Whitepaper

NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™

Table of Contents

A Brief History of GPU Computing

The graphics processing unit (GPU), first invented by NVIDIA in 1999, is the most pervasive parallel processor to date. Fueled by the insatiable desire for life-like real-time graphics, the GPU has evolved into a processor with unprecedented floating-point performance and programmability; today's GPUs greatly outpace CPUs in arithmetic throughput and memory bandwidth, making them the ideal processor to accelerate a variety of data parallel applications.

Efforts to exploit the GPU for non-graphical applications have been underway since 2003. By using high-level shading languages such as DirectX, OpenGL and Cg, various data parallel algorithms have been ported to the GPU. Problems such as protein folding, stock options pricing, SQL queries and MRI reconstruction achieved remarkable performance speedups on the GPU. These early efforts that used graphics APIs for general purpose computing were known as GPGPU programs.

While the GPGPU model demonstrated great speedups, it faced several drawbacks. First, it required the programmer to possess intimate knowledge of graphics APIs and GPU architecture. Second, problems had to be expressed in terms of vertex coordinates, textures and shader programs, greatly increasing program complexity. Third, basic programming features such as addressable read and write were not supported, greatly restricting the programming model. Lastly, the lack of double precision support (until recently) meant some scientific applications could not be run on the GPU.

To address these problems, NVIDIA introduced two key technologies—the G80 unified graphics and compute architecture (first introduced in GeForce 8800®, Quadro FX 5600®, and Tesla C870® GPUs), and CUDA, a software and hardware architecture that enabled the GPU to be programmed with a variety of high level programming languages. Together, these two technologies represented a new way of using the GPU. Instead of programming dedicated graphics units with graphics APIs, the programmer could now write C programs with CUDA extensions and target a general purpose, massively parallel processor. We called this new way of GPU programming "GPU Computing"—it signified broader application support, wider programming language support, and a clear separation from the early "GPGPU" model of programming.

The G80 Architecture

NVIDIA's GeForce 8800 was the product that gave birth to the new GPU Computing model. Introduced in November 2006, the G80 based GeForce 8800 brought several key innovations to GPU Computing:

- G80 was the first GPU to support C, allowing programmers to use the power of the GPU without having to learn a new programming language.
- G80 was the first GPU to replace the separate vertex and pixel pipelines with a single, unified processor that executed vertex, geometry, pixel, and computing programs.
- G80 was the first GPU to utilize a scalar thread processor, eliminating the need for programmers to manually manage vector registers.
- G80 introduced the single-instruction multiple-thread (SIMT) execution model where independent multiple threads execute concurrently using a single instruction.
- G80 introduced shared memory and barrier synchronization for inter-thread communication.

In June 2008, NVIDIA introduced a major revision to the G80 architecture. The second generation unified architecture—GT200 (first introduced in the GeForce GTX 280, Quadro FX 5800, and Tesla T10 GPUs)—increased the number of streaming processor cores (subsequently referred to as CUDA cores) from 128 to 240. Each processor register file was doubled in size, allowing a greater number of threads to execute on-chip at any given time. Hardware memory access coalescing was added to improve memory access efficiency. Double precision floating point support was also added to address the needs of scientific and high-performance computing (HPC) applications.

When designing each new generation GPU, it has always been the philosophy at NVIDIA to improve both existing application performance and GPU programmability; while faster application performance brings immediate benefits, it is the GPU's relentless advancement in programmability that has allowed it to evolve into the most versatile parallel processor of our time. It was with this mindset that we set out to develop the successor to the GT200 architecture.

NVIDIA's Next Generation CUDA Compute and Graphics Architecture, Code-Named "Fermi"

The Fermi architecture is the most significant leap forward in GPU architecture since the original G80. G80 was our initial vision of what a unified graphics and computing parallel processor should look like. GT200 extended the performance and functionality of G80. With Fermi, we have taken all we have learned from the two prior processors and all the applications that were written for them, and employed a completely new approach to design to create the world's first computational GPU. When we started laying the groundwork for Fermi, we gathered extensive user feedback on GPU computing since the introduction of G80 and GT200, and focused on the following key areas for improvement:

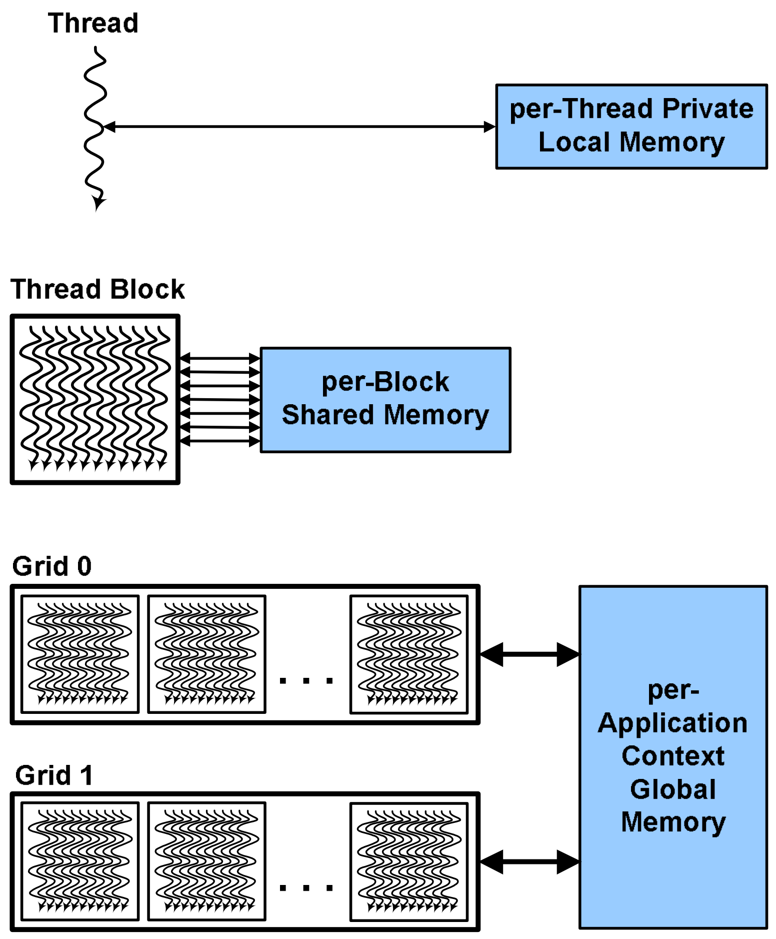
- While single precision floating point performance was on the order of ten times the performance of desktop CPUs, some GPU computing applications desired more double precision performance as well.
- ECC memory so some GPU computing users could deploy large numbers of GPUs in datacenter installations.
- Some parallel algorithms were unable to use the GPU shared memory, and users requested a true cache architecture to aid them.
- Many CUDA programmers requested more than 16 KB of SM shared memory to speed up their applications.
- Users requested faster context switches between application programs and faster graphics and computing interoperation.
- Users requested faster read-modify-write atomic operations for their parallel algorithms.

