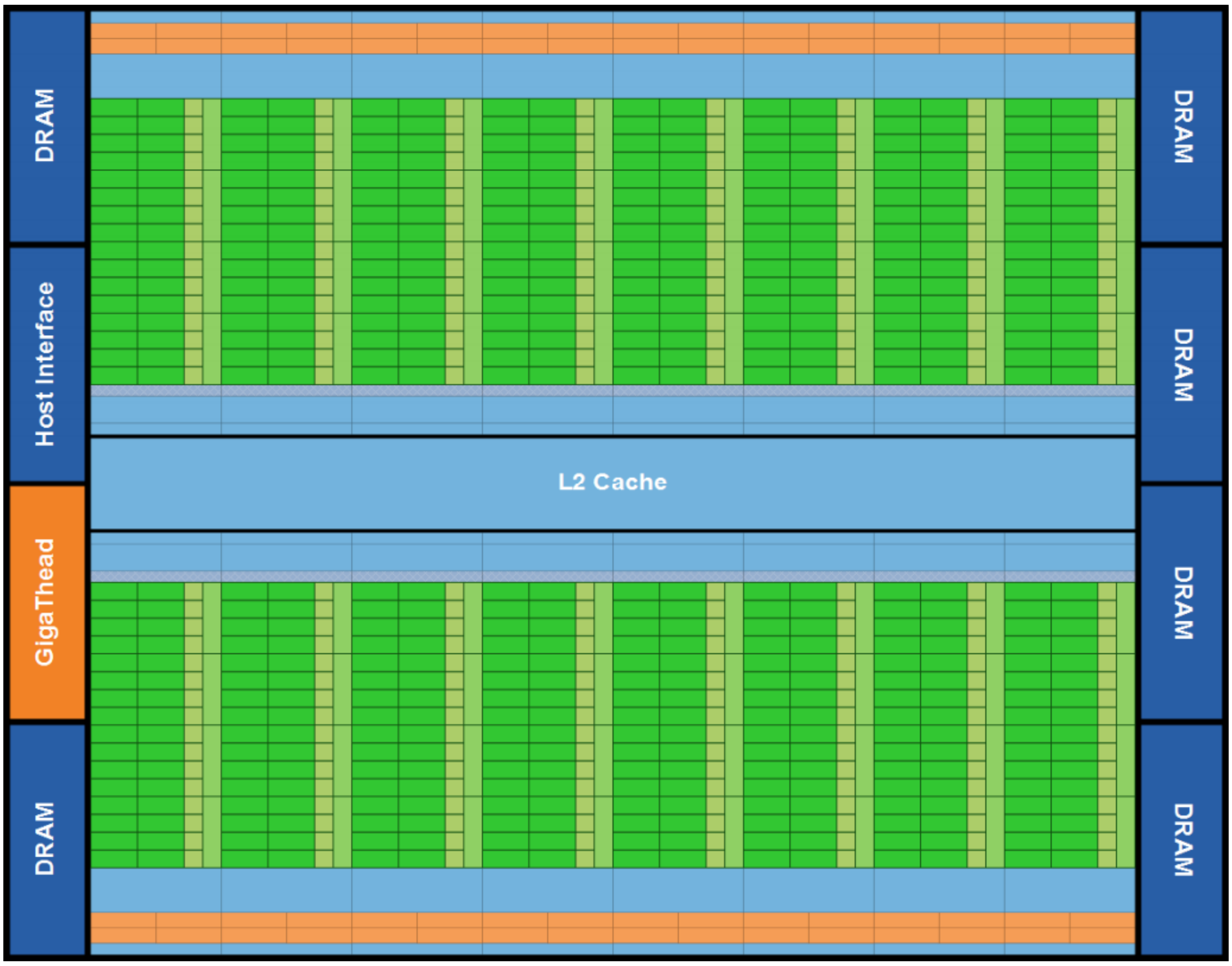
The Fermi architecture

Edit New Page Jump to bottom

Orange Owl edited this page on Apr 22, 2017 · 82 revisions

1. Overview and characteristics

Fermi Graphic Processing Units (GPUs) feature `3.0` billion transistors.



Convention in figures: orange - scheduling and dispatch; green - execution; light blue -registers and caches.

