A Survey of Nvidia Architectures and their Support for CUDA

Tyler Paul

CS 590 - Advanced Computer Architecture

1 Abstract

Graphics processing units (GPUs) have become increasingly parallel machines, for this reason GPUs have begun to be used for computing alongside the CPU. Nvidia developed a programming model called CUDA which their Tesla, Fermi, and Kepler GPU architectures support. There are many differences among these architectures but there are also many similarities. These architectures all contain many cores called streaming processors (SPs). A GPU exhibiting a Tesla architecture often contains only 128 SPs while the Fermi and Kepler architecture GPUs may contain 512 and 1536 SMs respectively. The Nvidia architectures also contain a number of components called warp schedulers. Warp schedulers can be used to schedule threads in multithreaded CUDA programs to the SPs in GPUs exhibiting Nvidia architectures where they can then be executed. Thus a GPU can execute more threads in parallel upon an increase in the number of SPs in the GPU. For this reason programs written in CUDA exhibit a high degree of scalability and CUDA proves to be a useful programming platform for the future. The number of warp schedulers in Nvidia GPUs has increased for each generation of their architectures. This allows for more threads to be executed in parallel and therefore allows for multithreaded programs (in particular CUDA programs) to run faster. CUDA programs offer a nice programming abstraction that allows for programmers to not worry about the actual Nvidia GPU architecture. A CUDA program consists of entities called kernels which consist of grids of thread blocks. The Nvidia hardware allows for threads within a block to share memory and for each thread block to share a different global memory. Caching techniques are also implemented in the hardware to make memory sharing more efficient. Overall, the Nvidia architectures allow for CUDA programs to run efficiently in parallel.

2 Introduction

Graphics processing units (GPUs) have become increasingly parallel machines, for this reason GPUs have begun to be used for computing alongside the CPU. This development is called General-purpose computing on graphics processing units or GPGPU. Nvidia's GPU architectures, which support the CUDA programming model, made it possible for developers to easily take advantage of the parallel nature of GPUs in order to write multithreaded code for some applications. In this paper we discuss three architectures developed by Nvidia and briefly describe how they support CUDA. Throughout this paper, we will often use the GeForce 8800 GTX and Tesla architecture synonymously (the same with the GeForce 480 GTX and Fermi architecture along with the GeForce 680 GTX and Kepler architecture). However, these GPUs represent their architectures well and there are not that many large differences between GPUs in a given architecture.

3 Common Components Across Nvidia Architectures

Nvidia's Tesla architecture, first released in 2006, was the first architecture to support CUDA [3]. Since then the Fermi, Kepler, and Maxwell architecture have been developed. We will not discuss the Maxwell architecture for reasons of brevity. Across all of these architectures, there are many common components. We discuss the function of several of them before discussing the Tesla, Fermi, and Kepler architectures.

- **Streaming Processors (SPs)**: The basic building block of the Nvidia architectures is the streaming processor. These are also referred to as a CUDA core. Each SP consists of two basic units: An ALU for integer operations and a FPU for floating point operations [5].
- **Special Function Units (SFUs)**: A SFU computes instructions such as that would otherwise be time consuming on non-specialized hardware such as sine, cosine, and square roots [3, 5].
- **Streaming Multiprocessors (SMs)**: SMs do most of the work in the GPU. Most GPUs contain between two and a couple dozen SMs and each consists of: several SPs, a few SFUs, load/store units, shared memory, registers, and what is called a warp scheduler [3].
- **Shared Memory**: Since an SM consists of several SPs it is advantageous to have memory such that the SPs can share thus allowing the SPs to essentially communicate. For this reason, each SM has a portion of memory dedicated for this purpose [3].
- **Registers**: Each SM contains a large register file which individual SPs within the SM use during computation.
- **Load/Store Units**: These units load or store data in global memory (DRAM on the GPU) [5].
- **Warp Scheduler**: Each SM contains one to a few components called warp schedulers. Warp schedulers schedule threads in units of 32 threads (called a warp). Many warps are executed concurrently by a warp scheduler. One instruction from a warp is chosen at a time and is executed [3, 5]. Than a new warp is chosen and an instruction from that warp is executed, and so on. Note that each thread in a warp executes the same instruction but with different data (stored in the registers in the SM). Thus SIMD parallelism is exploited. We will discuss the architecture dependent features in a later section.
- **Host Interface**: The host interface allows the GPU and CPU to communicate. When the CPU sends a command, the GPU decodes the command and passes the work to appropriate GPU hardware. The physical connection between the GPU and CPU is done through PCI-Express [3, 5].
- **GigaThread Thread Scheduler**: As stated above, warp schedulers schedule threads in warps into the GPU hardware. But additionally, a seperate global work distribution engine schedules blocks of threads to the warp schedulers. These two components make up the GigaThread thread scheduler which is how the Nvidia architectures schedule their threads [5].