With these requests in mind, the Fermi team designed a processor that greatly increases raw compute horsepower, and through architectural innovations, also offers dramatically increased programmability and compute efficiency. The key architectural highlights of Fermi are:

- **Third Generation Streaming Multiprocessor (SM)**
  - 32 CUDA cores per SM, 4x over GT200
  - 8x the peak double precision floating point performance over GT200
  - Dual Warp Scheduler that schedules and dispatches two warps of 32 threads per clock
  - 64 KB of RAM with a configurable partitioning of shared memory and L1 cache
- **Second Generation Parallel Thread Execution ISA**
  - Unified Address Space with Full C++ Support
  - Optimized for OpenCL and DirectCompute
  - Full IEEE 754-2008 32-bit and 64-bit precision
  - Full 32-bit integer path with 64-bit extensions
  - Memory access instructions to support transition to 64-bit addressing
  - Improved Performance through Predication
- **Improved Memory Subsystem**
  - NVIDIA Parallel DataCache™ hierarchy with Configurable L1 and Unified L2 Caches
  - First GPU with ECC memory support
  - Greatly improved atomic memory operation performance
- **NVIDIA GigaThread™ Engine**
  - 10x faster application context switching
  - Concurrent kernel execution
  - Out of Order thread block execution
  - Dual overlapped memory transfer engines

A Quick Refresher on CUDA

CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages. A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks. The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.


Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid.

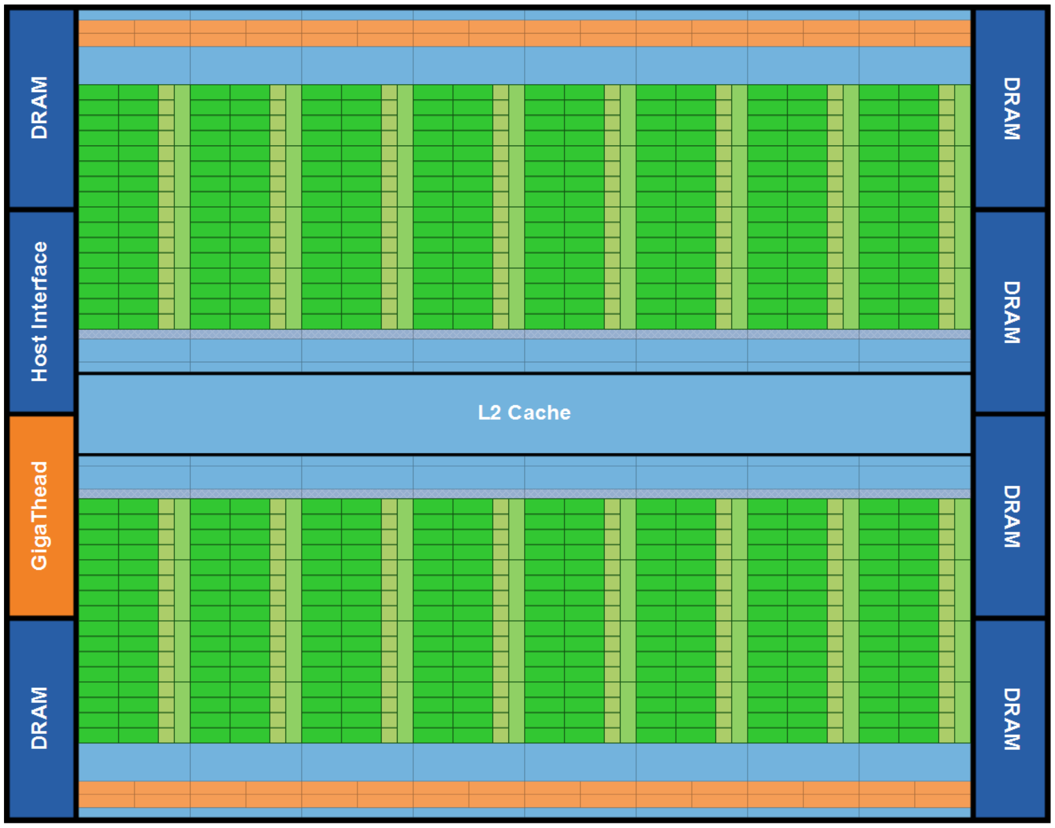
A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space, used for register spills, function calls, and C automatic array variables. Each thread block has a per-Block shared memory space, used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.

Hardware Execution

CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

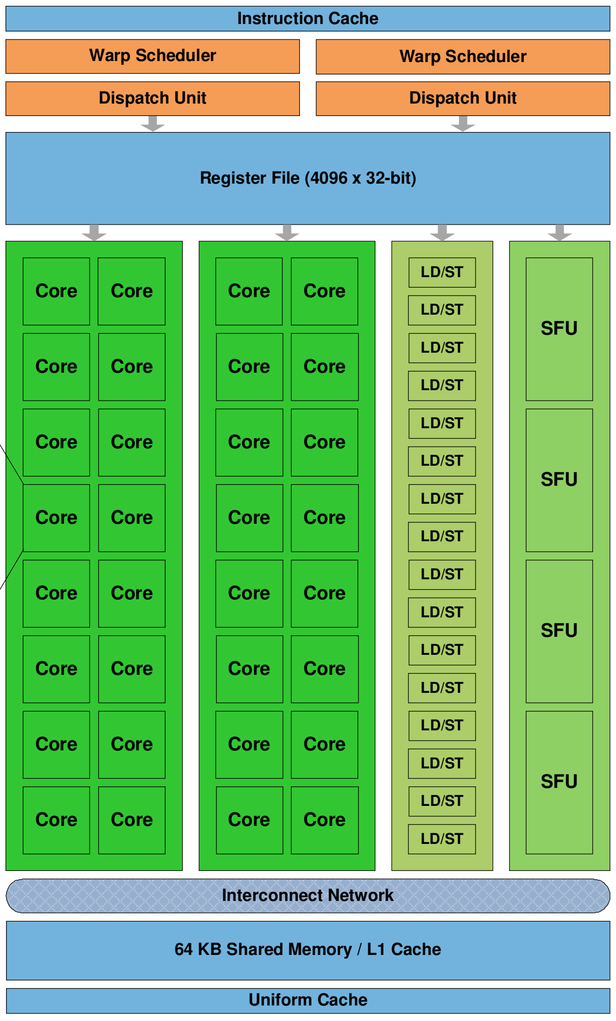
An Overview of the Fermi Architecture

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

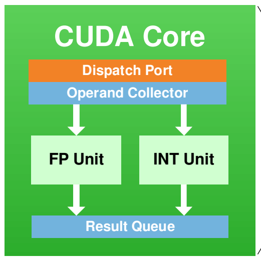
Third Generation Streaming Multiprocessor



Fermi Streaming Multiprocessor (SM)

The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

512 High Performance CUDA cores



Each SM features 32 CUDA processors—a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. FMA is more accurate than performing the operations separately. GT200 implemented double precision FMA.

