CUDA, Supercomputing for the Masses: Part 21

The Fermi architecture and CUDA

November 19, 2010
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/228300263

In CUDA, Supercomputing for the Masses Part 20, I focused on the analysis capability of Parallel Nsight v1.0 coupled with the NVIDIA Tools extension (NVTX) library to illustrate asynchronous I/O, hybrid CPU/GPU computing, and the performance of primitive restart to dramatically accelerate OpenGL rendering in CUDA applications. (Note that Parallel Nsight 1.5 has been released, which is now compatible with Visual Studio 2010 and further refines the Parallel Nsight experience.)

This article will focus on Fermi and the architectural changes that significantly broaden the types of applications that map well to GPGPU computing while maintaining the performance benefits provided by previous generations of CUDA-enabled GPUs. Particular attention will be paid to how the Fermi architecture affects CUDA memory spaces. Also discussed will be how the Fermi architecture moves GPU computing into mainstream 24/7 production computing with error correction and other robustness features.

Fermi is the internal name that NVIDIA uses for the GF100 architecture that has many expanded capabilities to overcome computational limitations in the previous G80 and follow-on GT200 series of architectures. Variants of the Fermi architecture are used in the GeForce 400 and Tesla 20-series of products.

GPGPU computing has now permeated all aspect of global computing technology. From ultra-low-power CUDA-enabled GPUs to the largest supercomputers in the world such as China's Tianhe-1A (meaning Milky Way), which can perform 1 quadrillion peak floating-point operations, GPGPU computing is redefining what is possible on a computer. Review my article for the GPU Source issue of *Scientific Computing*, "Redefining what is possible" and my GTC presentation "Supercomputing for the Masses: Killer-Apps, Parallel Mappings, Scalability and Application Lifespan" for a more in-depth analysis. Tianhe-1A was recently named as the world's fastest supercomputer in the Top500 list.

Aside from creating opportunities throughout science and industry for the developers of GPU software, this ubiquity (as illustrated by NVIDIA's claim of 250+ million CUDA-enabled GPUs sold to date) has driven the evolution of CUDA-enabled GPU architectures so they can efficiently run applications in more problem domains. Further, it has forced GPU hardware designers to harden GPGPU technology against common errors so that many GPGPUs can simultaneously be used in 24/7 production environments to reliably run applications for extended periods measured in days, weeks and months. Examples include rendering farms that create animated movies and supercomputers that run some of the largest physics simulations in the world.

Both as a result of Fermi and also due to the maturation of CUDA and GPU programming in general, the thinking behind how to program GPU technology is changing. Just as the Bebop (Berkeley Benchmarking and OPtimization) group led the way with publications like the Volkov and Demmel paper "Benchmarking GPUs to Tune Dense Linear Algebra" for high performance on earlier GPU architectures, so are they are changing the thinking about occupancy as is discussed in the hyperlink and will be discussed in this article.

An overview of Fermi changes

A brief overview of changes made in the GF100 architecture includes:

- A unified 64-bit memory space with:
  - Pointers can now refer to local, shared, and global memory locations, and are portable among threads.
  - Support for a per-thread stack and recursion.
  - Full 32-bit ALU (Arithmetic Logic Unit) integer operations.
  - Improved 64-bit data paths in shared memory.
  - ECC capability on all global and internal memory and other robustness improvements.
- Fermi's upgraded configurable L1 cache and unified coherent L2 cache across the GPU provide:
  - The ability to broadcast read-only cached data from global memory just like constant memory.
  - Registers now spill to fast cache rather than global memory, which might speed application performance.
  - Accelerated irregular memory accesses within the cache.
  - An order of magnitude (10x) faster atomic operations.
- An improved GigaThread engine that:
  - Supports concurrent kernel execution (and increased efficiency for unbalanced loads).
  - Provides 10x faster context switching (that demonstrates a nearly 5x faster frames per second, FPS, rate on the virtual terrain demo from Part 18).
  - Delivers concurrent bi-directional data transfers to/from the GPU across the PCIe bus.
- Numerous streaming-multiprocessor improvements including:
  - Dual-dispatch scheduling that allows better utilization of the SFU (Special Function Units), integer and other pipelines.
  - Hardware that accelerates small conditional branching and predication.
  - Improved speed and accuracy of various math operations.

All these hardware capabilities translate to a higher-performance more generalized CPU-like GPGPU programming experience that can efficiently support a broader range of applications. Kudos to the CUDA software development teams that have leveraged these capabilities to further increase performance and support Fermi architecture GPGPUs including:

- Recursive functions.
- Function pointers.
  - (Note: in CUDA 3.2 use `__forceinline__` on functions to force inlining again.)
- C++ features such as:
  - Virtual functions.
  - On GPU "new" and "delete" operators for dynamic objects on the GPU.
  - Try/catch/throw exception handling.