4 Tesla Architecture

The GeForce 8800 GTX released by Nvidia in 2006 was the first GPU that exhibited the Tesla architecture. See the figure below for a block diagram of the Tesla architecture. It has 16 SMs and 8 SPs per SM for a total of 128 SPs (or 128 CUDA cores). Each SM containes a single warp scheduler and 2 SFUs [3, 5]. Due to the fact that each SM contains less SPs than the size of a warp (32 threads) then there is often be a latency of several clock cycles for the warp scheduler to dispatch an entire warp instruction. For

example, suppose the warp scheduler chose to schedule a particular instruction in a warp which was an integer instruction. Then 32 SPs would be needed since each thread requires an ALU in a SP. Thus the thread scheduler would need to dispatch 8 threads out of the warp per clock cycle for duration of 4 clock cycles.

Additionally, the Tesla architecture only supported 24 bit integer multiplication [3, 5]. Consequently, if a developer wanted to obtain 32 bits from an integer multiplication several instructions would need to be used. Obviously, this would negatively impact performance.

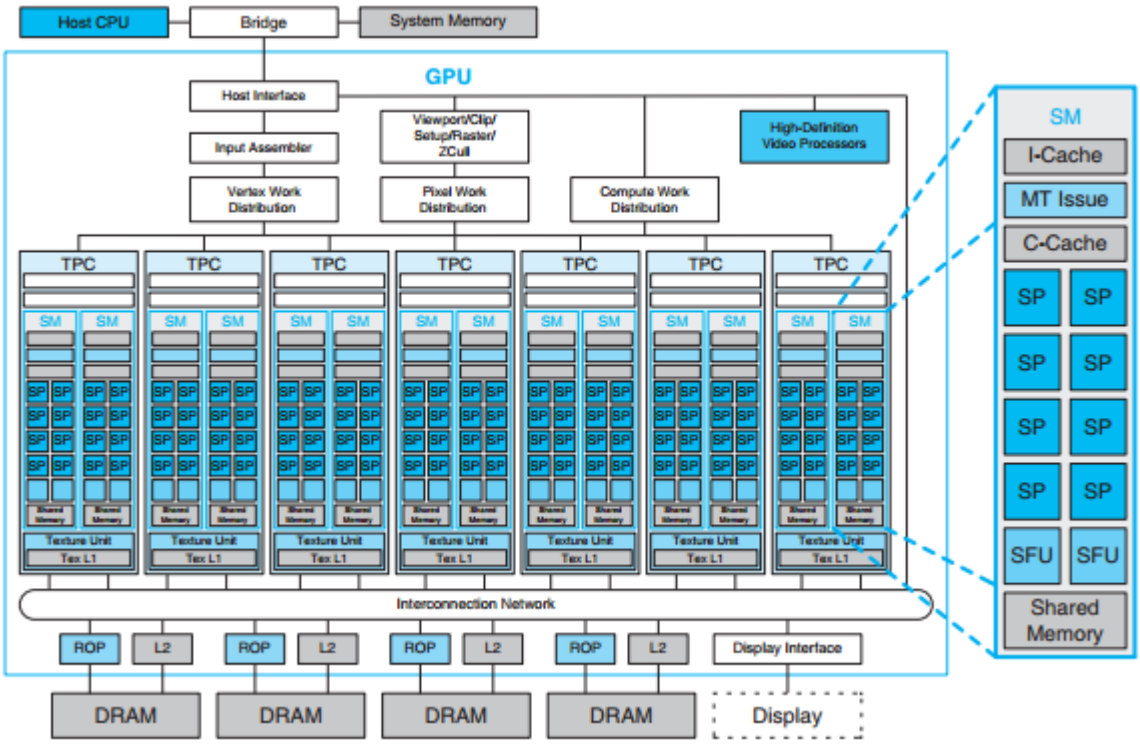

Figure 1: Tesla Architecture. Image taken from [2]

5 Fermi Architecture

In 2010 Nvidia released the GeForce 480 GTX, the first GPU exhibiting a Fermi architecture [3]. This architecture offered several improvements over its predecessor. See the figure below for a block diagram of the Fermi architecture.