In GT200, the integer ALU was limited to 24-bit precision for multiply operations; as a result, multi-instruction emulation sequences were required for integer arithmetic. In Fermi, the newly designed integer ALU supports full 32-bit precision for all instructions, consistent with standard programming language requirements. The ALU is optimized to efficiently support 64-bit and extended precision operations. Various instructions are supported, including Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count.

16 Load/Store Units

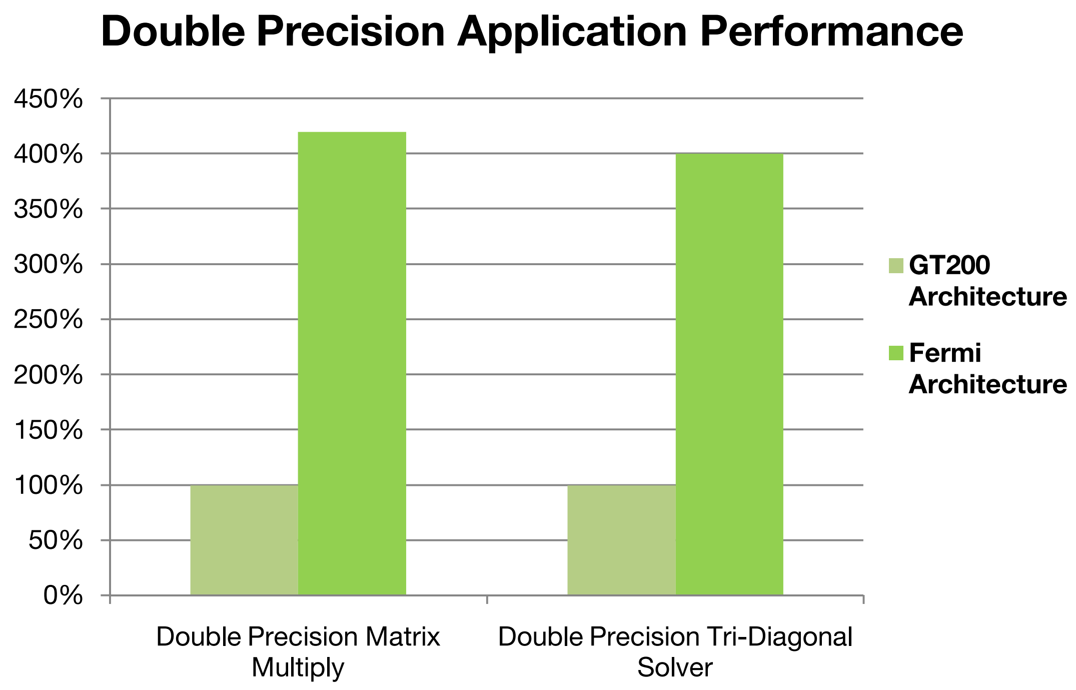
Each SM has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock. Supporting units load and store the data at each address to cache or DRAM.

Four Special Function Units

Special Function Units (SFUs) execute transcendental instructions such as sin, cosine, reciprocal, and square root. Each SFU executes one instruction per thread, per clock; a warp executes over eight clocks. The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.

Designed for Double Precision

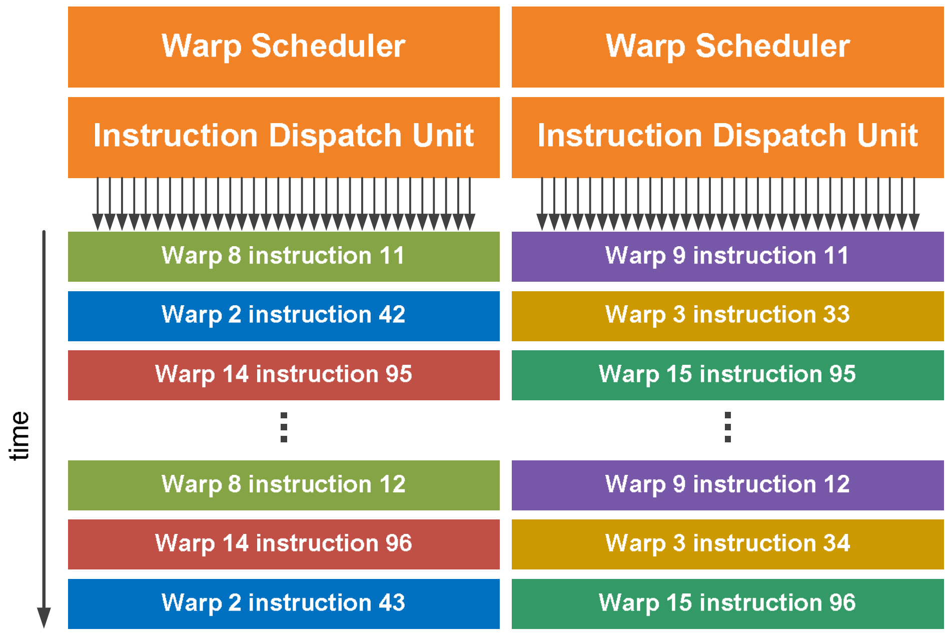
Double precision arithmetic is at the heart of HPC applications such as linear algebra, numerical simulation, and quantum chemistry. The Fermi architecture has been specifically designed to offer unprecedented performance in double precision; up to 16 double precision fused multiply-add operations can be performed per SM, per clock, a dramatic improvement over the GT200 architecture.



Early performance evaluations show Fermi performing up to 4.2x faster than GT200 in double precision applications.

Dual Warp Scheduler

The SM schedules threads in groups of 32 parallel threads called warps. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. Fermi's dual warp scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen load/store units, or four SFUs. Because warps execute independently, Fermi's scheduler does not need to check for dependencies from within the instruction stream. Using this elegant model of dual-issue, Fermi achieves near peak hardware performance.