Please see the Fermi Compatibility guide to understand how Fermi related changes have affected the `nvcc` compiler command-line arguments for building Fermi CUDA applications.

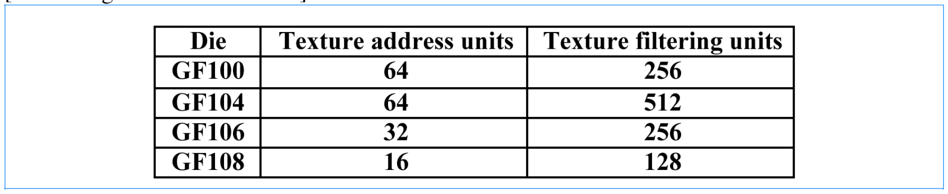
Fermi architecture products include the GF100 and variants classified as the GF104/106/108 and just released GF110 series. Differences include:

- **GF100:**
  - Designated as compute capability 2.0 devices.
  - Each Streaming Multiprocessor (SM) contains:
    - 32 Shader Processors (SP).
    - 4 SFU (Special Function Units).
    - 4 texture filtering units for every texture address unit or Render Output Unit (ROP).

- **GF104/106/108:**
- Designated as compute capability 2.1 devices.
- Each Streaming Multiprocessor (SM) contains:
  - 48 Shader Processors (SP).
  - 8 SFU (Special Function Units).
  - 8 texture filtering units for every texture address unit or Render Output Unit (ROP).

Each complete die contains varying amounts of texture capabilities shown in Table 1:

[Click image to view at full size]



Die	Texture address units	Texture filtering units
GF100	64	256
GF104	64	512
GF106	32	256
GF108	16	128

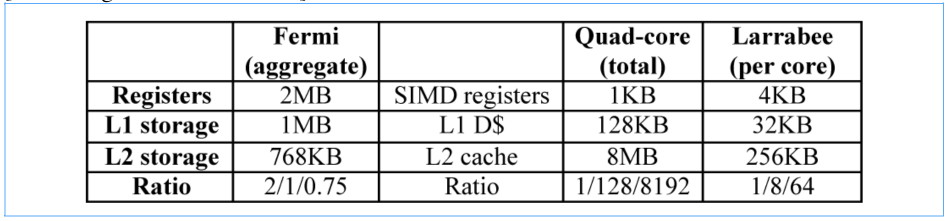
Table 1

Various features of the GF100 architecture are available only on the more expensive Tesla series of cards. For consumer products, double precision performance has been limited to a quarter of that of the "full" Fermi architecture. Error checking and correcting memory (ECC) is also disabled on consumer cards.

Vasily Volkov has an excellent set of slides discussing how Fermi follows the trend towards an inverse memory hierarchy, "Programming inverse memory hierarchy: case of stencils on GPUs." The basic idea is that registers and on-chip local memory is fast and can scale with local processors and massive numbers of threads. This suggests that tiled/stencil algorithms on such systems should be stored in the upper, larger levels of the memory hierarchy such as registers instead of the traditionally used lower, smaller levels such as caches or local stores. (His slides, "Better Performance at Lower Occupancy" from the 2010 GTC conference are also an excellent source of information.)

Fermi, along with other processors, seem to be following this trend towards an inverse memory hierarchy as shown in this table created with data from, "Programming inverse memory hierarchy: case of stencils on GPUs." Note how the gap in memory hierarchy ratios is decreasing as parallelism increases. Succinctly, a single-thread won't see the inverse hierarchy. The inversion is caused by massive-parallelism, which motivates the use of large amounts of thread local data that can scale along with the number of processing elements. This also ties into the ILP (Instruction Level Parallelism) discussion in "Registers and warp scheduling" section of this article, see Table 2.

[Click image to view at full size]



	Fermi (aggregate)		Quad-core (total)	Larrabee (per core)
Registers	2MB	SIMD registers	1KB	4KB
L1 storage	1MB	L1 D$	128KB	32KB
L2 storage	768KB	L2 cache	8MB	256KB
Ratio	2/1/0.75	Ratio	1/128/8192	1/8/64

Table 2

A unified 64-bit address space

The Fermi architecture now supports a 64-bit address space. Providing 40-bits of physical addressing capability, Fermi combines local, shared and global memory in the same address space, which means pointers work as expected! New GPUs such as the C2050 and C2070 currently support 3 and 6 GB of global memory respectively. It is likely that future GPUs will contain even more onboard memory.

Fermi also significantly improves support for working with 64-bit quantities over previous generations of GPGPUs both in moving data to/from shared memory and when operating on 64-bit quantities in terms of the speed and accuracy.

32-bit integer ALU operations

To support calculations in the larger address space, the integer ALU now supports full 32-bit precision for all instructions, and has been optimized to efficiently support 64-bit and extended precision operations. (Various other instructions are also supported including Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count.)