First of all, it contains many more CUDA cores. In particular it contains 16 SMs where each SM contained 32 SPs for a total of 512 SPs.

Additionally, there is much improvement in terms of the memory hierarchy. A 768KB L2 cache was put between two rows of the SMs. This allows for memory to be shared between SMs. Additionally, each SM allows for memory to be shared by the SPs within it. Nvidia put 64 KB of memory within each SM and developed a mechanism to allow developers to choose how much of this memory they wanted to designate as shared memory and how much as a L1 cache. Thus developers could choose whatever scheme would be beneficial to their particular application.

The Fermi architecture also allows for a 32 bit multiplier to replace the 24 bit multiplier of the Tesla architecture.

There was also an upgrade in the number of warp schedulers per SM. For Fermi, there are 2 warp schedulers and two dispatch units per SM which allows for 2 instructions from 2 warps to be executed at

the same time. Each instruction from a warp can be issued to 16 SPs, 16 load/store units, or 4 SFUs. For example, suppose two integer instructions from two different warps are scheduled. Then at the start of one clock cycle each dispatch unit can dispatch 16 threads to the ALU in each SP. Then during the next clock cycle, the remaining 16 threads of each warp can be dispatched to ALUs. Thus two instructions can be dispatched over two clock cycles [3, 5].

Starting with Fermi, Nvidia GPUs began to support error correcting codes (EECs). This allows for single bit errors to be detected in hardware. This is especially useful in large clusters since bit errors become likely to occur due to the large amount of hardware [5].

Lastly, the Fermi architecture allows for 16 kernels to be executed at the same time. The Tesla architecture could only execute a single kernel at a time, so kernels were processed serially. We will discuss what kernels are in a later section about CUDA.

Lastly, the overall goal in building the Kepler architecure was to make GPUs power efficient and many more mechanism were included in the architecture to support this goal.