Most instructions can be dual issued; two integer instructions, two floating instructions, or a mix of integer, floating point, load, store, and SFU instructions can be issued concurrently. Double precision instructions do not support dual dispatch with any other operation.

64 KB Configurable Shared Memory and L1 Cache

One of the key architectural innovations that greatly improved both the programmability and performance of GPU applications is on-chip shared memory. Shared memory enables threads within the same thread block to cooperate, facilitates extensive reuse of on-chip data, and greatly reduces off-chip traffic. Shared memory is a key enabler for many high-performance CUDA applications.

G80 and GT200 have 16 KB of shared memory per SM. In the Fermi architecture, each SM has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache.

For existing applications that make extensive use of Shared memory, tripling the amount of Shared memory yields significant performance improvements, especially for problems that are bandwidth constrained. For existing applications that use Shared memory as software managed cache, code can be streamlined to take advantage of the hardware caching system, while still having access to at least 16 KB of shared memory for explicit thread cooperation. Best of all, applications that do not use Shared memory automatically benefit from the L1 cache, allowing high performance CUDA programs to be built with minimum time and effort.

Summary Table

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Second Generation Parallel Thread Execution ISA

Fermi is the first architecture to support the new Parallel Thread eXecution (PTX) 2.0 instruction set. PTX is a low level virtual machine and ISA designed to support the operations of a parallel thread processor. At program install time, PTX instructions are translated to machine instructions by the GPU driver.

The primary goals of PTX are:

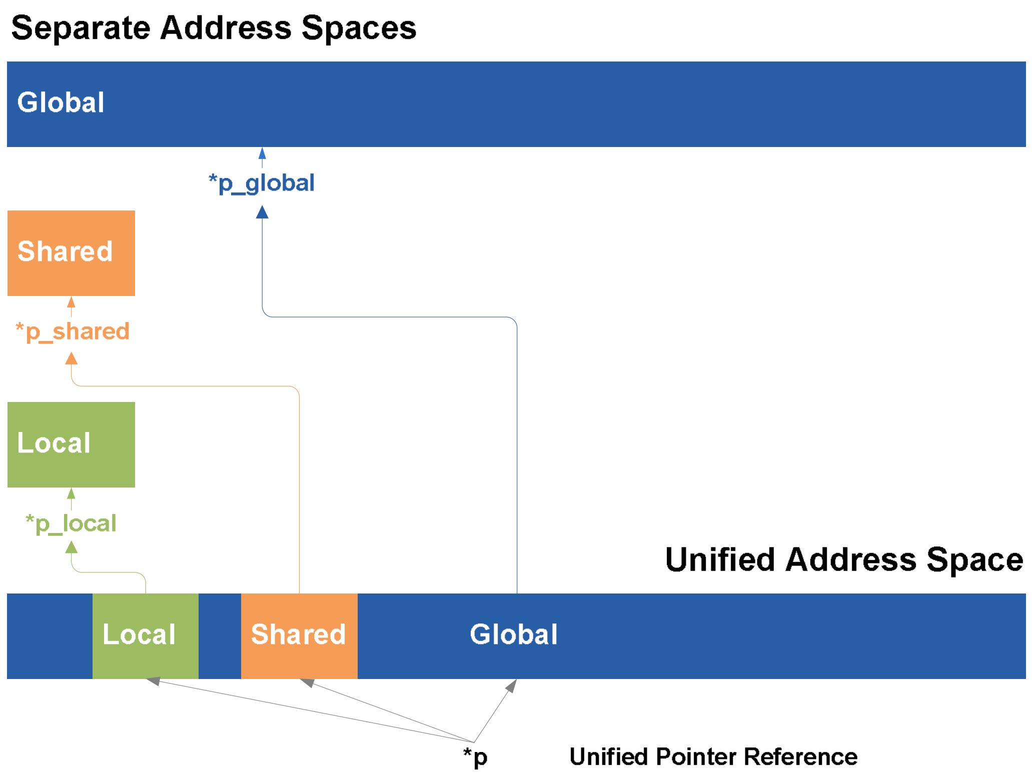
- Provide a stable ISA that spans multiple GPU generations
- Achieve full GPU performance in compiled applications
- Provide a machine-independent ISA for C, C++, Fortran, and other compiler targets.
- Provide a code distribution ISA for application and middleware developers
- Provide a common ISA for optimizing code generators and translators, which map PTX to specific target machines.
- Facilitate hand-coding of libraries and performance kernels
- Provide a scalable programming model that spans GPU sizes from a few cores to many parallel cores

PTX 2.0 introduces several new features that greatly improve GPU programmability, accuracy, and performance. These include: full IEEE 32-bit floating point precision, unified address space for all variables and pointers, 64-bit addressing, and new instructions for OpenCL and DirectCompute. Most importantly, PTX 2.0 was specifically designed to provide full support for the C++ programming language.

Unified Address Space enables Full C++ Support

Fermi and the PTX 2.0 ISA implement a unified address space that unifies the three separate address spaces (thread private local, block shared, and global) for load and store operations. In PTX 1.0, load/store instructions were specific to one of the three address spaces; programs could load or store values in a specific target address space known at compile time. It was difficult to fully implement C and C++ pointers since a pointer's target address space may not be known at compile time, and may only be determined dynamically at run time.

With PTX 2.0, a unified address space unifies all three address spaces into a single, continuous address space. A single set of unified load/store instructions operate on this address space, augmenting the three separate sets of load/store instructions for local, shared and global. The 40-bit unified address space supports a Terabyte of addressable memory, and the load/store ISA supports 64-bit addressing for future growth.



The implementation of a unified address space enables Fermi to support true C++ programs. In C++, all variables and functions reside in objects which are passed via pointers. PTX 2.0 makes it possible to use unified pointers to pass objects in any memory space, and Fermi's hardware address translation unit automatically maps pointer references to the correct memory space.

Fermi and the PTX 2.0 ISA also add support for C++ virtual functions, function pointers, and 'new' and 'delete' operators for dynamic object allocation and de-allocation. C++ exception handling operations 'try' and 'catch' are also supported.

