Towards Microarchitectural Design of Nvidia GPUs — [Part 1]

Dung Le · Follow
Published in Distributed Knowledge
13 min read · Apr 28, 2020

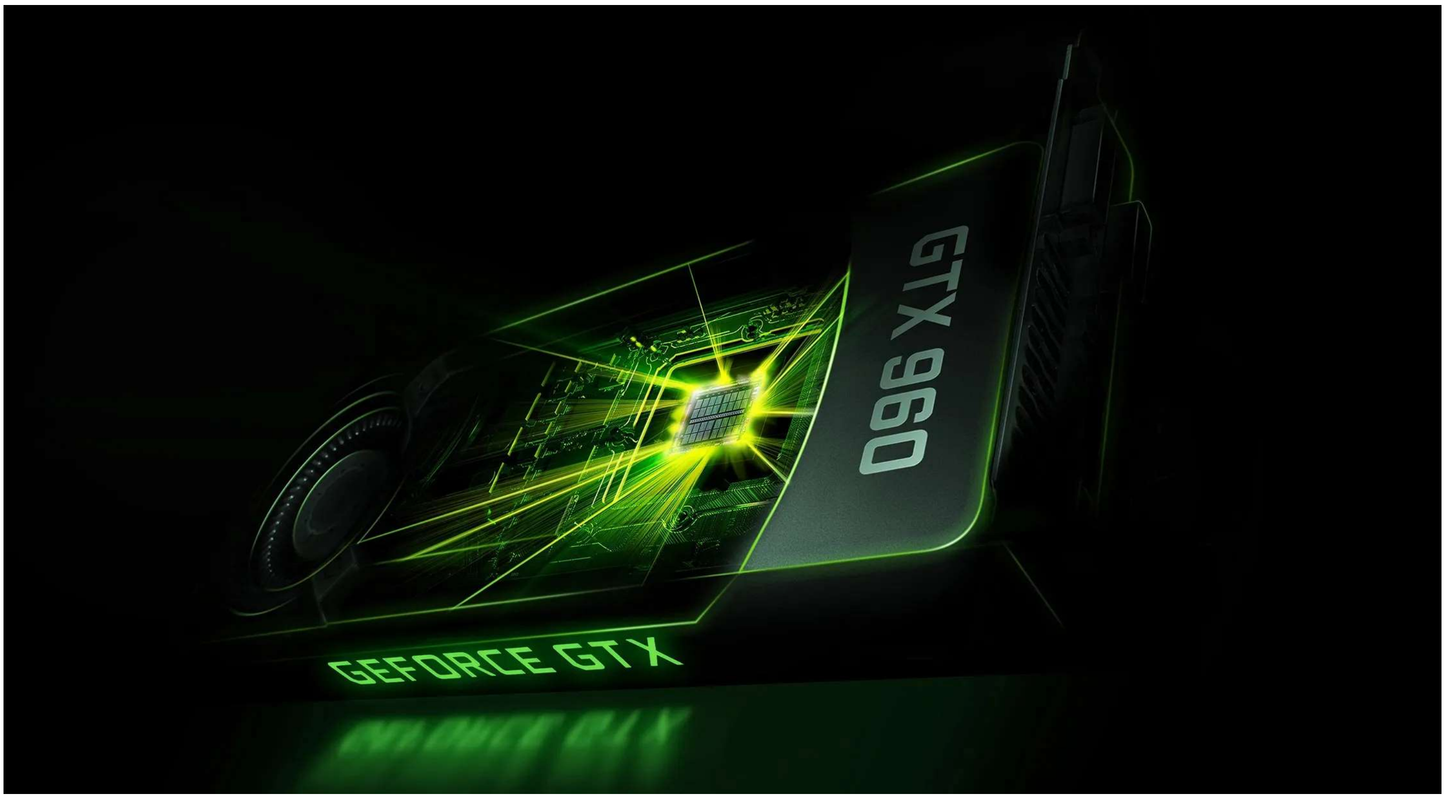


Figure 1: GeForce GTX 960. Source: Nvidia

There is no question within the Deep Learning community about Graphics Processing Unit (GPU) applications and its computing capability. From zero to hero, it can save your machine from smoking like a marshmallow roast when training DL models to transform your granny "1990s" laptop into a mini-supercomputer that can supports up to 4K streaming at 60 Frames Per Second (FPS) or above with little-to-no need to turn down visual settings, enough for the most graphically demanding PC games. However, stepping away from the hype and those flashy numbers, little do people know about the underlying architecture of GPU, the "pixie dust" mechanism that lends it the power of a thousand machines.

This series aims to outline all of the key architectural design and analysis of Nvidia GPUs, comprised of official documentation from Nvidia itself and academic resources. My target is to keep the information **short, relevant, and focus** on the most important topics which are absolutely required to be understood.

After finishing reading this series, you will be able to have a deep understanding of GPU Microarchitectures at system-level design.

The content of this blog will be organized as follows:

1. CPU vs GPU
2. CUDA scalable parallel architecture
3. Pascal Architecture
4. Take away
5. What's next?
6. References

CPU vs GPU

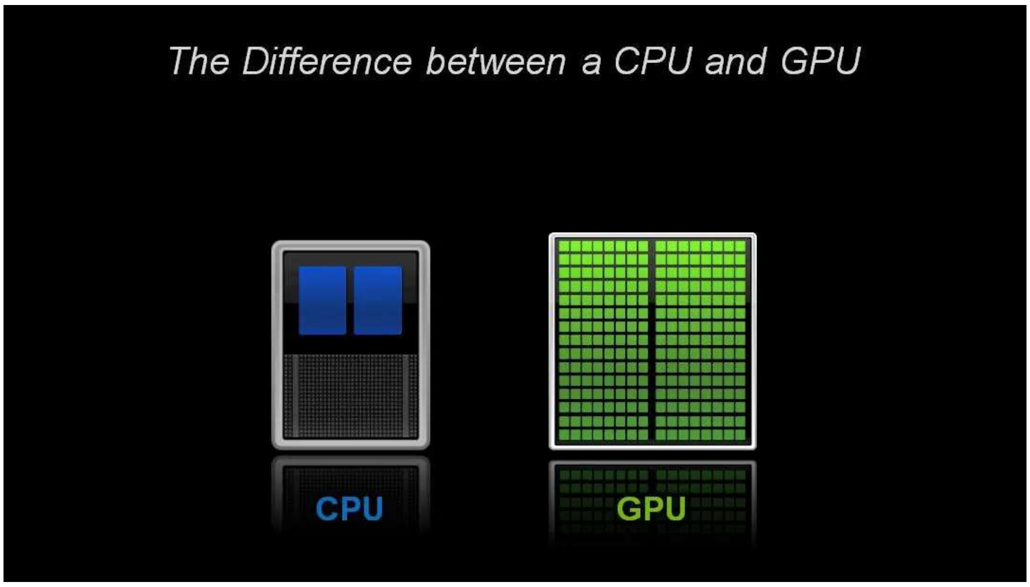


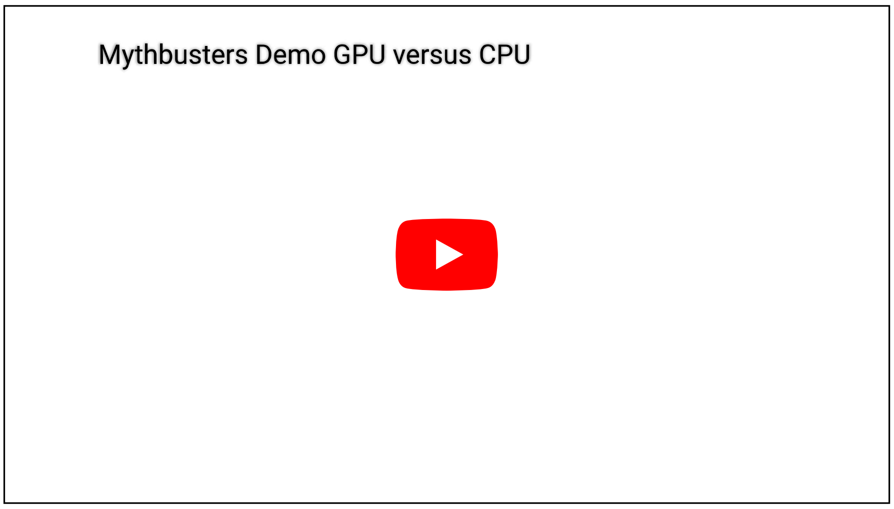
Figure 2: CPU and GPU Architectures. Source: Nvidia blog

Architecturally, the Central Processing Unit (CPU) is composed of just a few cores with lots of cache memory while a GPU is composed of hundreds of cores [6]. This difference in architectural design leads to different approaches in which they process their tasks.

Modern CPUs strongly favor lower latency of operations with clock cycles in the nanoseconds, optimized for sequential serial processing. They are designed to maximize the performance of a single task within a job; however, the range of tasks is wide. On the other hand, GPUs work best on problem sets that are ideally solved using massive fine-grained parallelism with thousands of smaller and more efficient cores, aiming at handling multiple functions at the same time for high work throughput. Modern GPUs provide superior processing power, memory bandwidth, and efficiency over their CPU counterparts. They are 50–100 times faster in tasks that require multiple parallel processes, such as machine learning and big data analysis.

One notable example where massive fine-grain parallelism is needed is high-resolution graphics processing. Let's take an example of continuously displaying 4096 x 2160 pixels/image for 60 FPS in 4K video, where each thread's job is to render a pixel. In this example, an individual task is relatively small and often a set of tasks is performed on data in the form of a pipeline. It's obvious that from this case that **the throughput of this pipeline is more important than the latency of the individual operations,** since we would prefer to have all pixels rendered to form a complete image with slightly higher latency rather than having a quarter of an image with lower latency. Because of its focus on latency, the generic CPU underperformed GPU, which was focused on providing a very fine-grained parallel model with processing organized in multiple stages where the data would flow through.

Here's a video that I found in Nvidia blog which's quite informative in understanding the fundamental difference between CPU and GPU demonstrated in the above example:



Video 1: CPU and GPU Comparison Metaphor. Source: Nvidia

CUDA scalable parallel architecture

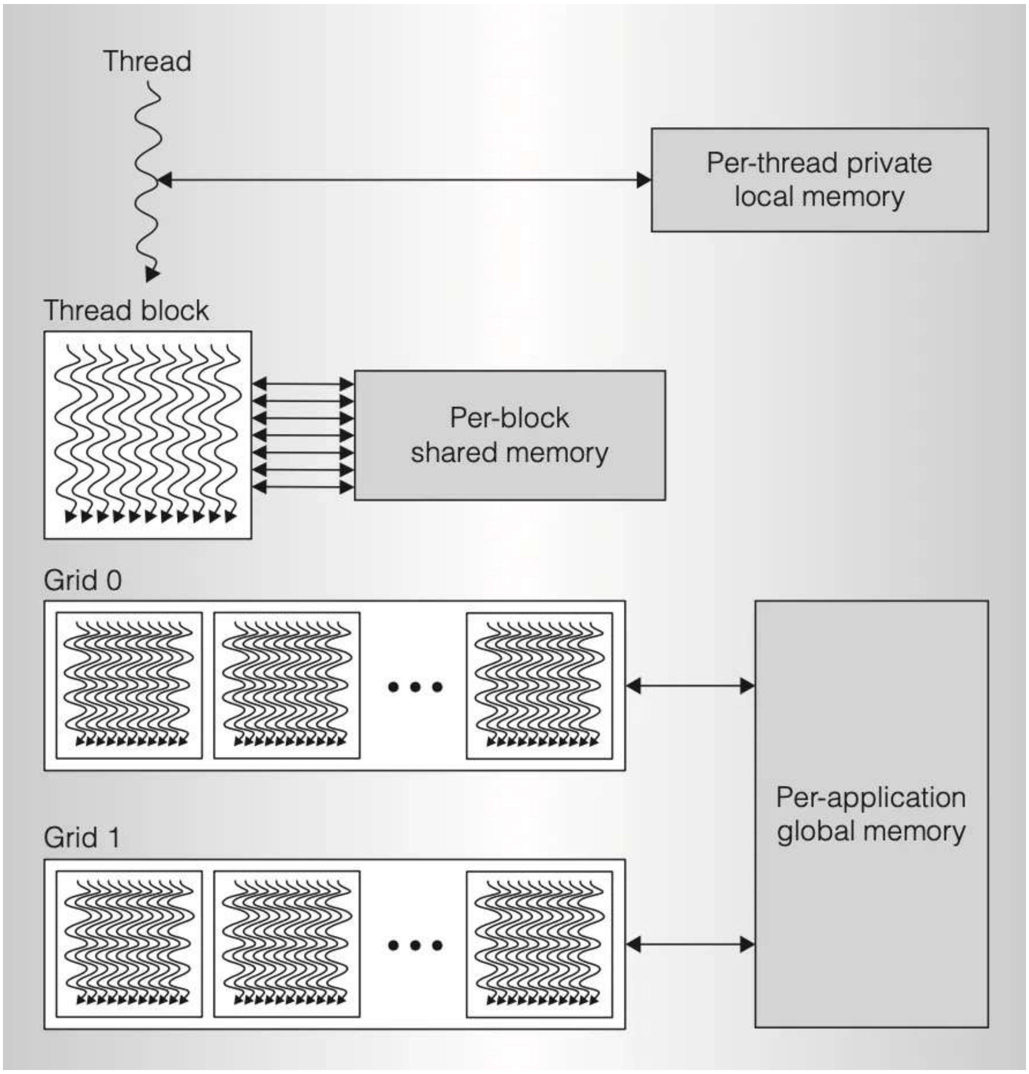

Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

Figure 3: CUDA Architecture hierarchy of threads, thread blocks, and grids of blocks

Before diving deep into GPU microarchitectures, let's familiarize ourselves with some common terminologies that will henceforth be used in this series. In GPU microarchitecture, a **host** means CPU, a **device** means GPU, and a **kernel** acts as a function that runs on the device.

A CUDA program comprises of a host program, consisting of one or more sequential threads running on a host, and one or more parallel kernels suitable for execution on a parallel computing GPU. Only one kernel is executed at a time, and that kernel is executed on a set of lightweight parallel threads. For better resource allocation (avoid redundant computation, reduce bandwidth from shared memory), threads are grouped into thread blocks. A **thread block** is a programming abstraction that represents a group of threads that can be executed serially or in parallel.

Multiple thread blocks are grouped to form a **grid**. Threads from different blocks in the same grid can coordinate using atomic operations on a global memory space shared by all threads. Sequentially dependent kernel grids can synchronize through global barriers and coordinate through global shared memory. Thread blocks implement coarse-grained scalable data parallelism and provide task parallelism when executing different kernels, while lightweight threads within each thread block implement fine-grained data parallelism and provide fine-grained thread-level parallelism when executing different paths.

Pascal Architecture

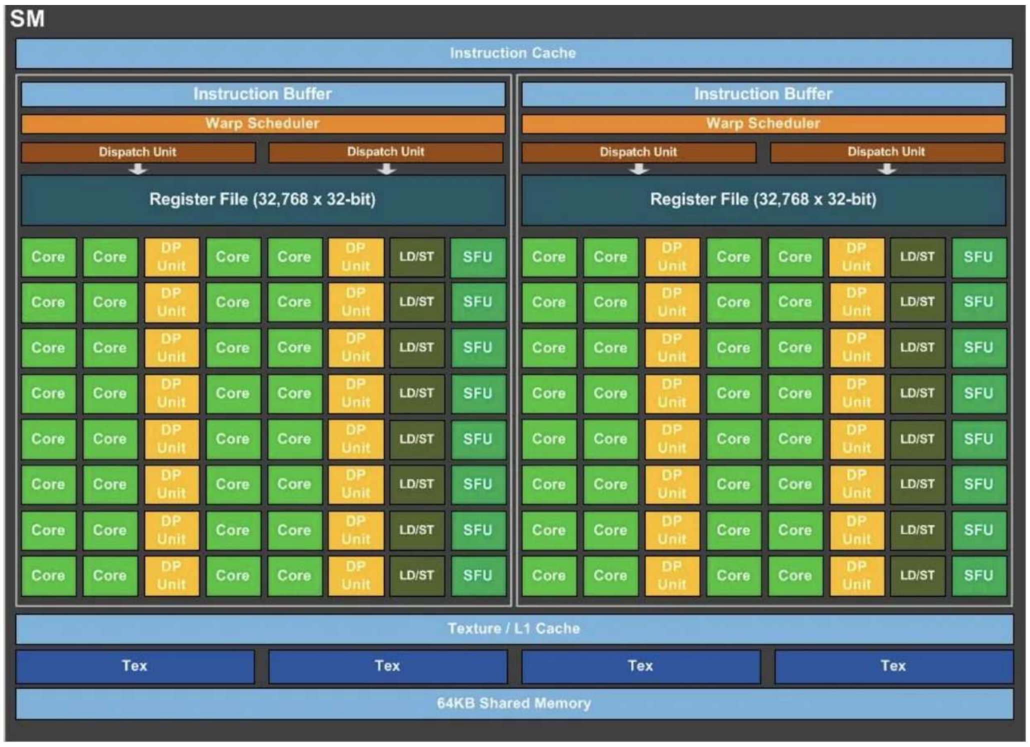


Figure 3: Pascal GPU computing microarchitecture. Source: Nvidia

Figure 3 illustrates the third-generation Pascal computing architecture on Geforce GTX 1080, configured with 20 **streaming multiprocessors** (SM), each with 128 CUDA processor **cores**, for a total of 2560 cores. The GigaThread work scheduler distributes CUDA thread blocks to SMs with available capacity, balancing load across GPU, and running multiple kernel tasks in parallel if appropriate. The multithreaded SMs schedule and execute CUDA thread blocks and individual threads. Each SM can process multiple concurrent threads to hide long-latency loads from DRAM memory. Each thread block completed executing its kernel program and released its SM resources before the work scheduler assigns a new thread block to that SM. A block is assigned to and executed on a single SM.