In contrast, the GT200 integer ALU was limited to 24 bits of precision. This caused multiple instructions to be utilized to perform a 32-bit integer multiply unless either `__mult24()` or `__umult24()` was specified by the programmer to perform the multiplication using 24-bits. On Fermi, basic indexing operations such as the calculation of the thread ID (illustrated below) now happen faster.

```
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

Developers with legacy applications should note that the situation has now reversed and that `__mult24()` or `__umult24()` now require multiple instructions on Fermi GPUs! While the performance impact will likely be negligible, legacy codes should use something like the following preprocessor define to get rid of 24-bit multiplications.

```
#define __mul24(a,b) ((a)*(b))
```

Pointers work as expected

Aside from the larger memory capacity, the big news about the unified address space is that pointers now work the way one expects. Basically pointers can now be passed around between threads and they will correctly point to the same physical memory location regardless of the thread block that reads or writes the pointer. Earlier architectures utilized offsets relative to a particular memory space, which meant that a pointer was meaningful only in the memory space to which it belonged. Incorporating a unified address space into the Fermi architecture greatly enhances the ability to share objects, data, and calculation results between threads, both within a single application's threads and between application kernels as well.

Per the NVIDIA Programming manual section 3.1.6, pointers are now 64-bit. Use `-m64` with the 32-bit `nvcc` compiler to emulate 64-bit pointers and `-m32` with the 64-bit nvcc compiler to generate 32-bit pointers.

Support for a per thread stack and recursion

Fermi now provides a stack, which is a fundamental data structure in computer science that is now usable on the GPU on a per thread basis. Basically, each thread can push data onto the stack (up to 1KB per thread per the 3.1 release notes) and use pop operations to pull data off the stack in a LIFO (Last-In First-Out) fashion.

Incorporating a stack per thread allows functions to recursively call themselves among other useful capabilities. Recursion is a technique used to break a large problem into increasingly smaller pieces. At some point, the problem becomes trivially small and easily solved - after which the individuals pieces are reassembled bit-by-bit to generate the solution to the overall, larger problem. Recursion is an important part of divide-and-conquer algorithms for sorting and other general computer science algorithms.

Certain rendering algorithms like ray tracing also utilize recursion. On previous architectures, developers had to implement a limited depth state machine or use in-line multiple nested function calls to emulate recursion in a limited sense. That is why ray-tracing and other recursive algorithms were able to run on GT200 and earlier GPGPU architectures. Now that GPGPUS have a stack, recursion is easy to implement as functions can simply call themselves, which provides CUDA developers a powerful ability to utilize recursive programming techniques to solve problems.

Fermi's upgraded configurable L1 cache

One of the most anticipated and exciting additions in the Fermi architecture is the addition of a true L1/L2 cache hierarchy.

Each SM has an increased amount (64KB) of local memory that can be partitioned to favor shared memory or dynamic read/write operations. Note that the L1 cache is designed for *spatial and not temporal reuse*. It is not an LRU cache like most CPU caches. In addition, the Fermi L1 cache can be deactivated (as discussed here in the forums) with the `-Xptxas -dlcm=cg` command-line argument to `nvcc`. Even when deactivated, both the stack and local memory still reside in the L1 cache memory.

The beauty in this configurability is that applications that reuse data or have misaligned, unpredictable or irregular memory access patterns can configure the L1 cache as a 48KB dynamic cache (leaving 16 KB for shared memory) while applications that need to share more data amongst threads inside a thread block can assign 48KB as shared memory (leaving 16KB for the cache). In this way, the NVIDIA designers empowered the application developer so they can configure the memory within the SM to achieve the best performance.

The L1 caches per-thread local data such as register spills and stack accesses. It caches global memory reads, which can provide a significant performance benefit if the compiler detects that all threads load the same value (see Constant Memory below). Stores to global memory bypass the L1 cache — the latency to shared memory and the L1 cache is 10 - 20 cycles.

Note that the L1 cache is *not* coherent, use the `volatile` keyword if threads in other blocks can modify the contents of a memory location. Otherwise, private data (registers, stack, etcetera) can be used without concern.

The unified coherent L2 cache

Fermi GPUs also have a 768KB unified L2 cache that provides read/write cache memory that is guaranteed to present a coherent view to all SMs in the GPU. In other words, any thread can modify a value held in the L2 cache. At a later time, any other thread on the GPU can read that particular data address and receive the correct, updated value. Of course, atomic updates must be used to guarantee that the store completes before other threads are allowed read access. (The good news is that the L2 cache and Fermi architecture has increased the speed of atomic operations by roughly an order of magnitude.)

Previous GPGPU architectures had challenges with this very common read/update operation because two separate data paths were utilized — specifically the read-only texture load path and the write-only pixel data output path. To ensure data correctness, older GPU architectures required that all participating caches along the read-path be potentially invalidated and flushed after any thread modified the value of an in-cache memory location. The Fermi architecture eliminated this bottleneck with the L2 cache along with the need for the texture and Raster Output (ROP) caches in earlier generation GPUs.