Optimized for OpenCL and DirectCompute

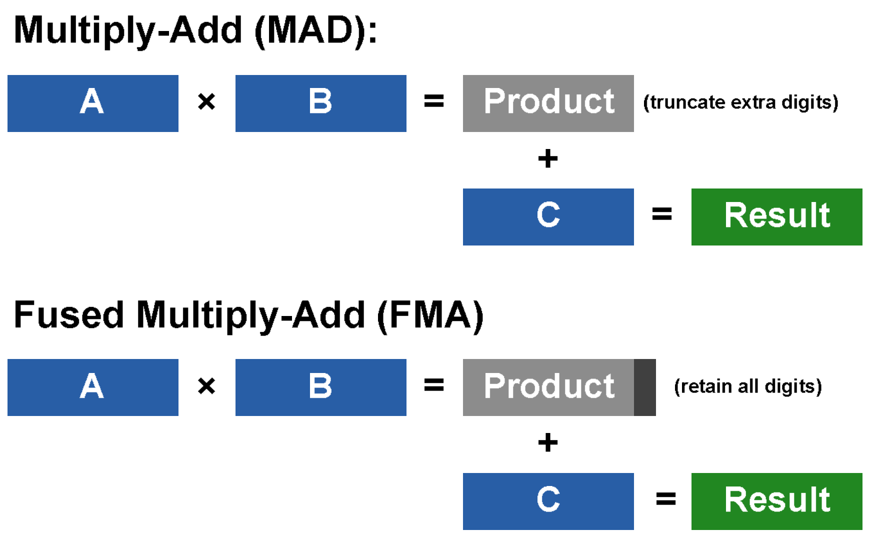
OpenCL and DirectCompute are closely related to the CUDA programming model, sharing the key abstractions of threads, thread blocks, grids of thread blocks, barrier synchronization, per-block shared memory, global memory, and atomic operations. Fermi, a third-generation CUDA architecture, is by nature well-optimized for these APIs. In addition, Fermi offers hardware support for OpenCL and DirectCompute surface instructions with format conversion, allowing graphics and compute programs to easily operate on the same data. The PTX 2.0 ISA also adds support for the DirectCompute instructions population count, append, and bit-reverse.

IEEE 32-bit Floating Point Precision

Single precision floating point instructions now support subnormal numbers by default in hardware, as well as all four IEEE 754-2008 rounding modes (nearest, zero, positive infinity, and negative infinity).

Subnormal numbers are small numbers that lie between zero and the smallest normalized number of a given floating point number system. Prior generation GPUs flushed subnormal operands and results to zero, incurring a loss of accuracy. CPUs typically perform subnormal calculations in exception-handling software, taking thousands of cycles. Fermi's floating point units handle subnormal numbers in hardware, allowing values to gradually underflow to zero with no performance penalty.

A frequently used sequence of operations in computer graphics, linear algebra, and scientific applications is to multiply two numbers, adding the product to a third number, for example, $D = A \times B + C$. Prior generation GPUs accelerated this function with the multiply-add (MAD) instruction that allowed both operations to be performed in a single clock. The MAD instruction performs a multiplication with truncation, followed by an addition with round-to-nearest even. Fermi implements the new fused multiply-add (FMA) instruction for both 32-bit single-precision and 64-bit double-precision floating point numbers (GT200 supported FMA only in double precision) that improves upon multiply-add by retaining full precision in the intermediate stage. The increase in precision benefits a number of algorithms, such as rendering fine intersecting geometry, greater precision in iterative mathematical calculations, and fast, exactly-rounded division and square root operations.

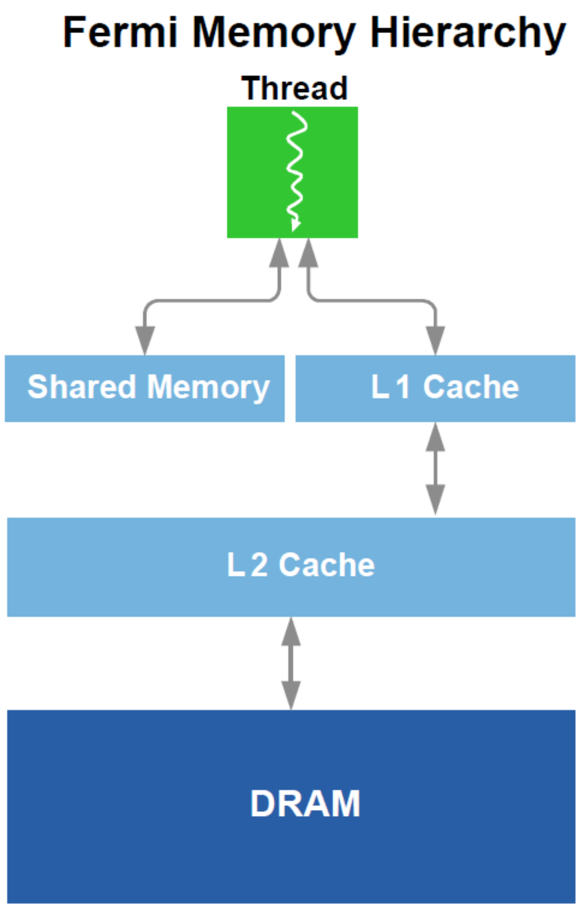


Improved Conditional Performance through Predication

In the Fermi ISA, the native hardware predication support used for divergent thread management is now available at the instruction level. Predication enables short conditional code segments to execute efficiently with no branch instruction overhead.

Memory Subsystem Innovations

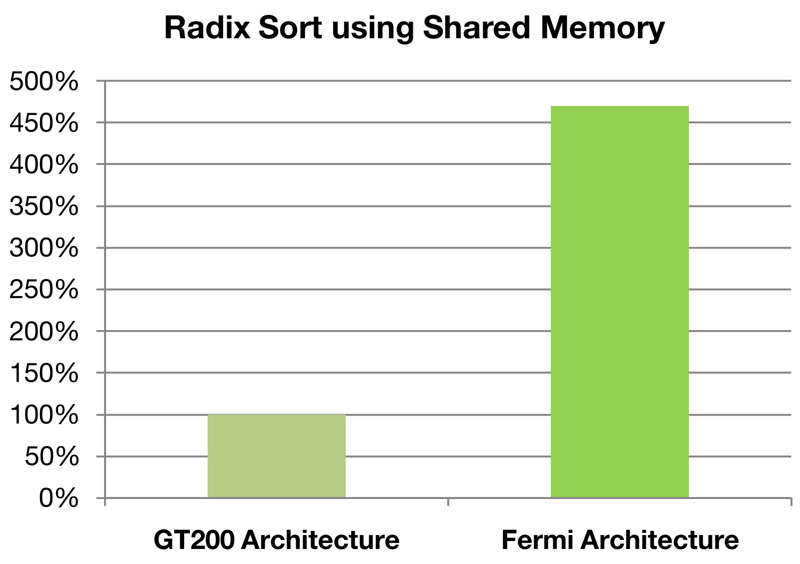
NVIDIA Parallel DataCache™ with Configurable L1 and Unified L2 Cache