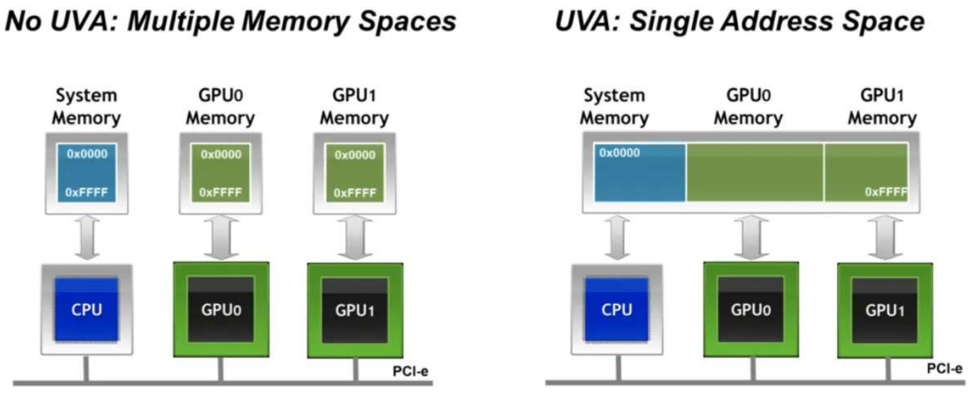


Figure 4: PCI-e connects CPU and its system memory to GPUs and their DRAM memories. Source: Nvidia

The GPUs and their DRAM memories are connected with the host CPU system memory using the **PCIe host interface.** The CPU+GPU coprocessing and data transfer use the directional PCIe interface. The SM threads access system memory and CPU threads access GPU DRAM memory using the PCIe interface.

Cached memory hierarchy

Each SM has an L1 cache, and the SMs share a common 768-Kbyte unified L2 cache. The L2 cache connects with six 64-bit DRAM interfaces and the PCIe interface, which connects with the host CPU, system memory, and PCIe devices. It caches DRAM memory locations and system memory pages accessed through the PCIe interface and responds to load, store, atomic, and texture instruction requests from the SMs and requests from their L1 caches.

Streaming multiprocessor

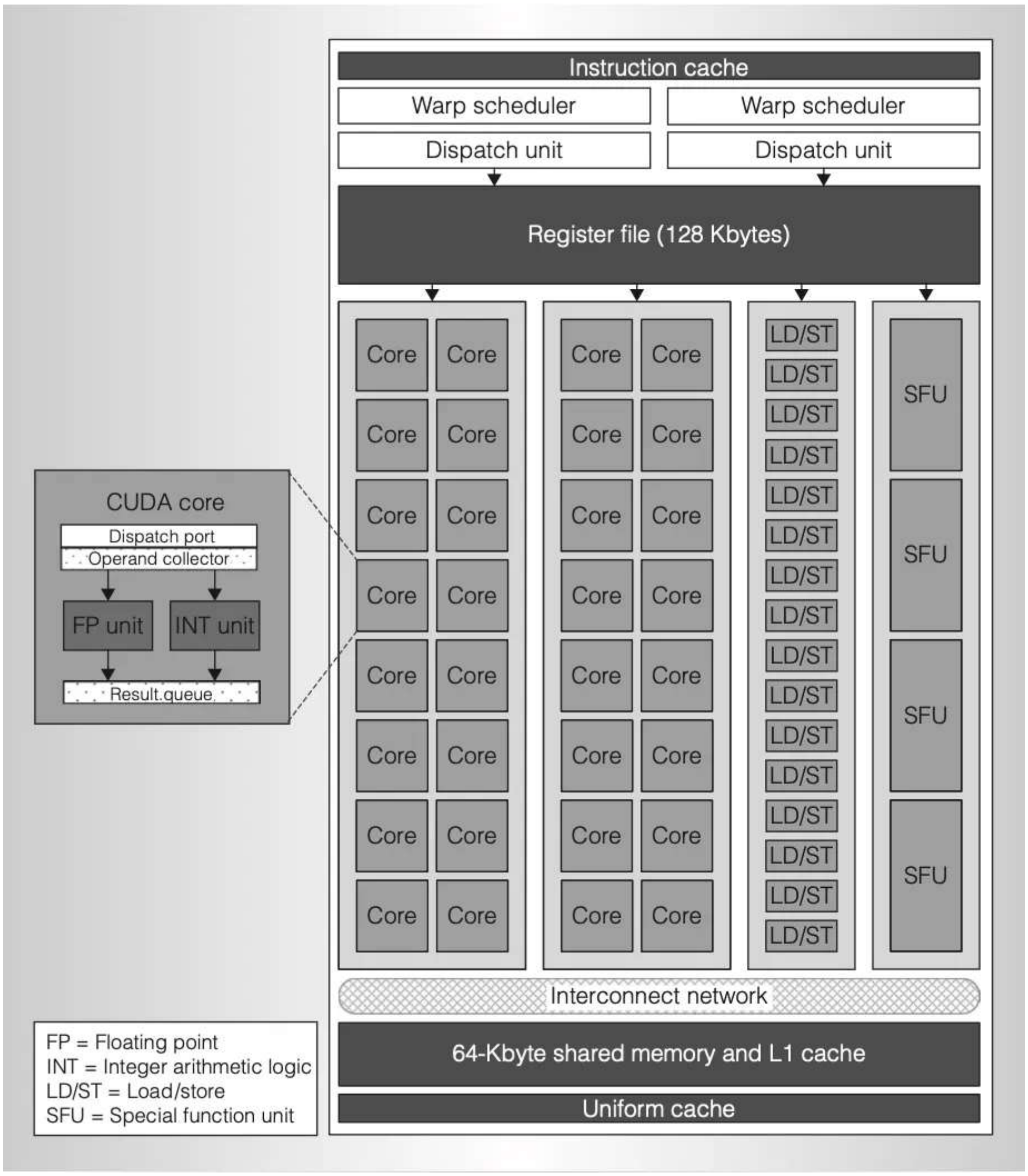


Figure 5: Fermi SM Architecture. Source: [1]

Fermi SM is designed with several architectural features to deliver higher performance and improve its programmability and applicability. Each SM includes 32 CUDA processor cores, 16 load/ store units, and four special function units (SFUs). It also possesses a 64-Kbyte configurable shared memory+L1 cache, 128-Kbyte register file, instructions cache, and two multi-threaded wrap schedulers and two instruction dispatch units.