They have up to `512` cores, each with a fully pipelined integer arithmetic logic unit (ALU) and a floating-point unit (FPU) executing one integer and one floating-point instruction per clock cycle, respectively. Each Streaming Multiprocessor (SM) is composed by `32` CUDA cores. A GigaThread globlal scheduler distributes thread blocks to the SM thread schedulers and manages the context switches between threads during execution. The host interface connects the GPU to the CPU via a PCI-Express v2 bus (peak transfer rate of `8GB/s`).

DRAM: supported up to `6GB` of GDDR5 DRAM memory thanks to the `64-bit` addressing capability.

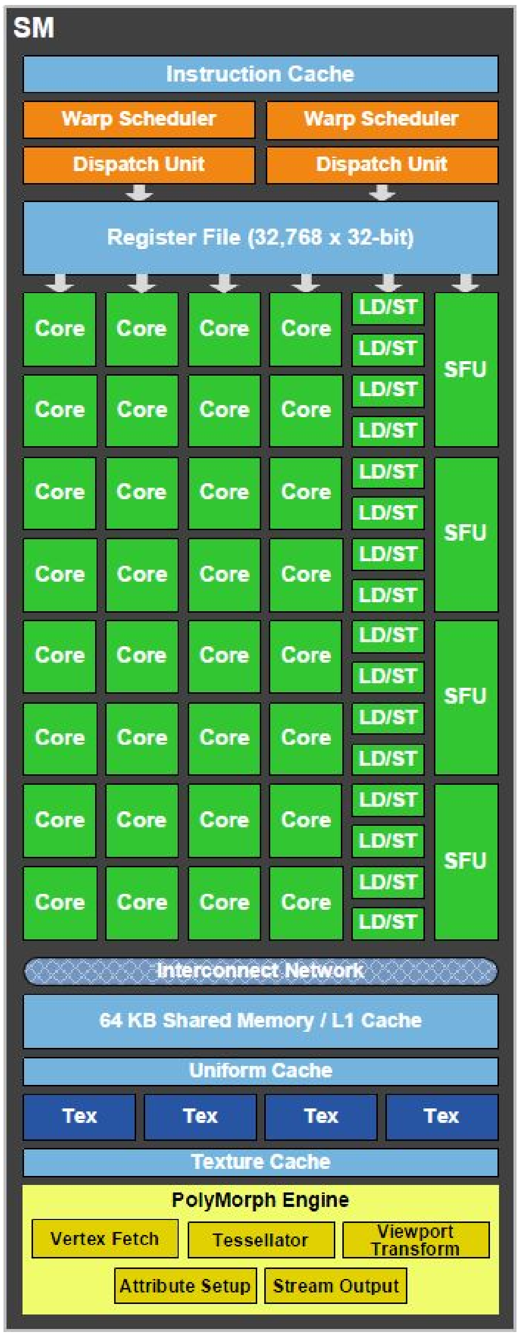
Clock frequency: `1.5GHz` (not released by NVIDIA, but estimated by Insight 64).

Peak performance: the Tesla C2070 card has peak performance of `1030Gflops` in single precision and `515Gflops` in double precision arithmetics.

Global memory clock: `2GHz`.

DRAM bandwidth: `192GB/s`.

2. Streaming Multiprocessor (SM)



Each SM features `32` single-precision CUDA cores, `16` load/store units, `4` Special Function Units (SFUs), a `64KB` block of high speed on-chip memory and an interface to the L2 cache.

2.A. Load/Store Units

Allow source and destination addresses (from/to cache or DRAM) to be calculated for `16` threads per clock. Data can be converted from one format to another (for example, from integer to floating point or vice-versa) as they pass between DRAM or core registers without transfer speed penalties. These format conversion features are examples of optimizations unique to GPUs. They are indeed not implemented in general-purpose CPUs.

2.B. Special Functions Units (SFUs)

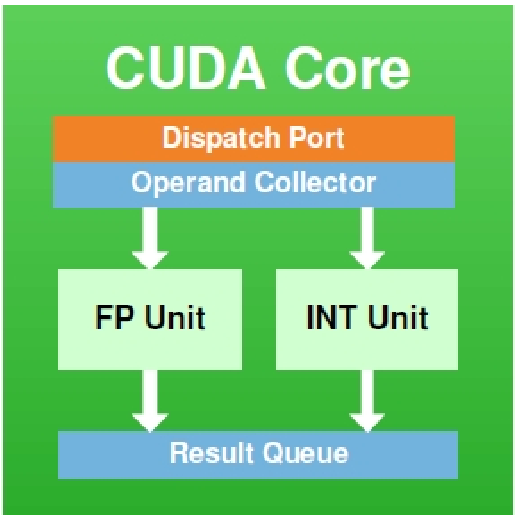
Execute transcendental instructions such as `sin`, `cosine` and square root. The device intrinsics (e.g., `__log2f()`, `__sinf()`, `__cosf()`) expose in hardware the instructions implemented by the SFU. The hardware implementation is based on quadratic interpolation in ROM tables using fixed-point arithmetic.

If `-ffast-math` is passed to `nvcc`, it will automatically use the intrinsic versions of the transcendentals, even if the transcendentals are not explicitly called in their intrinsic versions.

Each SFU can execute one intrinsic instruction per thread per clock cycle so that four of these operations can be issued per cycle in each SM since there are `4` SFUs in each SM. The SFU pipeline is decoupled from the latter, allowing the dispatch unit to issue other instructions to other execution units while the SFU is occupied. It should be noticed that CUDA intrinsics rarely map to a single SFU instruction, but usually map to sequences of multiple SFU and non-SFU instructions. Different GPU have different throughputs for the various operations involved, so if one needs to know the throughput of a particular intrinsic on a particular GPU it would be best to simply measure it. The performance of single-precision intrinsics can also vary with compilation mode, in particular depending on whether `-ftz={true|false}`.

3. CUDA core

A CUDA core handles integer and floating point operations (see Fig. `3`).



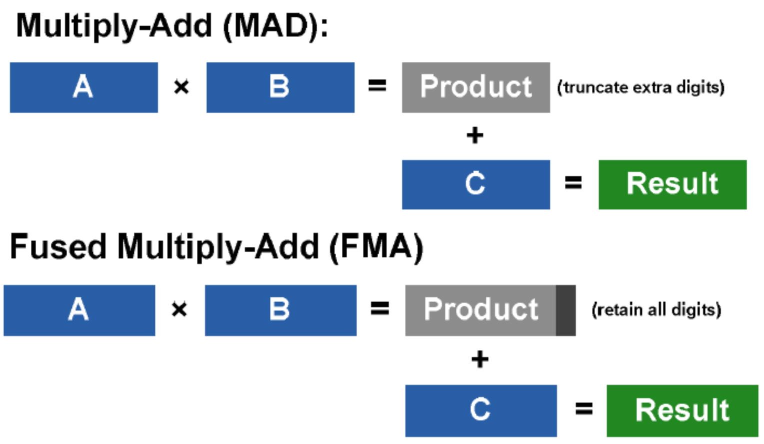
3.A. Integer Arithmetic Logic Unit (ALU)

Supports full `32-bit` precision for all usual mathematical and logical instructions, including multiplication, consistent with standard programming language requirements. It is also optimized to efficiently support `64-bit` and extended precision operations.

3.B. Floating Point Unit (FPU)

Implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetics. Up to `16` double precision fused multiply-add operations can be performed per SM, per clock.

4. Fused Multiply-Add



Fused Multiply-Add (FMA) performs Multiply-Add (MAD) operations (i.e., `A*B+C`) with a single final rounding step, with no loss of precision in the addition, for both `32-bit` single-precision and `64-bit` double-precision floating point numbers. FMA improves the accuracy upon MAD by retaining full precision in the intermediate stage. 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. Prior GPUs accelerated these calculations with the MAD instruction (multiplication with truncation, followed by an addition with round-to-nearest even) that allowed both operations to be performed in a single clock. A FMA counts as two operations when estimating performance, resulting in a peak performance rate of `1024` operations per clock (`32 cores x 16 SM x 2 operations`).

5. Rounding and subnormal numbers

Subnormal numbers are small numbers that lie between zero and the smallest normalized number of a given floating point number system. In the Fermi architecture, single precision floating point instructions support subnormal numbers by default in hardware, allowing values to gradually underflow to zero with no performance penalty, as well as all four `IEEE 754-2008` rounding modes (nearest, zero, positive infinity, and negative infinity).

This is a relevant point since prior generation GPUs flushed subnormal operands and results to zero, incurring in losses of accuracy, while CPUs typically perform subnormal calculations in exception-handling software, taking thousands of cycles.

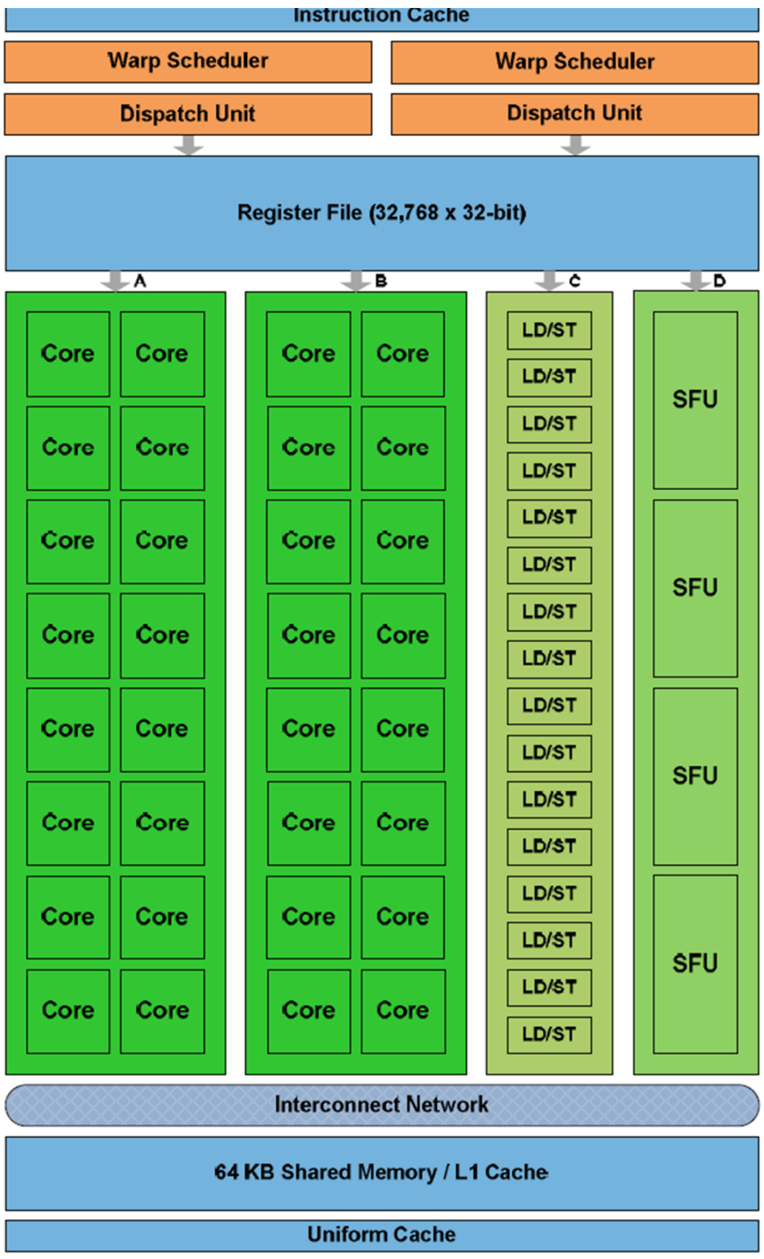
6. Warp scheduling

The Fermi architecture uses a two-level, distributed thread scheduler.

The GigaThread engine distributes thread blocks to the various schedulers of the SMs, while at the SM level, each warp scheduler distributes warps of `32` threads to the SM execution units.

Each SM features two warp schedulers and two instruction dispatch units. When a thread block is assigned to an SM, all threads in a thread block are divided into warps. The two warp schedulers select two warps and issue one instruction from each warp to a group of `16` CUDA cores, `16` load/store units, or `4` special function units. The Fermi architecture, compute capability `2.x`, can simultaneously handle `48` warps per SM for a total of `1,536` threads resident in a single SM at a time.

Fermi also supports concurrent kernel execution: multiple kernels launched from the same application context can be executed on the same GPU at the same time. Concurrent kernel execution allows programs that execute a number of small kernels to fully utilize the GPU.



Source: NVIDIA

Each SM can issue instructions consuming any two of the four green execution columns shown in the figure. For example, the SM can mix `16` operations from the `16` first column cores with `16` operations from the `16` second column cores, or `16` operations from the load/store units with four from SFUs, or any other combinations the program specifies.