All data loads and stores go through the L2 cache. It is important to note that all global memory transfers always go through the L2 cache *including CPU/GPU memory copies*. The italics stress the fact that host data transfers might unexpectedly affect cache hits and thus application performance.

Wikipedia also notes that, "The quantity of on-board SRAM per ALU actually decreased proportionally compared to the previous G200 generation, despite the increase of the L2 cache from 256kB per 240 ALUs to 768kB per 512 ALUs."

The Impact of Fermi on CUDA memory spaces

The Fermi unified L2 cache and Fermi architectural changes impact CUDA memory spaces in several different ways as is discussed below. Most applications will benefit from these architectural features. The NVIDIA designers recognized that some of these features will decrease the performance of some applications so they gave the programmer the option to turn some features off. In particular, ECC memory and the L1 cache can be disabled as they can adversely impact the performance of some global memory bandwidth limited applications.

David Kirk presented a nice summary of the performance capabilities of a C2050 GPU, which is a GF100 based product:

- 14 multiprocessors.
- Clocks:
  - Core:1150 MHz.
  - Mem:1500 MHz.
- Throughputs:
  - Instruction: 515 Ginstr/s:
    - Fp32: 1030 Gflops/s peak.
    - Fp64: 515 Gflops/s peak.
  - Memory:
    - Shared memory:1030 GB/s aggregate.
    - L1:1030 GB/s aggregate.
    - L2:230 GB/s (not affected by ECC).
    - DRAM:144 GB/s (115 GB/s with ECC).

Global Memory

The Fermi architecture makes some important changes in how CUDA programmers need to think about and use global memory.

From a hardware perspective memory requests are issued in groups of 32 threads (as opposed to 16 in previous architectures), which matches the instruction issue width. Thus, the 32 addresses of a warp should ideally address a contiguous, aligned region to fully utilize the cache line(s).

What this means for CUDA developers:

- 2D/3D thread blocks should be a multiple of 32 "wide".
- Data should be a multiple of 32 in the fastest-varying dimension.

Most applications will benefit from the L1 cache because it performs coalesced global memory loads and stores in terms of a 128 byte cache line size. Once data is inside the cache, applications can reuse data, perform irregular memory accesses, and spill registers without incurring the dramatic slowdown caused by having to rely on roundtrips to the much slower global memory. For this reason, the L1 cache is a very good thing. Note that L1 cache is limited to 16KB/48KB per SM, which is much smaller then the 128 KB register file. This limits the amount of data that can be spilled plus spilled data will evict. For these reasons, spilling data to L1 may help or hinder application performance.

The advantage to disabling the L1 cache is that it avoids use of the 128 byte cache line, permitting 32 byte data accesses. In some situations, an application may stride through, or access memory in such a way that the 128 byte cache line load will waste most of the global memory bandwidth. For these particular applications, disabling the L1 cache will provide a performance benefit.

[Click image to view at full size]

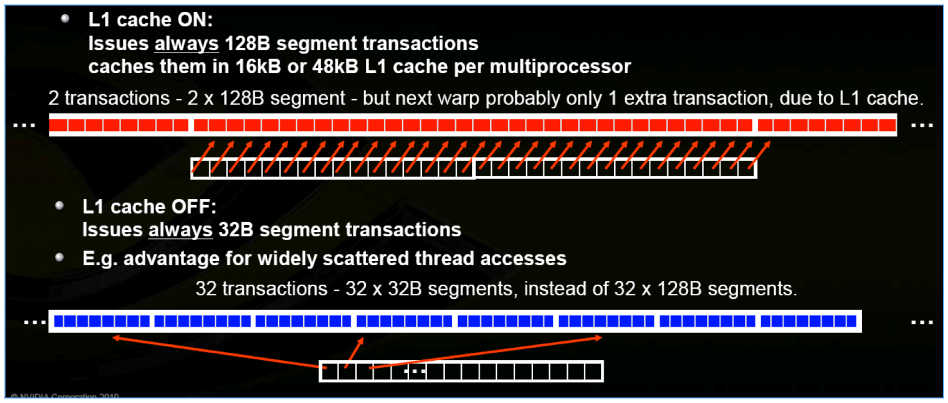

L1 cache ON:
Issues always 128B segment transactions
caches them in 16kB or 48kB L1 cache per multiprocessor
2 transactions - 2 x 128B segment - but next warp probably only 1 extra transaction, due to L1 cache.
L1 cache OFF:
Issues always 32B segment transactions
E.g. advantage for widely scattered thread accesses
32 transactions - 32 x 32B segments, instead of 32 x 128B segments.

Figure 1: source NVIDIA slide 41 from http://people.maths.ox.ac.uk/gilesm/cuda/cudaconf_oxford.pdf

In addition, the caching of local memory in the L1 cannot be disabled (Section 5.3.2.2), but programmers can control local memory usage by limiting the amount of variables that the compiler is likely to place in local memory and by controlling register spilling via the `__launch_bounds()__` attribute (Section B.17) or the `-maxrregcount` compiler option. In particular, it is important to note that `__launch_bounds__` provides the compiler additional information to reduce register usage. Even when the L1 cache is disabled, the stack is always kept in L1 memory.

Since Fermi provides a unified caches, which eliminated the need for separate texture caches, the compute capability 1.X trick of using texture memory to accelerate irregular memory accesses discussed in Part 13 is now discouraged. However, texture units can provide additional computational capability (albeit with 9-bits of accuracy) that can accelerate some applications. One example is Chapter 7, "Leveraging the Untapped Computation Power of GPUs: Fast Spectral Synthesis Using Texture Interpolation" by Richard Townsend, Karthikeyan Sankaralingam, and Matthew D. Sinclair in the recent GPU CUDA Gems book edited by Wen-mei Hwu.

The Fermi memory subsystem provides for 6 partitions of GDDR5 memory with ECC capability on GF100 hardware. There is no longer a linear mapping between addresses and partitions, so typical access patterns are unlikely to all fall into the same partition. This avoids avoid partition camping (bottlenecking on a subset or even a single controller) as discussed in this thread. In addition, ECC can be turned off at the driver level to gain an additional 20% in memory bandwidth and added memory capacity, which can benefit global memory bandwidth limited applications. The Linux `nvidia-smi` command added a `-e` option for controlling ECC. There is a control panel option for Windows.

As discussed next, the Fermi cache hierarchy provides the ability to act like constant memory with the ability to broadcast read-only data.

Constant memory

Constant memory is still an excellent way to store and broadcast read-only data to all the threads on the GPU. Compute capability 2.0 and higher devices allow developers to access global memory with the efficiency of constant memory (e.g., not serializing on the load) under certain conditions where the compiler can recognize and use the *LDU* (LoaD Uniform) instruction. Specifically, the data must:

- Reside in global memory.
- Be read-only in the kernel (programmer can enforce this using the `const` keyword).
- Must not depend on the thread ID.

