load/store instructions. (The load/store instructions support 64-bit addresses to allow for future growth.)

The Cache and Memory Hierarchy

Like earlier GPUs, the Fermi architecture provides for local memory in each SM. New to Fermi is the ability to use some of this local memory as a first-level (L1) cache for global memory references. The local memory is 64K in size, and can be split 16K/48K or 48K/16K between L1 cache and shared memory.

Shared memory, the traditional use for local SM memory, provides low-latency access to moderate amounts of data (such as intermediate results in a series of calculations, one row or column of data for matrix operations, a line of video, etc.). Because the access latency to this memory is also completely predictable, algorithms can be written to interleave loads, calculations, and stores with maximum efficiency.

The decision to allocate 16K or 48K of the local memory as cache usually depends on two factors: how much shared memory is needed, and how predictable the kernel's accesses to global memory (usually the off-chip DRAM) are likely to be.

A larger shared-memory requirement argues for less cache; more frequent or unpredictable accesses to larger regions of DRAM argues for more cache.

Some embedded processors support local memory in a similar way, but this feature is almost never available on a PC or server processor because mainstream operating systems have no way to manage local memory; there is no support for it in their programming models. This is one of the reasons why high-performance computing applications running on general-purpose processors are so frequently bottlenecked by memory bandwidth; the application has no way to manage where memory is allocated and algorithms can't be fully optimized for access latency.

Each Fermi GPU is also equipped with an L2 cache (768KB in size for a 512-core chip). The L2 cache covers GPU local DRAM as well as system memory.

The L2 cache subsystem also implements another feature not found on CPUs: a set of memory read-modify-write operations that are atomic—that is, uninterruptible—and thus ideal for managing access to data that must be shared across thread blocks or even kernels. Normally this functionality is provided through a two-step process; a CPU uses an atomic test-and-set instruction to manage a semaphore, and the semaphore manages access to a predefined location or region in memory.

Fermi can implement that same solution when needed, but it's much simpler from the software perspective to be able to issue a standard integer ALU operation that performs the atomic operation directly rather than having to wait until a semaphore becomes available.

Fermi's atomic operations are implemented by a set of integer ALUs logically that can lock access to a single memory address while the read-modify-write sequence is completed. This memory address can be in system memory, in the GPU's locally connected DRAM, or even in the memory spaces of other PCI Express-connected devices. During the brief lock interval, the rest of memory continues to operate normally. Locks in system memory are atomic with respect to the operations of the GPU performing the atomic operation; software synchronization is ordinarily used to assign regions of memory to GPU control, thus avoiding conflicting writes from the CPU or other devices.

Consider a kernel designed to calculate a histogram for an image, where the histogram consists of one counter for each brightness level in the image. A CPU might loop through the whole image and increment the appropriate counter value based on the brightness of each pixel. A GPU without atomic operations might assign one SM to each part of the image and let them run until they're all done (by imposing a synchronization barrier) and then run a second short program to add up all the results.

With the atomic operations in Fermi, once the regional histograms are computed, those results can be combined into the final histogram using atomic add operations; no second pass is required.

Similar improvements can be made in other applications such as ray tracing, pattern recognition, and linear algebra routines such as matrix multiplication as implemented in the commonly used Basic Linear Algebra Subprograms (BLAS). According to NVIDIA, atomic operations on Fermi are 5× to 20× faster than on previous GPUs using conventional synchronization methods.

The final stage of the local memory hierarchy is the GPU's directly connected DRAM. Fermi provides six 64-bit DRAM channels that support SDDR3 and GDDR5 DRAMs. Up to 6GB of GDDR5 DRAM can be connected to the chip for a significant boost in capacity and bandwidth over NVIDIA's previous products.

Fermi is the first GPU to provide ECC (error correcting code) protection for DRAM; the chip's register files, shared memories, L1 and L2 caches are also ECC protected. The level of protection is known as SECDED: single (bit) error correction, double error detection. SECDED is the usual level of protection in most ECC-equipped systems.

Fermi's ECC protection for DRAM is unique among GPUs; so is its implementation. Instead of each 64-bit memory channel carrying eight extra bits for ECC information, NVIDIA has a proprietary (and undisclosed) solution for packing the ECC bits into reserved lines of memory.

The GigaThread controller that manages application context switching (described earlier) also provides a pair of streaming data-transfer engines, each of which can fully saturate Fermi's PCI Express host interface. Typically, one will be used to move data from system memory to GPU memory when setting up a GPU computation, while the other will be used to move result data from GPU memory to system memory.

Conclusions

In just a few years, NVIDIA has advanced the state of the art in GPU design from almost purely graphics-focused products like the 7800 series to the flexible Fermi architecture.

Fermi is still derived from NVIDIA's graphics products, which ensures that NVIDIA will sell millions of software-compatible chips to PC gamers. In the PC market, Fermi's capacity for GPU computing will deliver substantial improvements in gameplay, multimedia encoding and enhancement, and other popular PC applications.

Those sales to PC users generate an indirect benefit for customers interested primarily in high-performance computing: Fermi's affordability and availability will be unmatched by any other computing architecture in its performance range.

Although it may sometimes appear that GPUs are becoming more CPU-like with each new product generation, fundamental technical differences among computing applications lead to lasting differences in how CPUs and GPUs are designed and used.

CPUs will continue to be best for dynamic workloads marked by short sequences of computational operations and unpredictable control flow. Modern CPUs, which devote large portions of their silicon real estate to caches, large branch predictors, and complex instruction set decoders will always be optimized for this kind of code.

At the other extreme, workloads that are dominated by computational work performed within a simpler control flow need a different kind of processor architecture, one optimized for streaming calculations but also equipped with the ability to support popular programming languages. Fermi is just such an architecture.

Fermi is the first computing architecture to deliver such a high level of double-precision floating-point performance from a single chip with a flexible, error-protected memory hierarchy and support for languages including C++ and FORTRAN. As such, Fermi is the world's first complete GPU computing architecture.