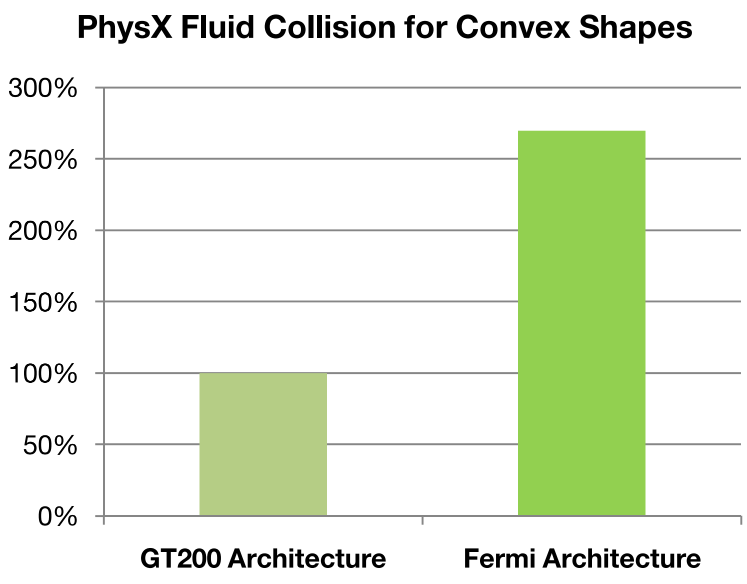
Working with hundreds of GPU computing applications from various industries, we learned that while Shared memory benefits many problems, it is not appropriate for all problems. Some algorithms map naturally to Shared memory, others require a cache, while others require a combination of both. The optimal memory hierarchy should offer the benefits of both Shared memory and cache, and allow the programmer a choice over its partitioning. The Fermi memory hierarchy adapts to both types of program behavior.

Adding a true cache hierarchy for load / store operations presented significant challenges. Traditional GPU architectures support a read-only "load" path for texture operations and a write-only "export" path for pixel data output. However, this approach is poorly suited to executing general purpose C or C++ thread programs that expect reads and writes to be ordered. As one example: spilling a register operand to memory and then reading it back creates a read after write hazard; if the read and write paths are separate, it may be necessary to explicitly flush the entire write / "export" path before it is safe to issue the read, and any caches on the read path would not be coherent with respect to the write data.

The Fermi architecture addresses this challenge by implementing a single unified memory request path for loads and stores, with an L1 cache per SM multiprocessor and unified L2 cache that services all operations (load, store and texture). The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations. The 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache, or 16 KB of Shared memory with 48 KB of L1 cache. When configured with 48 KB of shared memory, programs that make extensive use of shared memory (such as electrodynamic simulations) can perform up to three times faster. For programs whose memory accesses are not known beforehand, the 48 KB L1 cache configuration offers greatly improved performance over direct access to DRAM.



When using 48 KB of shared memory on Fermi, Radix Sort executes 4.7x faster than GT200.



Physics algorithms such as fluid simulations especially benefit from Fermi's caches. For convex shape collisions, Fermi is 2.7x faster than GT200.

In either configuration, the L1 cache also helps by caching temporary register spills of complex programs. Prior generation GPUs spilled registers directly to DRAM, increasing access latency. With the L1 cache, performance scales gracefully with increased temporary register usage.

Fermi features a 768 KB unified L2 cache that services all load, store, and texture requests. The L2 provides efficient, high speed data sharing across the GPU. Algorithms for which data addresses are not known beforehand, such as physics solvers, raytracing, and sparse matrix multiplication especially benefit from the cache hierarchy. Filter and convolution kernels that require multiple SMs to read the same data also benefit.

First GPU with ECC Memory Support

Fermi is the first GPU to support Error Correcting Code (ECC) based protection of data in memory. ECC was requested by GPU computing users to enhance data integrity in high performance computing environments. ECC is a highly desired feature in areas such as medical imaging and large-scale cluster computing.

Naturally occurring radiation can cause a bit stored in memory to be altered, resulting in a soft error. ECC technology detects and corrects single-bit soft errors before they affect the system. Because the probability of such radiation induced errors increase linearly with the number of installed systems, ECC is an essential requirement in large cluster installations.

Fermi supports Single-Error Correct Double-Error Detect (SECDED) ECC codes that correct any single bit error in hardware as the data is accessed. In addition, SECDED ECC ensures that all double bit errors and many multi-bit errors are also be detected and reported so that the program can be re-run rather than being allowed to continue executing with bad data.

Fermi's register files, shared memories, L1 caches, L2 cache, and DRAM memory are ECC protected, making it not only the most powerful GPU for HPC applications, but also the most reliable. In addition, Fermi supports industry standards for checking of data during transmission from chip to chip. All NVIDIA GPUs include support for the PCI Express standard for CRC check with retry at the data link layer. Fermi also supports the similar GDDR5 standard for CRC check with retry (aka "EDC") during transmission of data across the memory bus.

Fast Atomic Memory Operations

Atomic memory operations are important in parallel programming, allowing concurrent threads to correctly perform read-modify-write operations on shared data structures. Atomic operations such as add, min, max, and compare-and-swap are atomic in the sense that the read, modify, and write operations are performed without interruption by other threads. Atomic memory operations are widely used for parallel sorting, reduction operations, and building data structures in parallel without locks that serialize thread execution.

Thanks to a combination of more atomic units in hardware and the addition of the L2 cache, atomic operations performance is up to 20× faster in Fermi compared to the GT200 generation.

GigaThread^{TM} Thread Scheduler

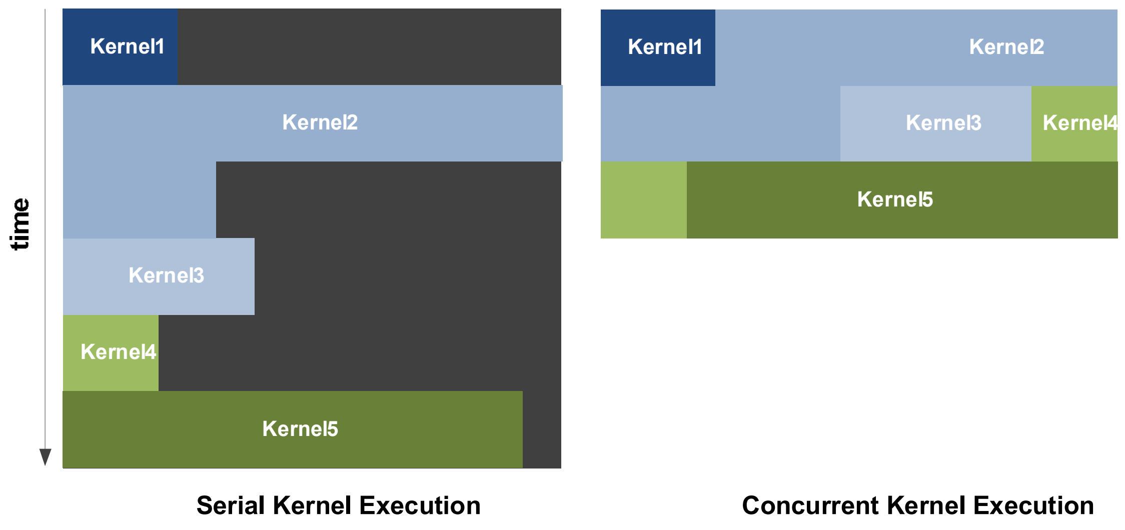
One of the most important technologies of the Fermi architecture is its two-level, distributed thread scheduler. At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units. The first generation GigaThread engine introduced in G80 managed up to 12,288 threads in realtime. The Fermi architecture improves on this foundation by providing not only greater thread throughput, but dramatically faster context switching, concurrent kernel execution, and improved thread block scheduling.

10x Faster Application Context Switching

Like CPUs, GPUs support multitasking through the use of context switching, where each program receives a time slice of the processor's resources. The Fermi pipeline is optimized to reduce the cost of an application context switch to below 25 microseconds, a significant improvement over last generation GPUs. Besides improved performance, this allows developers to create applications that take greater advantage of frequent kernel-to-kernel communication, such as fine-grained interoperation between graphics and PhysX applications.

Concurrent Kernel Execution

Fermi supports concurrent kernel execution, where different kernels of the same application context can execute on the GPU at the same time. Concurrent kernel execution allows programs that execute a number of small kernels to utilize the whole GPU. For example, a PhysX program may invoke a fluids solver and a rigid body solver which, if executed sequentially, would use only half of the available thread processors. On the Fermi architecture, different kernels of the same CUDA context can execute concurrently, allowing maximum utilization of GPU resources. Kernels from different application contexts can still run sequentially with great efficiency thanks to the improved context switching performance.

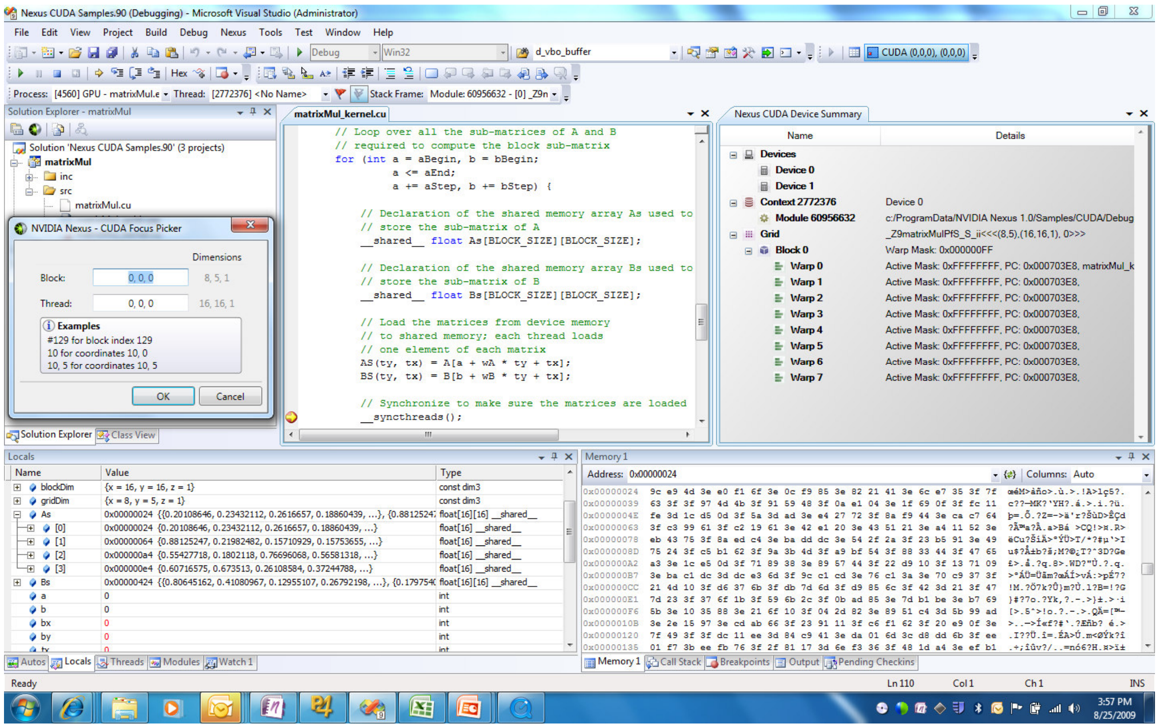


Introducing NVIDIA Nexus

NVIDIA Nexus is the first development environment designed specifically to support massively parallel CUDA C, OpenCL, and DirectCompute applications. It bridges the productivity gap between CPU and GPU code by bringing parallel-aware hardware source code debugging and performance analysis directly into Microsoft Visual Studio, the most widely used integrated application development environment under Microsoft Windows.

Nexus allows Visual Studio developers to write and debug GPU source code using exactly the same tools and interfaces that are used when writing and debugging CPU code, including source and data breakpoints, and memory inspection. Furthermore, Nexus extends Visual Studio functionality by offering tools to manage massive parallelism, such as the ability to focus and debug on a single thread out of the thousands of threads running parallel, and the ability to simply and efficiently visualize the results computed by all parallel threads.

Nexus is the perfect environment to develop co-processing applications that take advantage of both the CPU and GPU. It captures performance events and information across both processors, and presents the information to the developer on a single correlated timeline. This allows developers to see how their application behaves and performs on the entire system, rather than through a narrow view that is focused on a particular sub-system or processor.


Nexus CUDA Samples.90 (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Nexus Tools Test Window Help
Debug
Win32
d_vbo_buffer
Hex
CUDA (0,0,0), (0,0,0)
Solution Explorer - matrixMul
matrixMul_kernel.cu
Nexus CUDA Device Summary
Solution Explorer
Class View
Locals
Memory 1
Autos
Locals
Threads
Modules
Watch 1
Call Stack
Breakpoints
Output
Pending Checkins
OK
Cancel

Process: [4560] GPU - matrixMul.e Thread: [2772376] <No Name> Stack Frame: Module: 60956632 - [0] _Z9n

Solution 'Nexus CUDA Samples.90' (3 projects)
matrixMul
inc
src
matrixMul.cu

NVIDIA Nexus - CUDA Focus Picker

		Dimensions
Block:	0, 0, 0	8, 5, 1
Thread:	0, 0, 0	16, 16, 1

Examples
#129 for block index 129
10 for coordinates 10, 0
10, 5 for coordinates 10, 5

```
    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();
```

Name	Details
Devices	
Device 0	
Device 1	
Context 2772376	Device 0
Module 60956632	c:/ProgramData/NVIDIA Nexus 1.0/Samples/CUDA/Debug
Grid	_Z9matrixMulPfS_S_ii<<<(8,5),(16,16,1), 0>>>
Block 0	Warp Mask: 0x000000FF
Warp 0	Active Mask: 0xFFFFFFFF, PC: 0x000703E8, matrixMul_k
Warp 1	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 2	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 3	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 4	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 5	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 6	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,
Warp 7	Active Mask: 0xFFFFFFFF, PC: 0x000703E8,

Name	Value	Type
blockDim	{x = 16, y = 16, z = 1}	const dim3
gridDim	{x = 8, y = 5, z = 1}	const dim3
As	0x00000024 {{0.20108646, 0.23432112, 0.2616657, 0.18860439, ...}, {0.8812524	float[16][16] __shared__
[0]	0x00000024 {0.20108646, 0.23432112, 0.2616657, 0.18860439, ...}	float[16] __shared__
[1]	0x00000064 {0.88125247, 0.21982482, 0.15710929, 0.15753655, ...}	float[16] __shared__
[2]	0x000000a4 {0.55427718, 0.1802118, 0.76696068, 0.56581318, ...}	float[16] __shared__
[3]	0x000000e4 {0.60716575, 0.673513, 0.26108584, 0.37244788, ...}	float[16] __shared__
Bs	0x00000424 {{0.80645162, 0.41080967, 0.12955107, 0.26792198, ...}, {0.179754	float[16][16] __shared__
a	0	int
b	0	int
bx	0	int
by	0	int
tx	0	int

Address: 0x00000024 Columns: Auto

```
0x00000024  9c e9 4d 3e e0 f1 6f 3e 0c f9 85 3e 82 21 41 3e 6c e7 35 3f 7f  œéM>àño>.ù.>.!A>lç5?.
0x00000039  63 3f 3f 97 4d 4b 3f 91 59 48 3f 0a e1 04 3e 1f 69 0f 3f fc 11  c??—MK?'YH?.á.>.i.?ü.
0x0000004E  fe 3d 1c d5 0d 3f 5a 3d ad 3e e4 27 72 3f 8a f9 44 3e ca c7 64  þ=.Õ.?Z=->ä'r?ŠùD>ÊÇd
0x00000063  3f c3 99 61 3f c2 19 61 3e 42 e1 20 3e 43 51 21 3e a4 11 52 3e  ?Ã™a?Â.a>Bá >CQ!>¤.R>
0x00000078  eb 43 75 3f 8a ed c4 3e ba dd dc 3e 54 2f 2a 3f 23 b5 91 3e 49  ëCu?ŠíÄ>ºÝÜ>T/*?#µ'>I
0x0000008D  75 24 3f c5 b1 62 3f 9a 3b 4d 3f a9 bf 54 3f 88 33 44 3f 47 65  u$?Å±b?š;M?©¿T?ˆ3D?Ge
0x000000A2  a3 3e 1c e5 0d 3f 71 89 38 3e 89 57 44 3f 22 d9 10 3f 13 71 09  £>.å.?q.8>.WD?"Ù.?.q.
0x000000B7  3e ba c1 dc 3d dc e3 6d 3f 9c c1 cd 3e 76 c1 3a 3e 70 c9 37 3f  >ºÁÜ=Üãm?œÁÍ>vÁ:>pÉ7?
0x000000CC  21 4d 10 3f d6 37 6b 3f db 7d 6d 3f d9 85 6c 3f 42 3d 21 3f 47  !M.?Ö7k?Û}m?Ù.l?B=!?G
0x000000E1  7d 23 3f 37 6f 1b 3f 59 6b 2c 3f 0b ad 85 3e 7d b1 be 3e b7 69  }#?7o.?Yk,?.-.>}±.>·i
0x000000F6  5b 3e 10 35 88 3e 21 6f 10 3f 04 2d 82 3e 89 51 c4 3d 5b 99 ad  [>.5ˆ>!o.?.-.>.QÄ=[™-
0x0000010B  3e 2e 15 97 3e cd ab 66 3f 23 91 11 3f c6 f1 62 3f 20 e9 0f 3e  >..—>Í«f?#'.?Æñb? é.>
0x00000120  7f 49 3f 3f dc 11 ee 3d 84 c9 41 3e da 01 6d 3c d8 dd 6b 3f ee  .I??Ü.î=.ÉA>Ú.m<ØÝk?î
0x00000135  01 f7 3b ee fb 76 3f 2f 81 17 3d 6e f3 36 3f 48 1d a4 3e ef b1  .÷;îûv?/..=nó6?H.¤>ï±
```

Ready   Ln 110   Col 1   Ch 1   INS

3:57 PM
8/25/2009

NVIDIA Nexus integrated development environment

Conclusion

For sixteen years, NVIDIA has dedicated itself to building the world's fastest graphics processors. While G80 was a pioneering architecture in GPU computing, and GT200 a major refinement, their designs were nevertheless deeply rooted in the world of graphics. The Fermi architecture represents a new direction for NVIDIA. Far from being merely the successor to GT200, Fermi is the outcome of a radical rethinking of the role, purpose, and capability of the GPU.

Rather than taking the simple route of adding execution units, the Fermi team has tackled some of the toughest problems of GPU computing. The importance of data locality is recognized through Fermi's two level cache hierarchy and its combined load/store memory path. Double precision performance is elevated to supercomputing levels, while atomic operations execute up to twenty times faster. Lastly, Fermi's comprehensive ECC support strongly demonstrates our commitment to the high-performance computing market.

On the software side, the architecture brings forward support for C++, the world's most ubiquitous object-orientated programming language, and Nexus, the world's first integrated development environment designed for massively parallel GPU computing applications.

With its combination of ground breaking performance, functionality, and programmability, the Fermi architecture represents the next revolution in GPU computing.