```
__global__ void kernel( float *g_dst, const float *g_src )
{
g_dst = g_src[0] + g_src[blockIdx.x];
}
```

Essentially a uniform memory access is the same across a thread block. In this case there is no need for `__constant__` declaration and there is no fixed limit to the amount of data as is the case with constant memory. As noted in this thread, constant offset reads from constant memory might still be somewhat faster.

Shared Memory

Shared memory can be either 16KB or 48KB per SM. It is now arranged in 32 banks that are 32-bits wide, which means that working with 64-bit quantities in shared memory no longer causes warp serialization (NVIDIA Programming Guide section G.4.3.2). Previous generation GPUs required a workaround when storing 64-bit quantities in shared memory to avoid warp serialization that required splitting 64-bit data into two 32-bit values, storing them separately in shared memory then merging back to the original form at retrieval. This is no longer necessary on the Fermi, so if you used padding in legacy code to avoid shared memory bank conflicts on GT200/Tesla C1060:

```
__shared tile [16][17];
```

then be sure to change both tile size and padding to warp size for Fermi:

```
__shared tile [32][33];
```

However, the majority of 128-bit memory accesses will still cause a two-way bank conflict in shared memory.

Threads can communicate via shared memory without using `_syncthreads`, if they all belong to the same warp, e.g. `if (tid < 32) { ... }`

On Tesla C1060, a simple declaration was sufficient:

```
__shared__ int cross[32];
```

On C2050 (Fermi), make sure to have `volatile` in front of the shared memory declaration, if it is used for communication between warps in a thread block.