As stated above, each SM can process up to 1536 concurrent threads. In order to efficiently managed this many individual threads, SM employs the **single-instruction multiple-thread** (SIMT) architecture. The SIMT instruction logic creates, manages, schedules, and executed concurrent threads in groups of 32 parallel threads, or

warps. A thread block can have multiple warps, handled by two warp schedulers and two dispatch units. A scheduler selects a warp to be executed next and a dispatch unit issues an instruction from the warp to 16 CUDA cores. 16 load/store units, or four SFUs. Since the warps operate independently, each SM can issue two warp instructions to the designated sets of CUDA cores, doubling its throughput.

Threads

Threads in SM are independent by nature. Each has its own private registers, predicates, private per-thread memory & stack frame, instruction address, and thread execution state. SIMT instructions control the execution of an individual thread, including arithmetic, memory access, and branching and control flow instructions. For efficiency, the SIMT multiprocessor issues an instruction to a warp of 32 independent parallel threads. Threads in a single warp can only run 1 set of instructions at once.

Within a warp, it is optimal for performance when all of its threads execute the same path. If there's any divergence caused by a data-dependent conditional branch (if, ...), execution serialization for each branch path is taken, and all threads are synchronized to the same execution path when their diverged paths complete.

CUDA cores

Each pipelined CUDA core executes an instruction per clock for a thread. With 32 cores architecture, an SM can execute up to 32 thread instructions per clock. Executable instructions include scalar floating-point instruction, implemented by floating-point unit (**FP unit**), and integer instruction, implemented by integer unit (**INT unit**).

SFUs

The SFUs are in charge of executing 32-bit floating-point instructions for fast approximations of reciprocal, reciprocal square root, sin, cos, exp, and log functions. The approximations are precise to better than 22 mantissa bits.

Load/store units

The streaming multiprocessor load/store units execute load, store, and atomic memory access instructions. A warp of 32 active threads presents 32 individual byte addresses, and the instruction accesses each memory address. The load/store units coalesce 32 individual thread accesses into a minimal number of memory block accesses.

Fermi implements a unified thread address space that accesses the three separate parallel memory spaces: per- thread-local, per-block shared, and global memory spaces. A unified load/store instruction can access any of the three memory spaces, steering the access to the correct memory of the source/ destination, before loading/storing from/to cache or DRAM. Fermi provides a terabyte 40-bit unified byte address space, and the load/store ISA supports 64-bit byte addressing for future growth. The ISA also provides 32-bit addressing instructions when the program can limit its accesses to the lower 4 Gbytes of address space [1].

Configurable shared memory and L1 cache

On-chip shared memory provides low- latency, high-bandwidth access to data shared to co-operating threads in the same CUDA thread block. Fast shared memory significantly boosts the performance of many applications having predictable regular addressing patterns, while reducing DRAM memory traffic.

Fermi introduces a configurable-capacity L1 cache to aid unpredictable or irregular memory accesses, along with a configurable- capacity shared memory. Each streaming multiprocessor has 64 Kbytes of on-chip memory, configurable as 48 Kbytes of shared memory and 16 Kbytes of L1 cache, or as 16 Kbytes of shared memory and 48 Kbytes of L1 cache.

GPU memory model

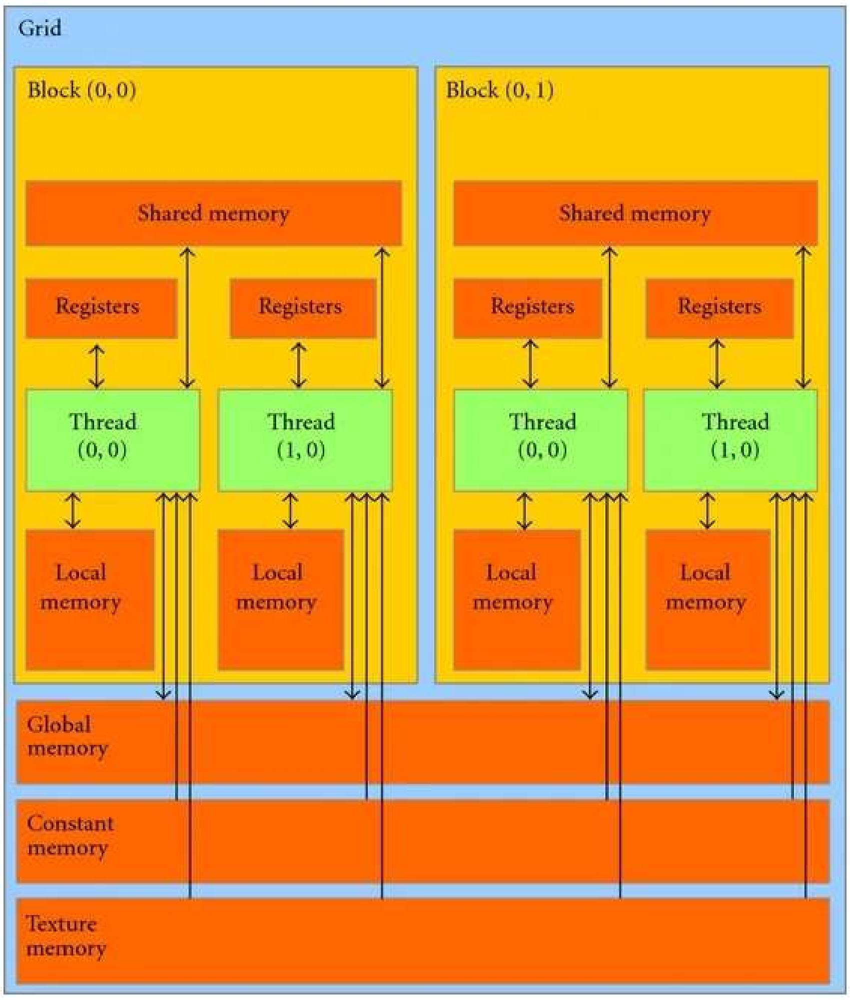


Figure 6: GPU Memory Model. Source: ResearchGate

GPU memory is broken down into 8 parts: Registers, Local memory, Global Memory, Shared memory, L1/L2 cache, Constant memory, Texture memory, Read-only cache.

Registers

Registers are the fastest forms of memory on the multi-processor, about 10x faster than shared memory. There are tens of thousands of registers in each SM, and generally, each thread can declare a maximum of 63 32-bit registers. Most stack variables declared in kernels are stored in registers, such as float x, int y, double z; statically indexed arrays stored on the stack are also sometimes put in registers.

Registers can only be accessed by the thread that creates them. They only exist during the lifetime of the thread.

Local memory

Local memory (LMEM) a GPU thread resides in the global memory and can be 150x slower than register or shared memory. It refers to memory where registers and other thread data is spilled, usually when one runs out of SM resources.

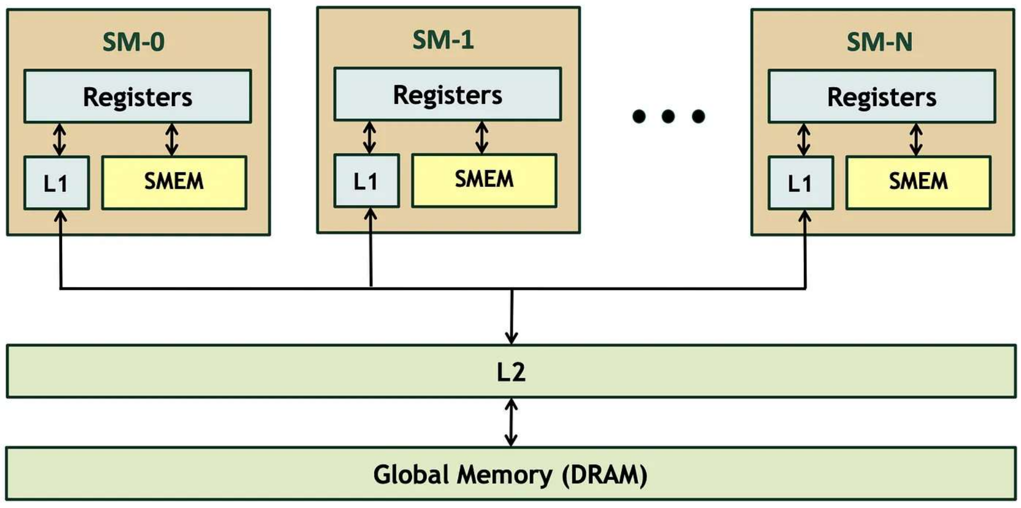


Figure 7: Fermi Memory Hierarchy. Source: [4]

LMEM can issue two access operations: store to write data, and load to read data. The store operation, when issued, writes a line to L1, propagated its write to L2 if the line is evicted from L1. The line could also be evicted from L2, in which case it's written to DRAM. The load operation requests the line from L1. If it's a hit, the operation is complete, else it then requests the line from L2, or DRAM if L2 is again a miss.

Global memory

The vast majority of GPU's memory is global memory. GPUs have .5–24GB of global memory, with most now having ~2GB. Global memory exhibits a potential 150x slower latency of ~600 ns on Fermi than that of registers or shared memory, especially underperforming for uncoalesced access patterns.

Let's take a step back to explain the previous point a bit. Perhaps from your Computer Architecture or OS class, you have familiarized yourself with the mechanism of cache lines, which is how extra memory near the requested memory is read into a cache improves cache hit ratio for subsequent accesses. For uncoalesced reads and writes, the chance of subsequent data to be accessed is unpredictable, which causes the cache miss ratio is expectedly high, requiring the appropriate data to be fetched continuously from the global memory with high latency. This overall degrades GPU performance and makes global memory access a huge application bottleneck.

Shared memory

This type of memory is located in the SM and has low latency (~5ns). It shares the same hardware as L1 cache, with ~48 Kbyte of memory (varies per CPU). The shared memory is used within the scope of the block, allowing that block's threads to synchronize and coordinate with each other. Because of its low latency compared to other types of memory, it's ideal if computation can be done as much as possible in the shared memory, and the output can be thenceforth copied back from to the global memory if needed.

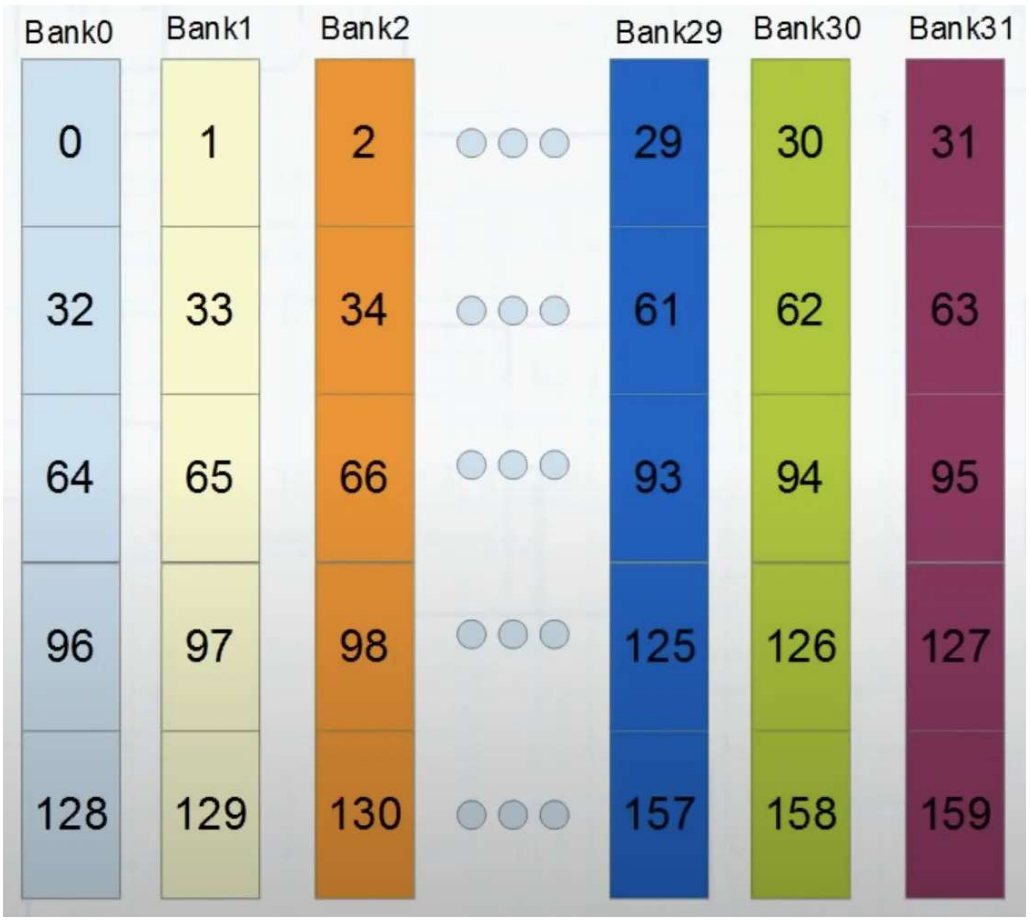


Figure 8: Fermi Inner-block threads' shared memory Layout. Source: What's a Creel

In Fermi architecture, shared memory for inner-block threads is divided into 32 **bank** units, which each can hold multiple 4-byte long data (word). If shared memory is divided into words, word i lies in bank i % 32. Normally, each thread would access any data element within these banks that corresponds to the thread's ID, which can be accessed using **threadIdx, blockIdx,** and **blockDim.** A more throughout analysis can be found in this lesson by NYU Center for Data Science and this article by Eranga Dulshan.

Because of the nature of data allocation in the shared memory, two concurrent threads in a warp can access different words in the same bank at the same time, causing a **bank conflict** that makes GPU serialize accesses the issued accesses to this bank. Since serialization in GPU is undesirable and clock-cycle costly, this access pattern should be avoided. An example of bank conflict can be demonstrated in this following figure:

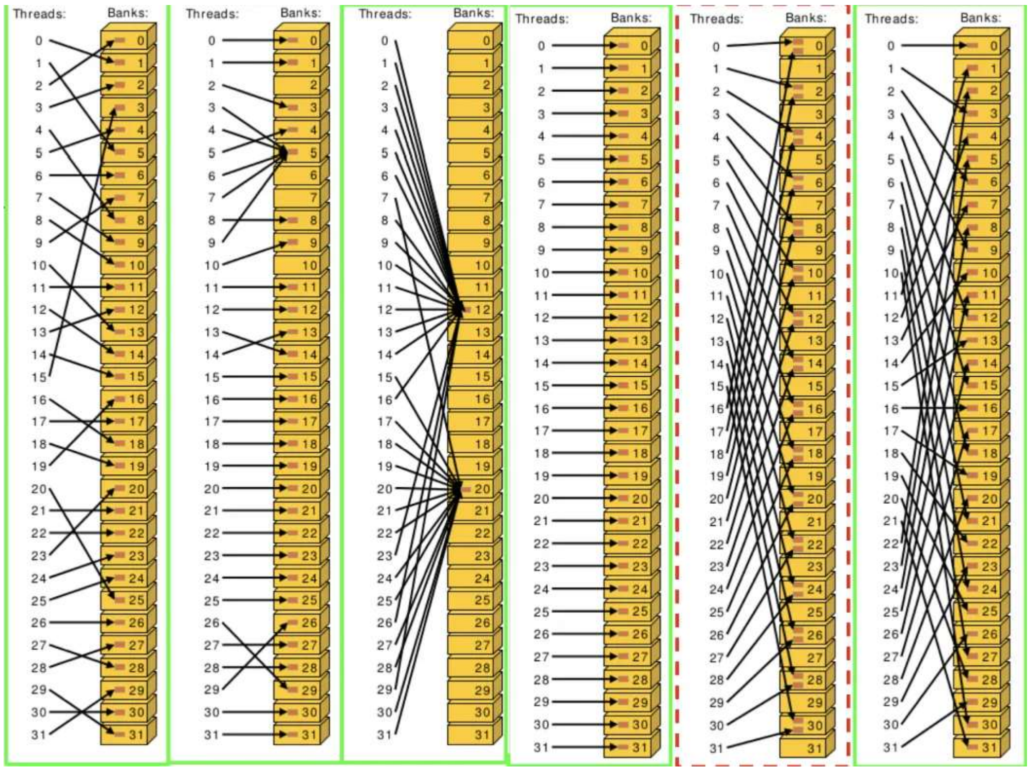


Figure 9: Bank conflicts examples. Source: [3]

From figure 9, it's easy to point out there's no bank conflict for access pattern 1, 2, 3, 4, and 6. The reason for no bank conflict may be trivial with pattern 1, 4, and 6 since there are no two threads that issue access instructions to the same bank. For pattern 2 and 3, when there're multiple threads that want to access the same bank **but for the same word location**, the compiler can sort this out by issuing a **multicast** (for a subset of threads)**/broadcast** (for all threads) packet, which delivers the data at the word location to all requesting threads. At pattern 5, multiple threads are requesting data **from different word locations** within the same bank, causing traffic congestion and bank conflict.

L1/L2 cache

Fermi maintains three levels of cache so that future requests for previous computed or pulled data can be served faster for accesses.

Each SM in Fermi architecture has its own L1 cache. L1 cache maintains data for local & global memory. From figure 5, we can see that it shares the same hardware as the shared memory. As stated above with the SM description, Nvidia used to allow a configurable size (16, 32, 48KB) (but dropped that in recent generations). L2 cache is also used to cached global & local memory accesses. Its total size is roughly 1MB, shared by all the SMs.

Constant memory

Constant memory is global memory with a special cache. It is used for the constants that cannot be compiled into the program. These constants must be set from the host before running the kernel function. It maintains roughly 64KB in memory for the user, 64KB for the compiler. One usage of constant memory is that it contains arguments that are passed to the kernel function.

Texture memory

Texture memory is read-only device memory and can be accessed using the device functions described in Texture Functions. Reading a texture using one of these functions is called a *texture fetch*. Texture memory traffic is routed through the texture cache (which is independent of the L1 data cache) and the L2 cache.

Texture memory is a complicated design and only marginally useful for general-purpose computation. It exploits 2D/3D spatial locality to read input data through texture cache and CUDA array, which the most common use case (data goes into special texture cache). The GPU's hardware support for texturing provides features beyond typical memory systems, such as customizable behavior when reading out-of-bounds, and interpolation filter when reading from coordinates between array elements, integers conversion to "unitized" floating-point numbers, and interaction with OpenGL and general computer graphics.

Take away

CPUs and GPUs have different architectures and are built for different purposes. For CPU, which is created with a smaller number of powerful cores, it's all about low latency task processing, completing complex tasks quickly. On the other hand, GPU, which comprises thousands of concurrent threads hundreds/thousands of less-powerful cores, focuses on improving work throughput by executing many simple tasks at once.

Fermi architecture was designed in a way that optimizes GPU data access patterns and fine-grained parallelism. Important notations include host, device, kernel, thread block, grid, streaming processor, core, SIMT, GPU memory model.

What's next?

Next up in the series, we will dissect one of the latest GPU microarchitecture, Volta, NVIDIA's first chip to feature Tensor Cores, specially designed cores that have superior deep learning performance over regular previous created CUDA cores. In-depth, we will again focus on architectural design and performance advancements Nvidia has implemented.

Stay tuned for the next article!

References

[1] John Nickolls, William J. Dally: The GPU Computing Era. 2010.

[2] Michael Doggett: Texture Caches. 2012

[3] Alan H. Barr: California Institute of Technology, CS179: GPU Programming, Lecture 4. 2020

[4] Nvidia: Local Memory & Register Spilling. 2011

[5] Nvidia Nsight Visual Studio Edition: Memory Statistics — Texture. 2015.

[6] Brian Caulfield, Nvidia: What's the Difference Between a CPU and a GPU? 2009.

Gpu Nvidia Microarchitecture Computer Architecture Parallel Computing

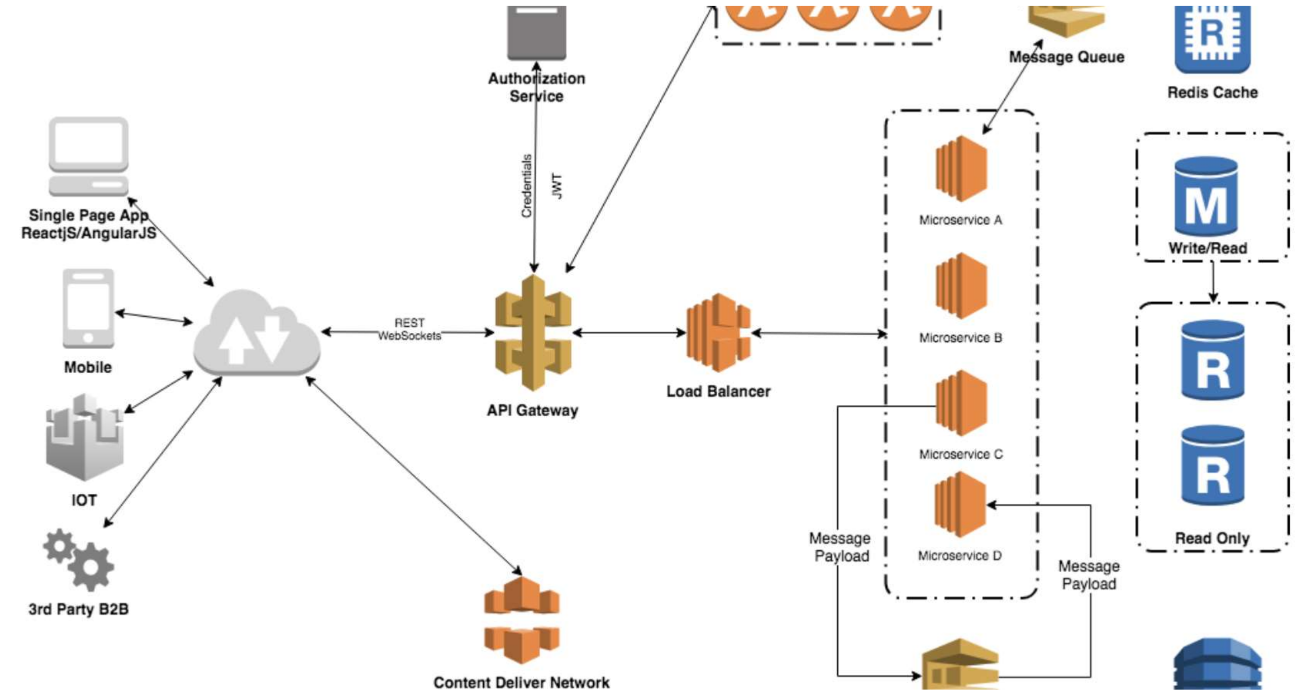
Follow

Written by Dung Le

397 Followers · Editor for Distributed Knowledge

Previously at @Citadel, @Facebook, @Tesla, @Amazon

More from Dung Le and Distributed Knowledge



Dung Le in Distributed Knowledge

Scalable Web Architectures Concepts & Design

For every web application, one of the fundamental factors that decides its success is its ability to seamlessly and efficiently...

✦ · 17 min read · May 18, 2020

1.6K 5