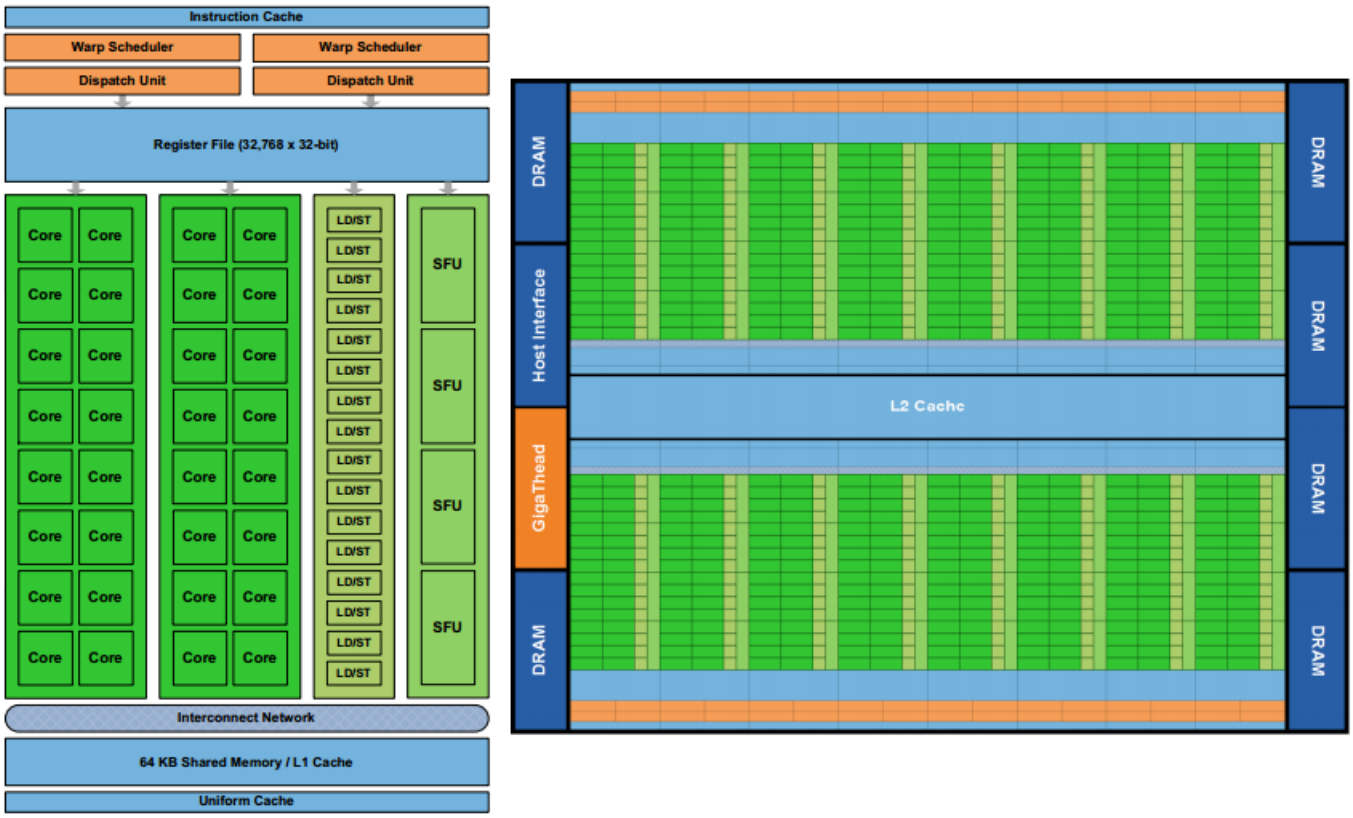

Figure 2: A Fermi SM is shown on the left and the overall Fermi architecture is shown on the right. Image taken from [5]

6 Kepler Architecture

In 2012 Nvidia released the GeForce 680 GTX, the first GPU exhibiting a Kepler architecture [3]. See the figure below for a block diagram of the Kepler architecture.

In the Kepler architecture, SMs were extended to include much more hardware. These extended SMs are called SMXs by Nvidia. A Kepler GPU contains 8 SMXs and each SMX contains 192 SPs and 32 SFUs. Thus a Kepler GPU can contain 1536 CUDA cores. Additionally, each SMX contains 4 warp schedulers and each warp scheduler can dispatch two instructions at once (via two dispatch units per warp scheduler)

[4].

Additionally, the memory clock was increased to 6008MHz compared to the smaller memory clock rates of the Tesla and Fermi architecture. The core clock runs at 1006MHz buts the Kepler architecture supports an interesting technique to increase this rate during certain conditions. This technology is refered to as GPU boost by Nvidia. When the GPU isn't drawing very much power, then GPU can increase it's clock rate. This allows for a clock rate of 1058MHz on the average (referred to as the boost clock rate). However, at any given time the clock rate can be even higher than that if the power being drawn in still low. A clock rate of 1.1GHz can sometimes be attained [4].

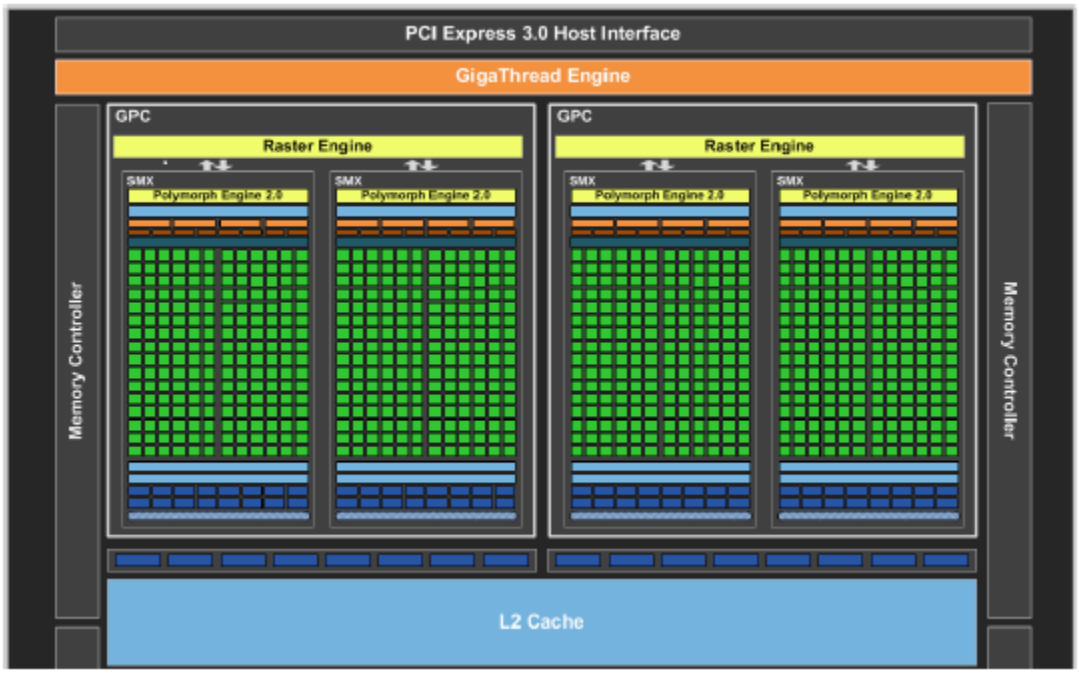

Figure 3: Top portion of the Kepler architecture. Four more SMXs are below the L2 cache. Image taken from [4]

7 Architecture Summary

A summary of various aspects of the three architectures discussed above are shown in the table below.