```
volatile __shared__ int cross[32];
```

The reason is:

- C1060 (GT200) could access shared memory directly as operand.
- C2050 (Fermi) uses load/store architecture into registers.

The `volatile` keyword avoids the risk that the compiler may silently cache the previously loaded shared memory value in a register, and fail to reload it again on next reference.

The bandwidth of shared memory is a challenge as noted in Vasily Volkov's GTC talk, "Better Performance at Lower Occupancy." Instead, it is recommended on slide 47 that the CUDA Programming Guide is incorrect - that accessing shared memory is not as fast as accessing a register even in the absence of a bank conflict. Further, this slide notes that the gap between shared memory and register bandwidth has increased with Fermi. Specifically:

- Shared memory bandwidth was more than 3x lower than register bandwidth on pre-Fermi GPUs.
- Shared memory bandwidth is more than 6x lower that register bandwidth on Fermi.

This presentation also notes on slide 50 that the gap between shared memory and arithmetic throughput has increased with Fermi. The data on slide 50 shows:

- G80-GT200: 16 banks vs. 8 thread processors (2:1)
- GF100: 32 banks vs. 32 thread processors (1:1)
- GF104: 32 banks vs. 48 thread processors (2:3)

For this reason, it is recommended that registers be used whenever possible even though the number of available registers has decreased on Fermi GPUs as noted below.

Register and local memory

Fermi supports 32K 32-bit registers that can provide up to 63 registers per thread and 21 if the threads are fully populated (according to David Kirk's August 2010 "CUDA and Fermi Update"). This is confusing because it looks like Fermi with 32k registers provides twice the number of registers (16k) that were available on a GT200 GPU. The reason is that a Fermi streaming multiprocessor is not the same as a G80 or GT200 streaming multiprocessor as is discussed next. In a nutshell:

- The maximum number of possible registers is 63 because that is all the bits available for indexing into the register store.
- If the SM is running 1,536 threads, then only 21 registers can be used due to limitations in the amount of memory available for registers.
- This number degrades gracefully from 63 to 21 as the workload (and hence resource requirements) increases by number of threads.

The good news is that registers always spill to the L1 cache, which will significantly increase application performance that requires local memory. In addition, the stack resides in up to 1KB of the L1 cache.

Fermi dual-issue streaming multiprocessors

Each Fermi streaming multiprocessor (SM) has its own:

- Control unit that decodes and issues instructions and schedules threads.
- Pipelines for integer, fp32, fp64, special function unit (SFU), DirectX 11 instructions.
- Registers and shared memory plus the L1 and constant caches.
- Texture units that utilize L1 cache.

BeHardware has a nice summary of how the SIMD units work together:

- Each of the multiprocessors has a double scheduler that runs at the low frequency and 4 execution blocks at the high frequency:
  - Two 16-way SIMD units (the 32 "cores"): 32 FMA FP32s, 32 ADD INT32s, 16 MUL INT32s, 16 FMA FP64s.
  - A quadruple SFU unit: 4 FP32 special functions or 16 interpolations
  - A 16-way 32-bit Load/Store unit.
- Each of the two 16-way SIMD units are distinct and work on a different instruction and different warp.
  - The first unit can execute 16 FMA FP32s while the second concurrently processes 16 ADD INT32s, which appears to the scheduler as if they executed in one cycle.
  - The quadruple SPU unit is decoupled and the scheduler can therefore send instructions to two SIMD units once it is engaged, which means the

SFUs and SIMDs can be working concurrently. This can be a big win for applications that use transcendental functions.
  - It is not clear how dual-issue scheduling works with double-precision as 64-bit instructions use all the register resources. Thus we don't know if a single SIMD handles all double-precision operations or if two SIMDs run together at half speed.

Issuing instructions

Streaming multiprocessor instructions are issued in-order per warp. A warp consists of 32 consecutive threads that are analogous to a 32-thread vector. Maximum performance occurs when all 32 threads execute the same instruction.

Hardware has been added that handles branching and predication. This means that each thread can execute its own code path. The Fermi whitepaper notes, "Predication enables short conditional code segments to execute efficiently with no branch instruction overhead." There is no performance reduction if all threads in a warp branch together. Concurrently, threads in different warps can take different code paths without reducing performance. As with previous architectures, there is a performance reduction when paths diverge within a warp as each code path within the warp must be taken.

Registers and warp scheduling

Registers and other thread state are partitioned among threads. Unlike a CPU core that can quickly switch between two hyper-threads, GPU multiprocessor may switch between dozens of warps containing many threads. However, a thread will block when an instruction argument is not available.

Understanding register usage and when a thread will block leads to a study of ILP (Instruction Level Parallelism) as opposed to TLP (Thread Level Parallelism). Most CUDA programmers focus on getting the highest possible occupancy to exploit TLP. The idea is to give the scheduler a large amount of threads from which to choose to hide latency and allow the GPU to schedule work to best utilize all available resources.

In contrast, ILP can be used to hide latency and achieve high efficiency at low occupancy and with fewer numbers of threads. The big advantage is more registers per thread. An excellent presentation, "Use registers and multiple outputs per thread" by Vasily Volkov discusses instruction level parallelism and shows that as few as 64 threads are enough to hide latency and maintain performance. A good discussion on this topic occurred on the NVIDIA developer forum here.

The Fermi architecture actually encourages the use of smaller blocks since it can schedule more blocks due to the additional resources per SM. Concurrent kernel execution also means that those smaller blocks don't even need to be from the same kernel (although they do need to be from the same context.) This presents a new way of thinking about problems to achieve both high utilization and performance.

As noted in a very nice presentation for the August 2010 Teragrid conference, "Analysis and Tuning Case Study", slide 71 notes that the benefits of ILP and smaller thread blocks include:

- Fewer threads means more registers per thread.
- More thread blocks per multiprocessor.
  - This implies only two 512-thread blocks as there is a maximum limit of 1024 threads per multiprocessor for a GT200 and 1532 for Fermi.
- An advantage is that doubling the work does not necessarily double register requirement.
  - Fewer registers per thread block implies more thread blocks (e.g. TLP).
- Even if the maximum thread blocks are used, registers are faster than shared memory so use them.

CPU developers will recognize ILP as a form of superscalar execution. Anandtech has a nice discussion on how the GF104 has gone superscalar where they note, "superscalar execution is a method of extracting Instruction Level Parallelism from a thread. If the next instruction in a thread is not dependent on the previous instruction, it can be issued to an execution unit for completion at the same time as the instruction preceding it." This article also provides a good explanation of differences between GF100 and GF104 products.

The challenge for CUDA programmers regardless of their use of ILP or TLP lies in understanding how registers and other resources are allocated on the SM. There isn't much information available to help. One of the best current sources is Coon et al. 2008. (Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators, U.S. Patent No. 7434032.)

Fermi has taken many steps in the right direction including spilling registers to high-speed L1 cache memory and unifying the memory spaces within the GPU. However, the compiler and PTX assembler will likely add unexpected resource consumption. (PTX-compliant binaries 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. Depending on architecture, this JIT compilation can further increase register use.) Experimentation is still the best measure.

The GigaThread engine, concurrent kernels, and host/GPU data transfers

In a nutshell, the Fermi GigaThread engine provides concurrent kernel execution as well as performs simultaneous bidirectional data transfers (reading and writing data at the same time) over the PCIe bus.

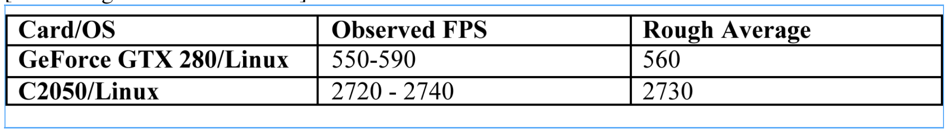
Succinctly, commands are read by the GPU via the host interface. The GigaThread engine then creates and dispatches thread blocks to various SMs. As previously discussed, individual SMs then schedule the warps and distributes work among the ALU (Arithmetic Logic Unit) and other execution units on the streaming multiprocessor.

Thread blocks from the first kernel in the execution queue are launched first. If there additional resources are available, then thread blocks from a kernel in another stream (but same program context) are launched. In an ideal case, two kernels can concurrently run on the same GPU when the sum total of all the resources is less than or equal to the resources available on the GPU. However, few CUDA developers specify their kernel execution configurations to run on half the GPU unless they purposely wish to utilize concurrent kernel execution. A more common scenario occurs when streaming multiprocessors gradually free up as a kernel completes, say if the load is unbalanced. In this case, the Fermi concurrent kernel capability allows some thread blocks from the next queued kernel to "get a head start" by running on the free SMs. This provides better GPU utilization and can reduce time to solution — especially if the currently running kernel has an unbalanced workload.

The GigaThread engine can manage 1,536 simultaneously active threads for each streaming multiprocessor across 16 kernels. Switching from one application to another is about 20 times faster on Fermi than on previous-generation GPUs. This is quick enough to maintain high utilization on a Fermi GPU even when running multiple applications, like a mix of compute code and graphics code. Please see Parts 15, 17, and 18 of this article series for examples of how to do this. Efficient multitasking is important for consumers for video games with physics-based effects and scientists who need to steer and view computationally intensive simulations.

Following are some frame rates for the primitive restart rendering of the virtual terrain example from Part 18, see Table 3.

[Click image to view at full size]



Card/OS	Observed FPS	Rough Average
GeForce GTX 280/Linux	550-590	560
C2050/Linux	2720 - 2740	2730

Table 3

In addition to managing application context switching, the GigaThread engine also provides a pair of streaming data-transfer engines, each of which can fully saturate Fermi's PCI Express host interface. The PCIe bus is bi-directional, so aggregate bandwidth will be doubled. Typically, one engine will be used to move data from host to GPU memory when setting up a GPU computation, while the other will be used to move results from GPU memory back to the host.

IEEE754-2008 arithmetic and atomic operations

Fermi improves the speed and accuracy of double precision calculations. Fully functional GF100 products provide eight times the peak double precision floating point performance over the GT200. Single precision operations have been accelerated and each SM has four Special Function Units (SFUs) that are dedicated to the fast execution of transcendental functions like sine, cosine, reciprocal, and square root.

Scientific and graphics applications benefit from the improved floating-point accuracy -especially when handling very small numbers. In particular Fermi provides better handling of subnormals, which are small numbers that lie between 0 and the absolute smallest normalized number the floating point system supports. Previous architectures rounded subnormals to zero, introducing a loss in precision that manifested itself as artifacts or a loss of detail in graphic applications and a loss of precision in scientific applications. Preserving accuracy benefits graphics applications because it can mean the difference between a protrusion on a demon's face looking like a pimple or a tiny menacing spike. Improved accuracy can prevent introducing non-physical artifacts into scientific calculations. For a general overview, I suggest viewing my *Scientific Computing* column, "Numerical precision: how much is enough?"

The default settings for computation on GPU are now more conservative to support HPC:

- Denormal support.
- IEEE-conformant division and square root.

If your application runs faster on Fermi with `-arch=sm_13` than `-arch=sm_20` then the PTX JIT has used "old" Tesla C1060 settings, which favor speed:

- Flush-to-zero instead denormals.
- No IEEE-precise division, no IEEE-precise square root

For similar results in `-arch=sm_20<`, use: `-ftz=true -prec-div=false -prec-sqrt=false`

Additional numerical speed and accuracy information is contained in the CUDA Programming Guide (v3.2 dated 10/22/2010) sections 5.4.1 and G.2

Fermi now utilizes a fused multiply add (FMA) capability instead of the multiply add (MAD) instruction in previous GPU generations. The reason is that FMA is more precise as it retains full precision in the intermediate state and only rounds at the end. For CUDA programmers, this means that numerical results will change (for the better) on Fermi architectures. The inclusion of FMA instructions also accelerates certain functions such as division and square root.

Atomic operations are nearly an order of magnitude faster on Fermi because of the L2 cache. If a memory address is in the L2 cache, then it is possible read, modify and update the data at that location without requiring a round-trip to and from global memory changes in the L2 cache without going back and forth to global memory. For more information on how to use atomic operation, consult the NVIDIA Programming manual or the Wu-chen Feng and Schucai Xiao paper, "To GPU Synchronize or not GPU Synchronize?"

Summary

The Fermi architecture provides both improved processing power and a more general computational framework that includes a real cache structure, support for ECC memory for increased reliability, greatly improved double precision performance and accuracy.

As they say, ""the proof is in the pudding". The fact that the world'"s largest production environments in animation, supercomputing, and the oil and gas industries are now running large numbers of CUDA-enabled GPGPUs demonstrates the validity of this computational model and hardware. They certainly appear tasty.

With over 250+ million CUDA-enabled GPGPUs already sold and many organizations considering GPGPU technology, knowledgeable GPGPU developers are in a wonderful position. The advent of the Fermi architecture means that GPGPU development has clearly not stagnated and it opens up even more opportunities for the cognoscenti!

It is also worth noting that CUDA is maturing and that pathways are being created to run CUDA on multicore processors and to translate it to OpenCL. Per slide 74 of the Oxford briefing, in Figure 2:

[Click image to view at full size]

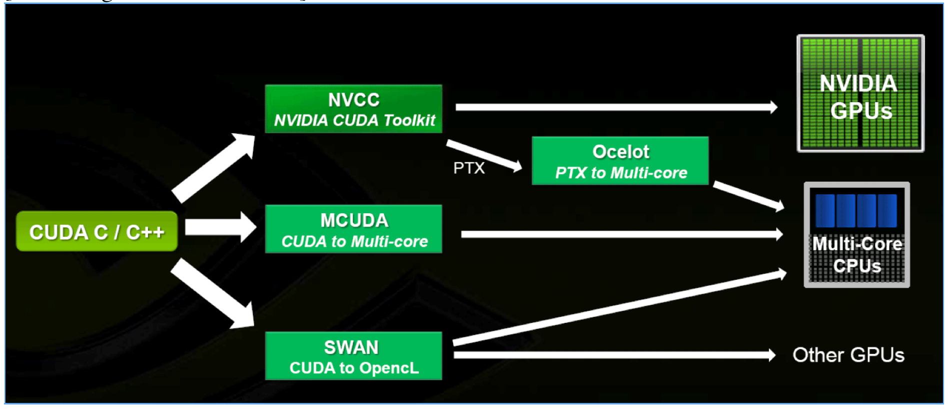


Figure 2: source NVIDIA slide 74 (http://people.maths.ox.ac.uk/gilesm/cuda/cudaconf_oxford.pdf)

References

Mcuda: http://impact.crhc.illinois.edu/mcuda.php

Ocelot: http://code.google.com/p/gpuocelot/

Swan: http://www.multiscalelab.org/swan

- CUDA, Supercomputing for the Masses: Part 20
- CUDA, Supercomputing for the Masses: Part 19
- CUDA, Supercomputing for the Masses: Part 18
- CUDA, Supercomputing for the Masses: Part 17
- CUDA, Supercomputing for the Masses: Part 16
- CUDA, Supercomputing for the Masses: Part 15
- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8
- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1