Note that `64-bit` floating point operations consume both the first two execution columns. This implies that an SM can issue up to `32` single-precision (`32-bit`) floating point operations or 16 double-precision (`64-bit`) floating point operations at a time.

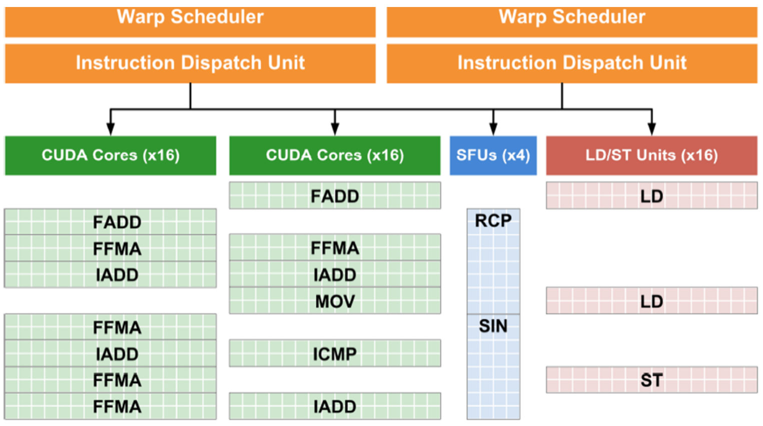
6.A Dual Warp Scheduler

Threads are scheduled in groups of `32` threads called warps. Each SM features two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently. The dual warp scheduler selects two warps, and issues one instruction from each warp to a group of `16` cores, `16` load/store units, or `4` SFUs.

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.

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.



7. Context switching

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 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.

Switching from one application to another takes just 25 microseconds. 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). As mentioned, this switching is managed by the GigaThread hardware thread scheduler.

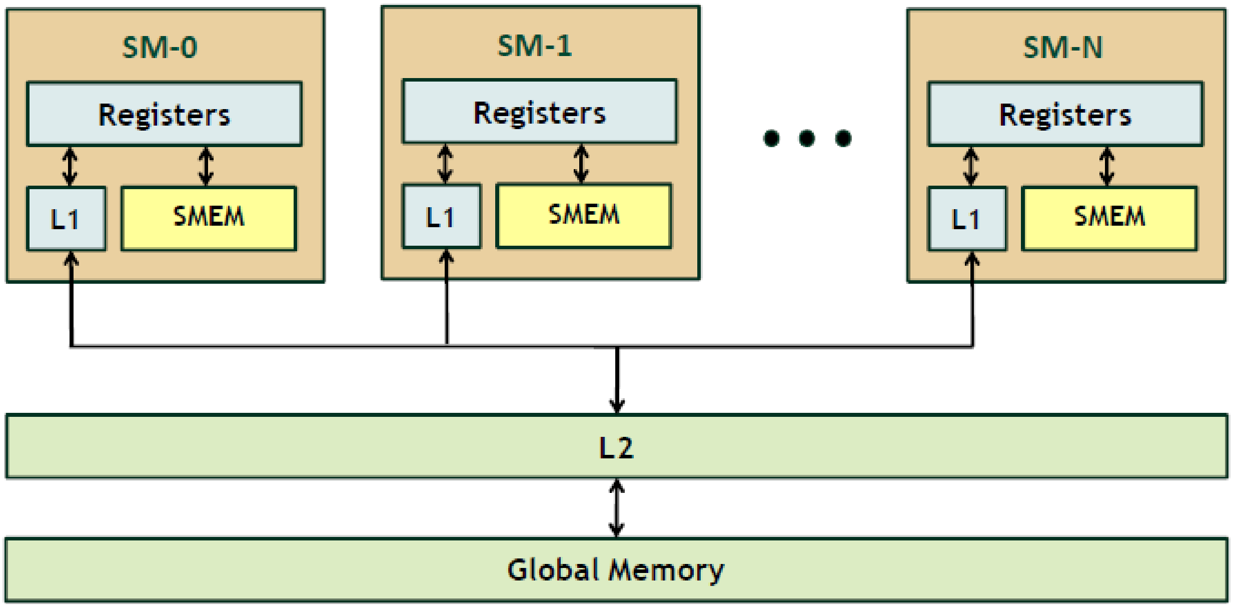
8. Memory

L1 cache per SM and unified L2 cache that services all operations (load, store and texture). Register files, shared memories, L1 caches, L2 cache, and DRAM memory are Error Correcting Code (see subsection on the Error Correcting Code) protected.

8.A Load/Store Units

Each multiprocessor has 16 load/store units, allowing source and destination addresses to be calculated for 16 threads (a half-warp) per clock cycle.

8.B Registers



Each SM has `32KB` of registers. Each thread has access to its own registers but not to those of other threads. The maximum number of registers that can be used by a CUDA kernel is `63`. The number of available registers gradually degrades from `63` to `21` as the workload (and hence resource requirements) increases by number of threads. Registers have a very high bandwidth: about `8,000 GB/s`.

8.C L1 + Shared Memory

On-chip memory that can be used either to cache data for individual threads (register spilling/L1 cache) and/or to share data among several threads (shared memory). This `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. CUDA provides a runtime API that can be used to adjust the amount of shared memory and L1 cache. Prior generation GPUs spilled registers directly to DRAM, increasing access latency. Also, 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 provides low-latency access (`10-20` cycles) and very high bandwidth (`1,600 GB/s`) to moderate amounts of data. Because the access latency to this memory is also completely predictable, algorithms can be written to interleave loads, calculations, and stores with maximum efficiency.

Algorithms for which data addresses are not known in advance, as physics solvers, ray tracing and sparse matrix multiplication especially benefit from this cache hierarchy level.

8.D Local memory

Local memory is meant as a memory location used to hold "spilled" registers. Register spilling occurs when a thread block requires more register storage than is available on an SM. Pre-Fermi GPUs spilled registers to global memory, causing a dramatic drop in performance. Compute `>=2.0` devices spill registers to the L1 cache first, which minimizes the performance impact of register spills. Register spilling increases the importance of the L1 cache. Pressure from register spilling and the stack (which consume L1 storage) can increase the cache miss rate by data eviction. Local memory is used only for some automatic variables (which are declared in the device code without any of the `__device__`, `__shared__`, or `__constant__` qualifiers). Generally, an automatic variable resides in a register except for the following:

- Arrays that the compiler cannot determine are indexed with constant quantities;
- Large structures or arrays that would consume too much register space;
- Any variable the compiler decides to spill to local memory when a kernel uses more registers than are available on the SM.

The `nvcc` compiler reports total local memory usage per kernel (lmem) when compiling with the `-ptxas-options=-v` option.

8.E L2 cache

`768 KB` unified L2 cache, shared among the `16 SMs`, that services all load and store from/to global memory, including copies to/from CPU host, and also texture requests. As an example, if one copies `512KB` from CPU to GPU, those data will reside both in the global memory and in L2; a kernel needing those data immediately after the CPU->GPU copy will find them in L2. The L2 cache is thus the memory where the SMs can exchange the data with high speed. The L2 cache subsystem also implements atomic operations, used for managing access to data that must be shared across thread blocks or even kernels.

Filter and convolution kernels that require multiple SMs to read the same data benefit from this cache hierarchy level.

8.F Global memory

Accessible by all threads as well as host (CPU). High latency (`400-800` cycles).

8.G Error Correcting Code

The Fermi architecture supports the 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. ECC checkbit fetches from DRAM necessarily consume some amount of DRAM bandwidth, resulting in performance difference between ECC-enabled and ECC-disabled operations. The effect is of course more relevant for memory bound applications. Register files, shared memories, L1 cache, L2 cache and DRAM memory are protected by a Single-Error Correct Double-Error Detect (SECDED) ECC code.

9. References

[1] "NVIDIA’s Next Generation CUDA Compute Architecture: Fermi."

[2] N. Brookwood, "NVIDIA Solves the GPU Computing Puzzle."

[3] P.N. Glaskowsky, "NVIDIA’s Fermi: The First Complete GPU Computing Architecture."

[4] N. Whitehead, A. Fit-Florea, "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs," 2011

[5] S.F. Oberman, M. Siu, "A high-performance area-efficient multifunction interpolator," Proc. of the 17th IEEE Symposium on Computer Arithmetic, Cap Cod, MA, USA, Jul. 27-29, 2005, pp. 272–279.

[6] R. Farber, "CUDA Application Design and Development," Morgan Kaufmann, 2011.

[7] P. Micikevicius, "Local Memory and Register Spilling"

General CUDA programming