Architecture	Tesla	Fermi	Kepler
Number of SMs (or SMXs)	16	16	8
Number of SPs (CUDA cores)	8	32	192
Total CUDA cores	128	512	1536
Number of SFUs per SM	2	4	32
Number of Warp Schedulers per SM	1	2	4
Graphics Core Clock	575MHz	607MHz	1006MHz
Unified L2 Cache Size	None	768KB	512KB
Concurrent Kernels	1	up to 16	up to 16 (or 32)
ECC Support	no	yes	yes
GPU Boost Support	no	no	yes

Figure 4: Tesla, Fermi, and Kepler Architectures Compared

8 CUDA Paradigm

CUDA is a programming model developed by Nvidia. The three architectures described above all were designed to support CUDA. The main advantage of CUDA is that programs written in CUDA scale extremely well on parallel hardware.

In order for a programmer to create a multithreaded program written in CUDA they must organize the threads in their programs accordingly. First of all, in order to parallize code in a CUDA program, the programmer should write what are called kernels. In software, a kernel can be written in a similar way as a regular function call would be written, but when invoked it will run in parallel. Each kernel needs to be organized into grids of thread blocks [1]. A thread block is can be thought of as simply a collection of threads. All the threads in a thread block will execute concurrently. Thread blocks in a grid of thread blocks should be independent while threads in a thread block can share memory. When a kernel is invoked all the threads in the grids of thread blocks will run an instance of the kernel. Each thread is given a thread ID and block ID, so if these numbers are used inside of the kernel then each thread will operate on a different data [1, 5]. However the programmer only needs to write a single kernel, thus kernels are essentially an example of SIMD.

As an example, we can write a kernel which can be used to compute the vector sum of two vectors where each vector has 512 entries.

```
__global__ void vectorSum(int *x, int *y, int *z) {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      z[i] = x[i] + y[i];
}
//invoke kernel using
vectorSum<<<2, 256>>>(x, y, z);
```

In this example, two thread blocks each containing 256 threads are used to launch the kernel. After all of these threads complete their executed, z will contain the vector sum of x and y. The key point is that each thread has a different block ID and thread ID, so each block will use a different i inside the kernel. Thus each thread will operate on a different entry (out of the 512 entries) in the vector sum.

Now that we understand the basics of CUDA we can discuss how Nvidia GPUs support CUDA. First of all, when a kernel is invoked the grid of thread blocks in sent to the GPU (via PCI Express). Next, the scheduling system assigns each block in the grid to a particular SM (thus allowing the different blocks to run in parallel). Within each SM, the threads in the block are scheduled in warps (groups of 32) by the warp scheduler to the various SPs [1, 5]. Additionally, each thread is given a portion of the register file in their SM to use during computations to store local variables. Also, threads in a thread block can use the shared memory in their SM to communicate. Lastly, threads from different thread blocks can communicate using the global DRAM [1]. Thus the Nvidia architectures allow many threads in a kernel to run concurrently and to share memory efficiently.

9 Conclusion

We have now seen how Nvidia's Tesla, Fermi, and Kepler architecture was constructed and how these architectures support CUDA. Kernels in CUDA programs offer an abstraction which allows for the programmer to not worry about the hardware the code is being run on since the GPU will handle scheduling threads to SPs. Therefore programs written in CUDA will scale extremely well upon increasing the number of cores in hardware. For this reason, CUDA proves to be a useful programming platform for the future where the amount of cores in GPUs is expected to keep rising in accordance with Moore's Law.

References

[1] J. Nickolls, *Scalable Parallel Programming with CUDA*, http://queue.acm.org/detail.cfm?id=1365500

[2] D. Patterson and J. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, Waltham, 2014.

[3] N. Wilt, *The Cuda Handbook*, Addison-Wesley, Upper Saddle River, 2013.

[4] *NVIDIA GeForce GTX 680*, http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[5] *NVIDIA's Next Generation CUDA Compute Architecure: Fermi*, http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf