2016

# Adaptation of a GPU simulator for modern architectures

Piriya Kristofer Hall
*Iowa State University*

# Adaptation of a GPU simulator for modern architectures

by

Piriya Kristofer Hall

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Iowa State University

Ames, Iowa

2016

## DEDICATION

I would like to dedicate this thesis to my parents, Mark E. Hall and Suwimon Hall, who have supported me and encouraged me in all my endeavors, especially in academia. I would also like to thank my friends for being there for me, to help keep my sanity in check, and all the members of the Thai Student Association for acting like a second family away from home.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. Firstly, Dr. Zambreno for his guidance, and patience during this project and the writing of this thesis. I would also like to thank my committee members, Dr. Jones and Dr. Daniels for their patience and understanding. Additionally, I would like to thank Xian Zhu for his help with this project especially with sanity checking of results and model decisions.

## ABSTRACT

GPUs have evolved quite radically during the last ten years, providing improvements in the areas of performance, power consumption, memory, and programmability, increasing interest in them. This increase in interest, especially in academic research into GPU architecture, has led to the creation of the widely used GPGPU-Sim, a GPU simulator for general purpose computation workloads. The simulation models currently available for simulation are based on older architectures, and as new GPU architectures have been introduced, GPGPU-Sim has not been updated to model them.

This project attempts to model a more modern GPU, the Maxwell based GeForce GTX Titan X. This is accomplished by modifying the existing configuration files for one of the older simulation models. The changes made to the configuration files include changing the GPU's organization, updating the clock domains, and increasing cache and memory sizes. To test the accuracy of the model, eleven GPGPU programs, some having multiple kernels, were chosen to be executed by the model and by the physical hardware, and compared using IPC as the metric.

While for some of the kernels the model performed within 16% of the GeForce GTX Titan X, there were an equal number of kernels for which the model performed either much faster or much slower than the hardware. It is suspected that the cases for which the model performed much faster were ones in which either the hardware executed single precision instructions as double precision instructions, or the hardware ran an entirely different machine code for the same kernel than the model. The cases for which the model performed much slower are suspected to be due to the fact that the Maxwell memory subsystem cannot currently be accurately modeled in GPGPU-Sim.

# CHAPTER 1. INTRODUCTION

Graphics processing units, or GPUs, designed originally to be used purely as accelerators for 3D graphics [14], have evolved to become flexible and powerful coprocessors. They are now used in a variety of applications ranging from real-time 3D graphics rendering to high performance computing. [30] This evolution is fueled by the increase in performance of the GPU compared to the CPU, as shown in Figure 1.1 on the next page, and by the increased ease of programming the GPU for more general-purpose applications. [23, 30] This, in turn, has caused an increase in interest in GPUs over the last ten years.

Due to this increase in interest, a team of researchers at the University of British Columbia developed GPGPU-Sim, a cycle-level simulator focused on general-purpose GPU computing workloads. Initially, it was developed to test branch reconvergence schemes of threads in a batch on an SIMD pipeline. [12] It was later updated to support CUDA C, and it was used to test if different hardware configurations would improve the performance of certain programs that have been refactored to execute on the GPU. [5] It is now used mainly for GPU architectural research and exploration.

Currently, GPGPU-Sim has models of the following GPUs to simulate: the Quadro FX 5600, the Quadro FX 5800, the Tesla C2050, and the GeForce GTX 480. The two Quadro cards run on the Tesla architecture, which was originally introduced by NVIDIA in 2006, and updated in 2008. The Tesla C2050 and GeForce GTX 480 run on the Fermi architecture, which was released by NVIDIA in 2009. [14]

The Fermi architecture has been succeeded by two newer architectures, Kepler and Maxwell. Due to this, it was attempted to create a GPU model for GPGPU-Sim to simulate a more modern GPU: one based on the Maxwell architecture. The specific graphics card that was modeled is the GeForce GTX Titan X, which was launched on March 17th, 2015. This update

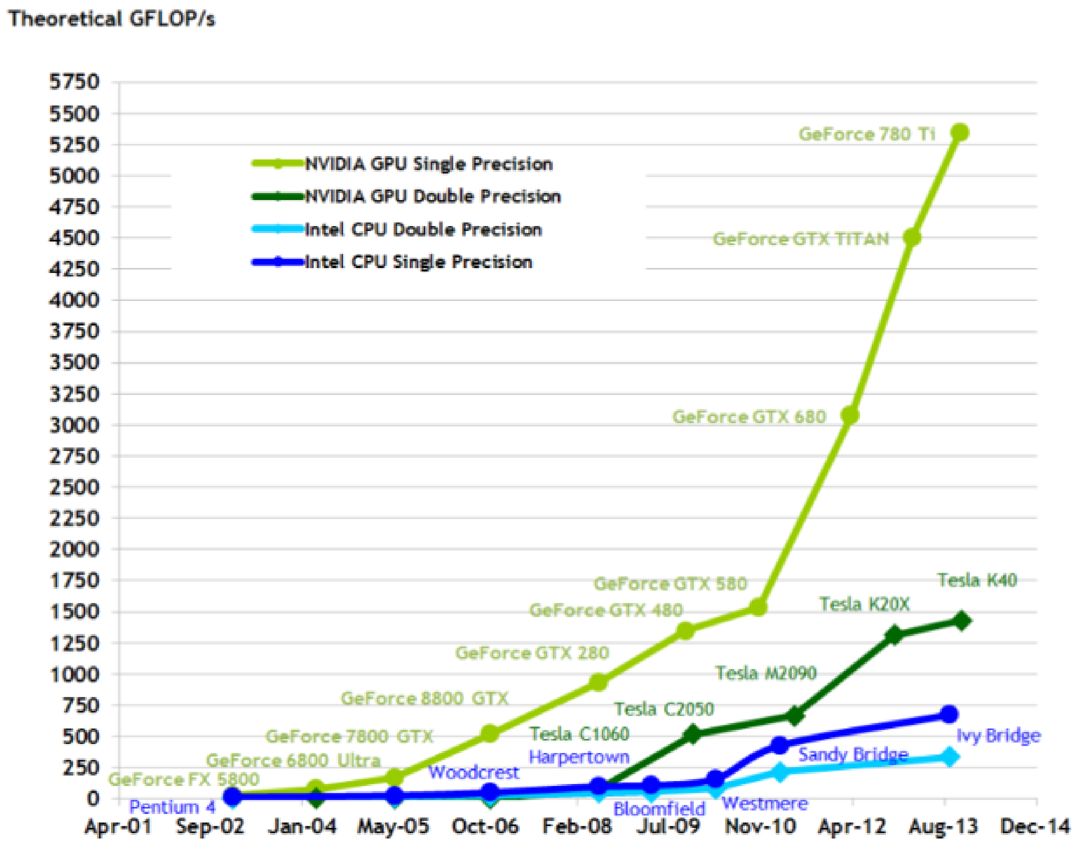

Figure 1.1 GPU FLOPS vs. CPU FLOPS [30]

was accomplished mostly through modifications made to the existing configuration files for the GeForce GTX 480 GPU, which are provided with the latest version of GPGPU-Sim.

The rest of the paper is laid out as follows: Chapter 2 briefly discusses relevant technologies, i.e., NVIDIA GPUs, especially their microarchitectures; Chapter 3 details the methodology used in the creation of the model to simulate the Maxwell architecture, along with its testing; Chapter 4 details the results of the testing of the model and an analysis of the results; scholarly works related to what is performed in this paper are discussed in Chapter 5; and Chapter 6 concludes the paper with a summary and an outline of possible future work.

# CHAPTER 2. EVOLUTION OF GPUS

As shown in the introduction, modern GPUs have a much higher performance, especially in compute tasks, than modern multi-core CPUs. Though not shown, GPUs also have a much higher bandwidth for memory than modern CPUs.

Such a disparity between the two can be attributed to the tasks they were designed to perform. The tasks which the CPU was desigend for are less computationally intensive, and more general purpose, such as, e.g., taking user input and executing a sequence of code based on the input, or opening a file, reading its contents, and displaying it for the user. These tasks often have branches and random memory access patterns. The tasks that the GPU was designed for are primarily acceleration of real-time 3D rendering. This requires a lot of parallel computation with very little branching, and has a more regular memory access pattern. Due to the differences in these tasks, the design of a CPU favors more control flow logic and larger caches to hide memory access latency with little to no need for high memory bandwidth; while the GPU favors more computation units with minimal cache and control flow logic, and sufficient memory bandwidth to feed the computation units. A graphical representation of this can be seen in Figure 2.1 on the next page.

This general architectural pattern has fueled GPU design from the early days of its development.

## 2.1 Early GPUs

Early graphics processors started off as multi-chip 3D rendering engines in the 1980s and consisted of fixed-function pipeline processors for vertex transformation and lighting, and pixel-fragment filling. These processors became integrated on the same chip by 1994. As the

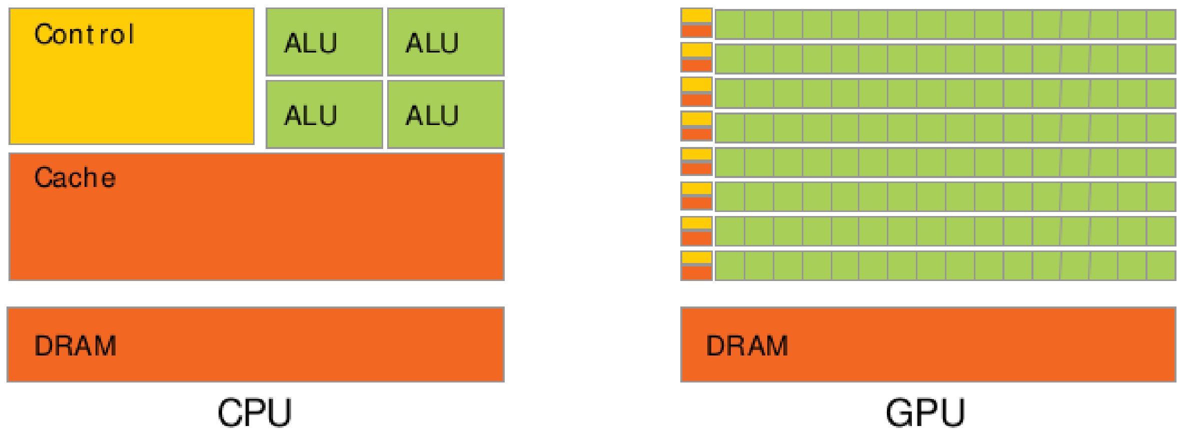

Figure 2.1 The General Architecture of a CPU vs a GPU [30]

complexity of vertex and pixel-fragment processing increased, the respective processors became programmable, starting with the vertex processor, with the release of the GeForce 3 in 2001 [21, 14].

While such an architecture was adequate, its flaws became more aparent as time progressed. One flaw is the problem of load balancing. In the typical case, GPUs handle more pixels than vertices; thus pixel-fragment processors should outnumber vertex processors. [21] However, finding the right fixed ratio of vertex processors to pixel processors to handle all cases effectively is not an easy task.

For example, if a scene with large triangles is to be rendered, the vertex processors will be mostly idle while the pixel processors are fully busy. An opposite scene containing a large number of small triangles, would see the vertex processor fully busy and the pixel processors mostly idle. [21]

Another flaw with this architecture is that if a new rendering stage is introduced, another processor would need to be added to handle the new stage. Then the load balancing problem presents itself again, where a fixed ratio needs to be found for three processors.

These flaws influenced the decision to design a unified processor architecture. NVIDIA addressed these flaws with the release of the Tesla architecture in November of 2006. [21]

## 2.2 Tesla Architecture

The main goal for the Tesla architecture was to allow execution of both vertex and pixel-fragment shader programs on the same unified processor architecture. Other goals were architecture scalability, performance, power, and area efficiency. [21]

This goal was motivated by the growing complexity in both the vertex and pixel-fragment processors, complexity of workload balancing between the vertex and pixel processors, the introduction of a new shader stage, and the potential for the reduction in human resource usage, i.e., there is no longer a need for separate teams to design the different stages in the graphics pipeline as only one team is needed to work on the processor. [21]

Tesla's overall architectural layout can be seen in Figure 2.2. It is based on a scalable processor array, comprised of eight texture/processor clusters or TPCs. Each TPC, as seen in Figure 2.3, contains two streaming multiprocessors, or SMs, with L1 cache used only for textures in the Texture Unit. [21] Not shown in the overall block diagram is the fact that the processors in the SMs are running at a clock rate that is twice as fast as the clock rate of the rest of the chip. This allows the chip to perform at the desired throughput without increasing the number of processors. [21, 29]

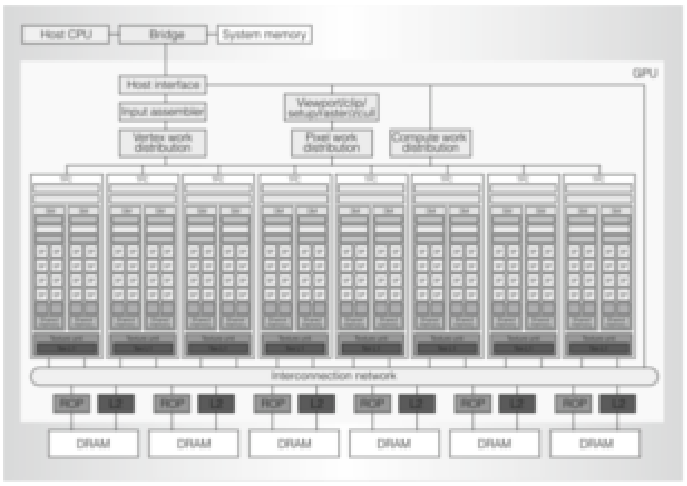

Figure 2.2 G80 Block Diagram [21]

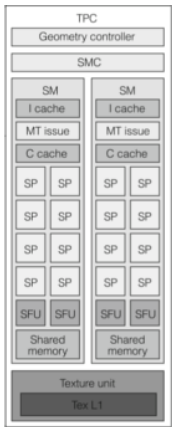

Figure 2.3 TPC Block Diagram [21]

Each individual SM, as shown in Figure 2.4, contains eight streaming-processors, or SPs, two special function units, or SFUs, instruction cache, constant read-only cache, shared memory, and a multithreaded instruction fetch and instruction issue unit. The SM can execute only one warp (defined in the next section) at any given time, but it manages a pool of 24 warps that it needs to execute. [21]

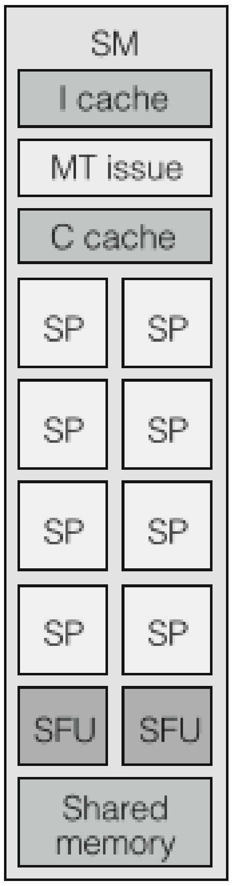

Figure 2.4 Tesla SM [21]

The Tesla architecture allowed for new parallel-computing capabilities. Before Tesla, computation programs had to be refactored to where the data needed for the computations, written as shader programs, was presented to the GPU as vertex or texture data. To simplify this process the CUDA programming model, and the extensions made to ANSI C for it, was developed alongside the development of the Tesla architecture. [21]

## 2.3 CUDA History and Terminology

The Compute Unified Device Architecture, or CUDA, is an extension made to ANSI C to deliver an easier and more effective way to write GPGPU programs. Central to a GPGPU program is a kernel. A kernel, which got its name from signal processing, is a function that describes how one thread would compute a result. Each kernel is executed N times, in parallel, by N different threads when invoked. [14, 21, 30] When a kernel is invoked in CUDA, the programmer specifies how many threads will execute the kernel.

A thread block, also called a Cooperative Thread Array, or CTA, is a group of multiple threads that all run on the same SM, with a limit to the number of threads in a thread block. The threads within the thread block can cooperate and share memory [21, 14]. Each thread block is divided into warps, a term originating from textile weaving. A warp is a group of 32 threads from the same thread block, and is the main unit of dispatch in each SM. [21, 14]

## 2.4 Fermi Architecture

The next microarchitecture developed by NVIDIA after Tesla was Fermi. Fermi was designed to provide better performance on all workloads compared to Tesla [24]. The full GF100, the high end part among graphics processors based on the Fermi architecture, is shown in Figure 2.5. The chip layout is again based on a scalable array comprised of four graphics processing clusters, or GPCs. Note that the TPC was replaced by the new GPC. This is because each SM now contains four dedicated texture units, eliminating the need for TPCs. [24] Each GPC contains four SMs as shown in Figure 2.6. The clock domains remain the same as in Tesla, i.e., the processors in the Fermi SMs are running at a clock rate twice as fast as the rest of the chip. [29] The GF100 also has 768KB of unified L2 data cache.

One of the biggest changes from Tesla to Fermi was in the SMs. As can be seen in Figure 2.7, each SM now contains 32 SPs, four SFUs, 16 Load/Store units, instruction cache, L1 cache/shared memory, texture cache, and two instruction fetch and dispatch units. The new SPs were designed for maximum performance and efficiency, and they fully implement the IEEE 754-2008 floating-point standard. This provides the ability to execute the fused multiply-add

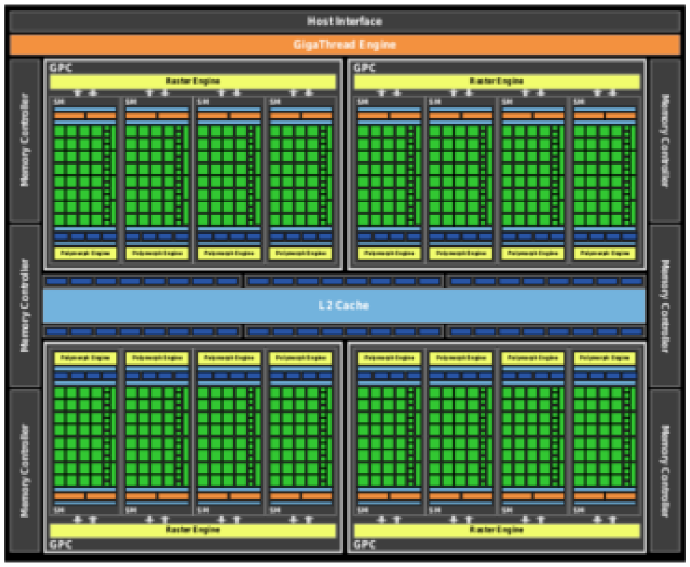

Figure 2.5 GF100 Block Diagram [24]

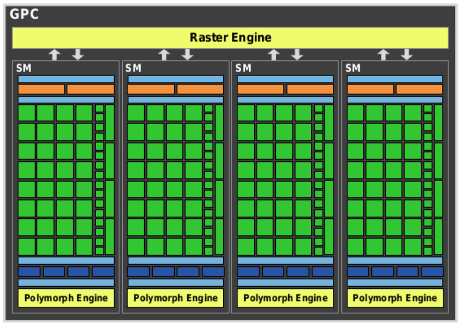

Figure 2.6 GPC Block Diagram [24]

instruction for both single and double precision computation. With two instruction fetch and dispatch units the SM can now have two warps, which do not have to be from the same thread block, executing simultaneously. However, double precision instructions cannot be dispatched at the same time as other instructions. [24] Note that L1 cache is now a part of the SM and not part of the texture unit. The L1 cache shares space with the shared memory and can be configured to be split as 16KB/32KB in favor of shared memory, or 32KB/16KB in favor of the L1 cache. [24]

## 2.5 Kepler Architecture

The Kepler architecture came after Fermi. The main focus for the Kepler architecture was to improve compute performance, especially for double precision computations, while reducing power consumption and heat output. [29] There were also features that were added to help improve GPU utilization such as Dynamic Parallelism, Hyper-Q, and the creation of the Grid Management Unit. Compute capabilities were also updated, allowing for more warps to run on an SM, among other improvements [29].

The GK110 block diagram is shown in Figure 2.8. While it is similar to the GF100, note that the GPC is no longer used. Another change to the architecture, in a departure from Tesla

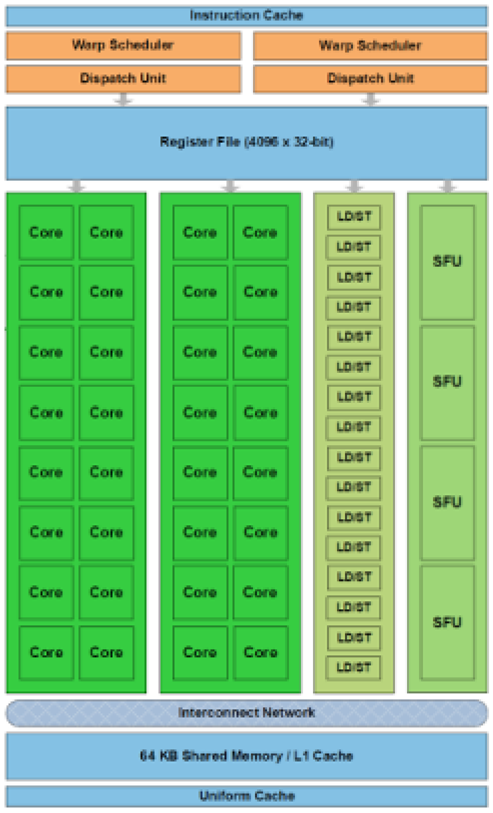

Figure 2.7 Fermi SM [14]

and Fermi, is that the processors in the SMs run at the same clock rate as the rest of the chip. This design decision was made to reduce power consumption as the logic of the clock for the processors required more power [29]. Finally, the unified L2 cache was increased in size from 768KB to 1536KB.

As can be seen in Figure 2.9, the Kepler SM doesn't depart too much from Fermi's SM, but there are still quite a few changes. The most noticeable is that the Kepler SM, referred to as an SMX, now has 192 SPs, 64 dedicated double precision units, 32 SFUs, and 32 LD/ST (load/store) units. SMX also has a larger front end containing four warp schedulers with two dispatch units per scheduler. This allows for two consecutive instructions, if possible, to be dispatched simultaneously. Another improvement to the scheduler and dispatch units is double precision instructions can now be paired with other instructions. The GK110 and GK210 have 64KB and 128KB of local SRAM respectively, to be split between L1 and shared memory just like Fermi; however, Kepler allows for more flexible options when splitting the SRAM, such as

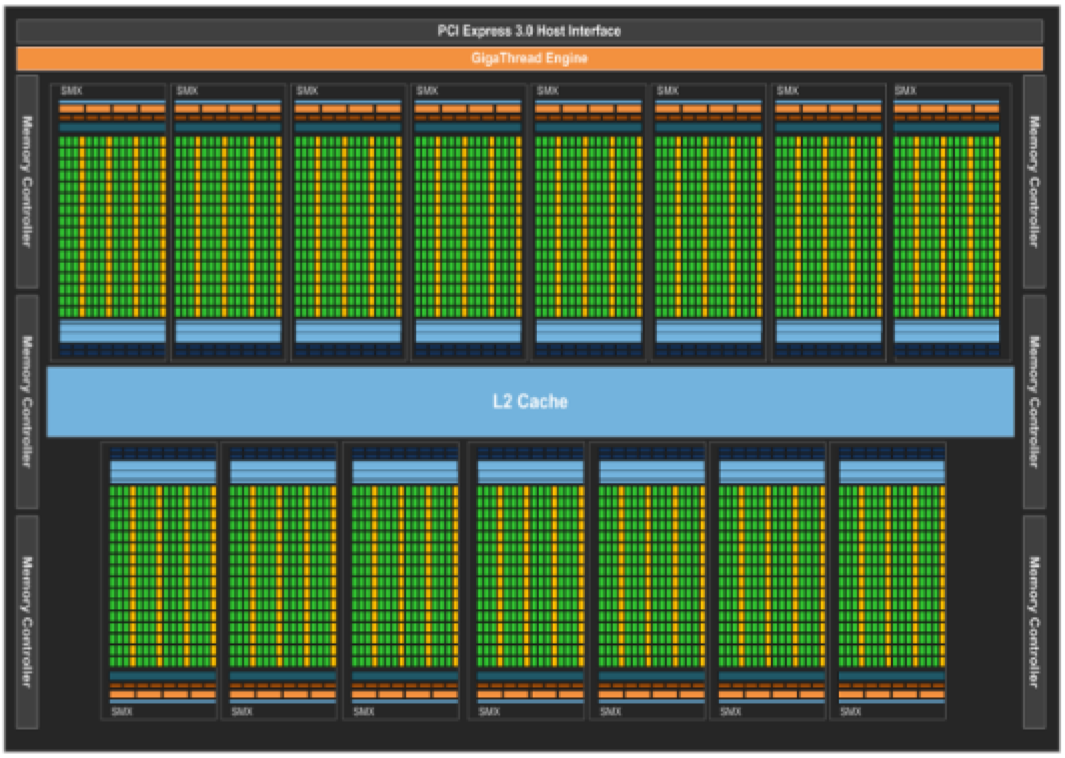

Figure 2.8 Kepler GK110/210 Block Diagram [29]

allowing for a 32KB/32KB split for GK110, and 32KB/96KB or 48KB/80KB splits between L1 and shared memory for GK210. A new addition to the Memory hierarchy for Kepler is 48KB of read-only data cache. Before Kepler this was only accessible through texture cache, but in Kepler it is now changed to be accessible from the SM directly [29].

## 2.6 Maxwell Architecture

NVIDIA's latest GPU architecture, Maxwell, is the successor to the Kepler architecture. Debuting February of 2014, Maxwell was designed with power efficiency in mind. The goal was to improve performance per watt and performance per area [27]. Maxwell still retains the same utilization technologies introduced in Kepler, such as Dynamic Parallelism while also supporting newer technologies introduced with the latest versions of the CUDA toolkit, such as unified memory [16].

The overall architecture of the GM200, the high end part among Maxwell-based GPUs, can be seen in Figure 2.10. A scalable array of six GPCs makes up the GPU with four Maxwell

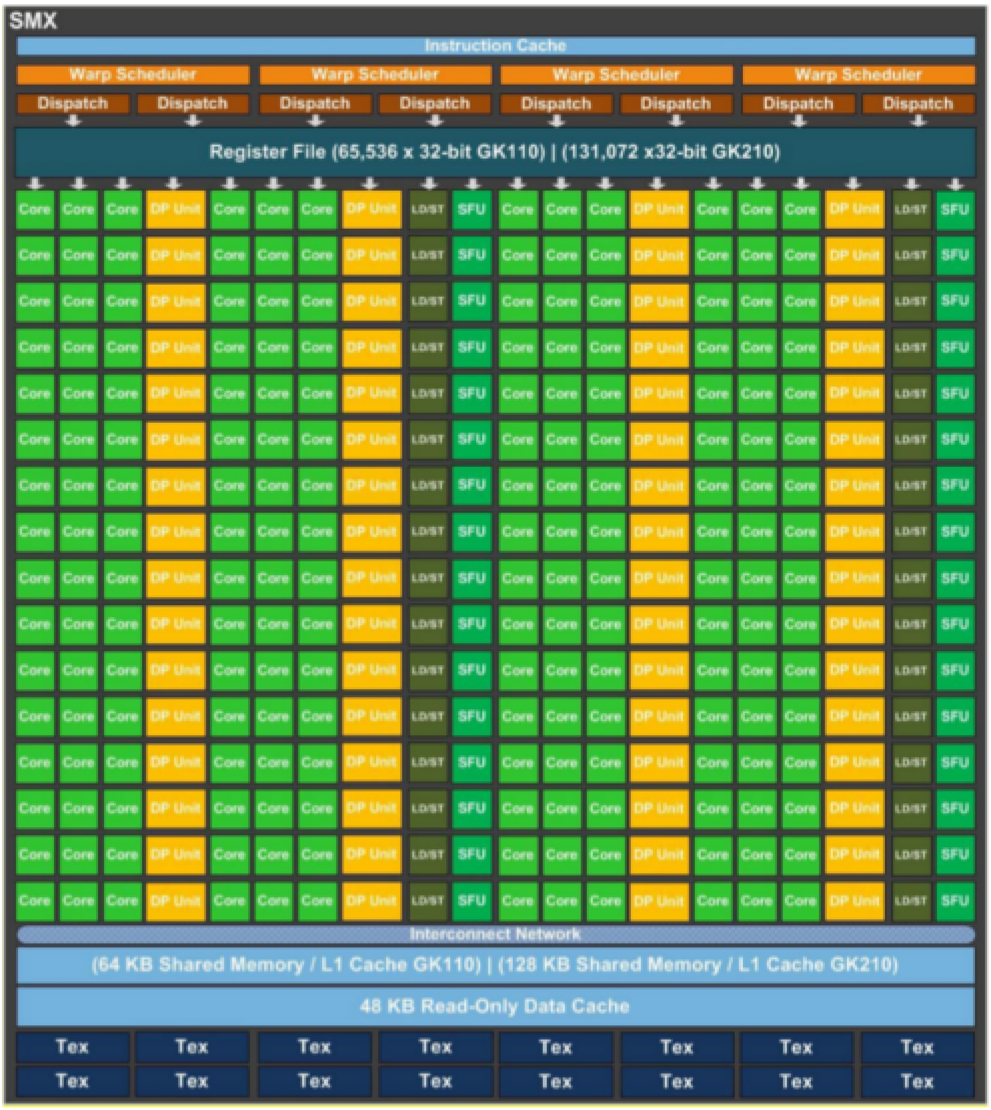
SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit GK110) | (131,072 x32-bit GK210)
Core
DP Unit
LD/ST
SFU
Interconnect Network
(64 KB Shared Memory / L1 Cache GK110) | (128 KB Shared Memory / L1 Cache GK210)
48 KB Read-Only Data Cache
Tex

Figure 2.9 Kepler GK110/210 SMX [29]

SMs, referred to as SMMs [27, 28], per GPC. Maxwell retains the same clocking configuration as Kepler, i.e., the processors run on the same clock as the rest of the GPU. A unified L2 cache 3072KB in size is shared across all six GPCs. [15, 2]

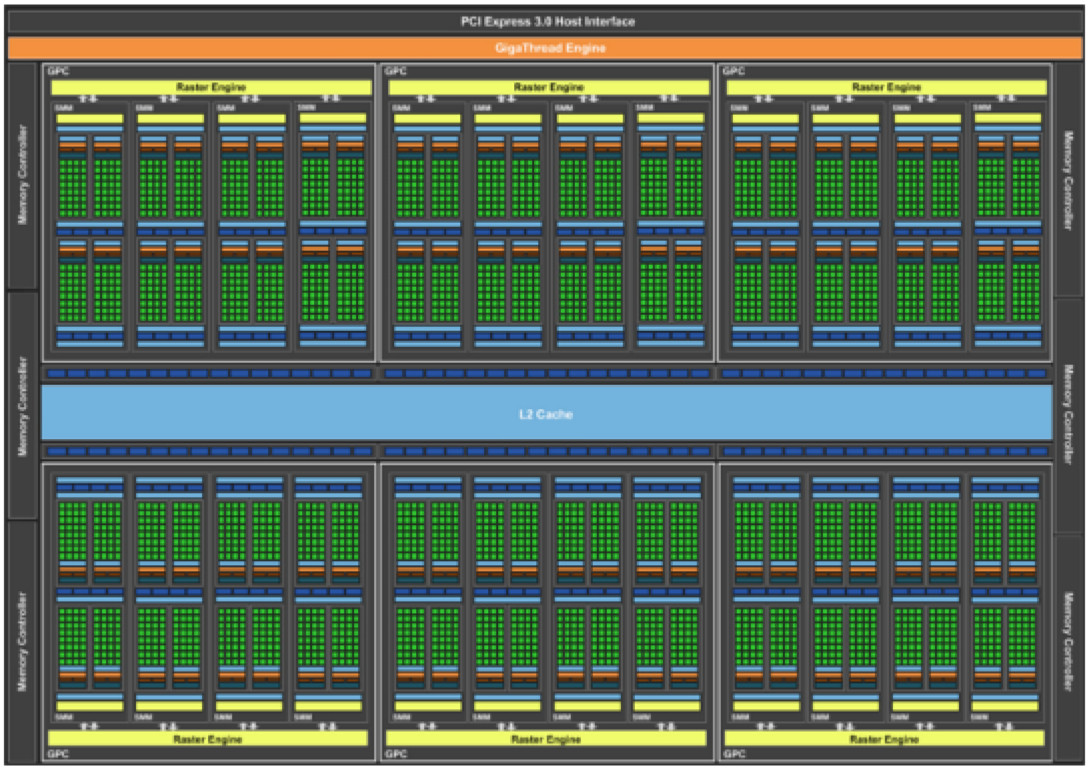

Figure 2.10 GM200 Block Diagram [37]

Each SMM, as shown in Figure 2.11, is divided into four processing blocks each containing its own instruction buffer, scheduler and dispatch units, 32 SPs, 8 LD/ST units, and 8 SFUs. This division simplifies the design and scheduling logic, reducing area, power, and computation latency. [28] This reduction in power, with minimal reduction in performance, allows for more SMMs to be placed in the GPU. [27, 28] All four blocks share 96KBs of dedicated shared memory, which is a departure from previous architectures. The L1 caching functionality has been moved to be shared with the texture cache. Two blocks share a single L1/texture cache. [27]

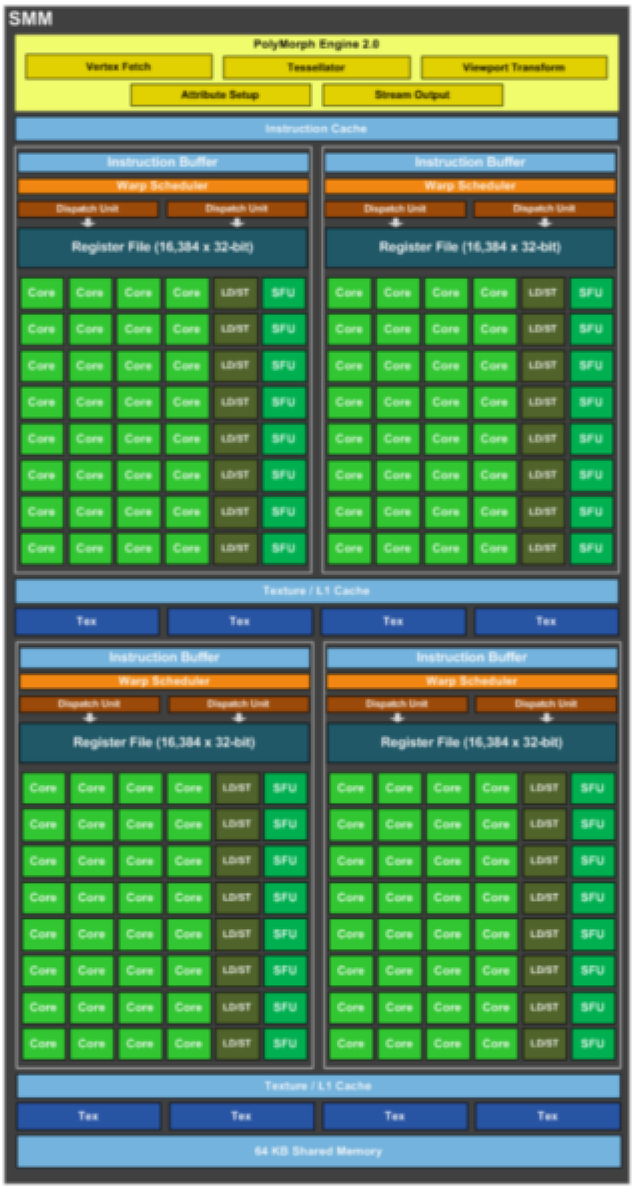

Figure 2.11 Maxwell SM(SMM) [28]

## 2.7 Pascal Architecture

Not yet available to the public, the Pascal architecture is the successor to the Maxwell architecture, and is designed to be the highest-performing parallel computing processor in the world. One of the major changes to Pascal from the previous architectures is the move to using high bandwidth memory, or HBM, as opposed to GDDR. [31]

Pascal's overall architecture improves upon what was learned in Maxwell providing it with better performance per watt. The full GP100, shown in Figure 2.12, is built using six GPCs each containing five TPCs, which in turn are each comprised of two SMs. The clocking configuration of Pascal is the same as that of Maxwell and Kepler. 4096KBs of unified L2 cache is shared between all six GPCs. [31]

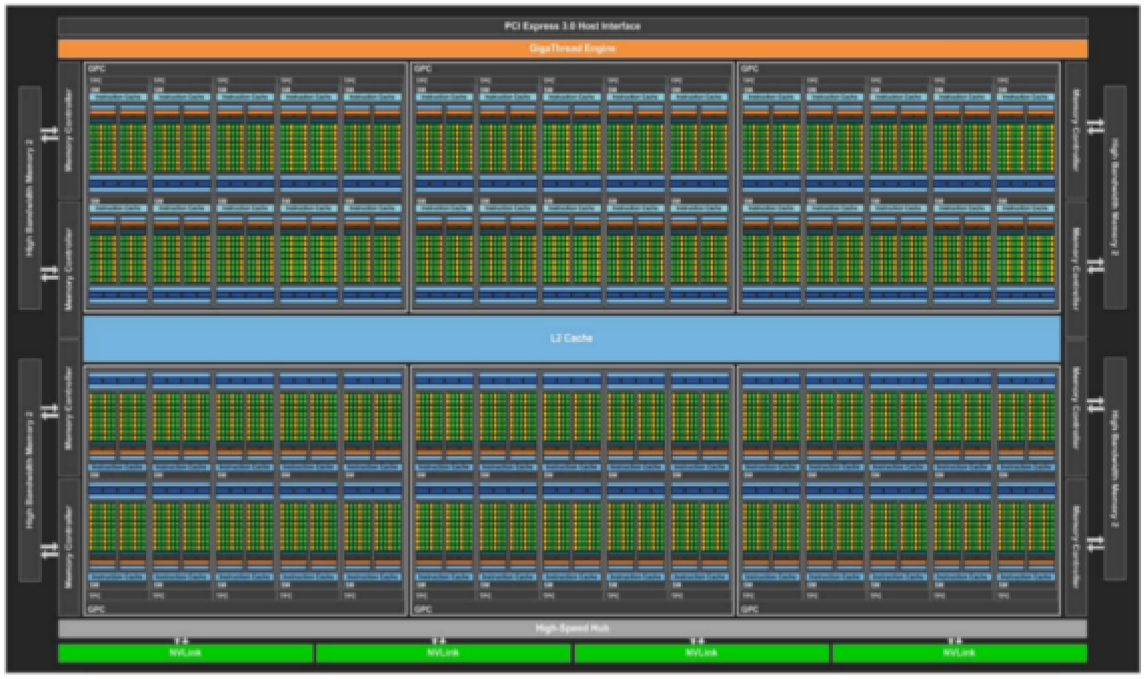

Figure 2.12 GP100 Block Diagram [31]

An SM is divided into two processing blocks as shown in Figure 2.13. Each processing block contains 32 SPs which can process both 16 and 32-bit instructions and data, its own instruction buffer, a warp scheduler with two dispatch units, 16 dedicated double precision units, 8 LD/ST units, and 8 SFUs. The SM maintains the same number of registers as the SMM, making the processing blocks in the Pascal SM have 32,768 registers. Each SM also has 64KB of shared memory available to the processing blocks. [31]

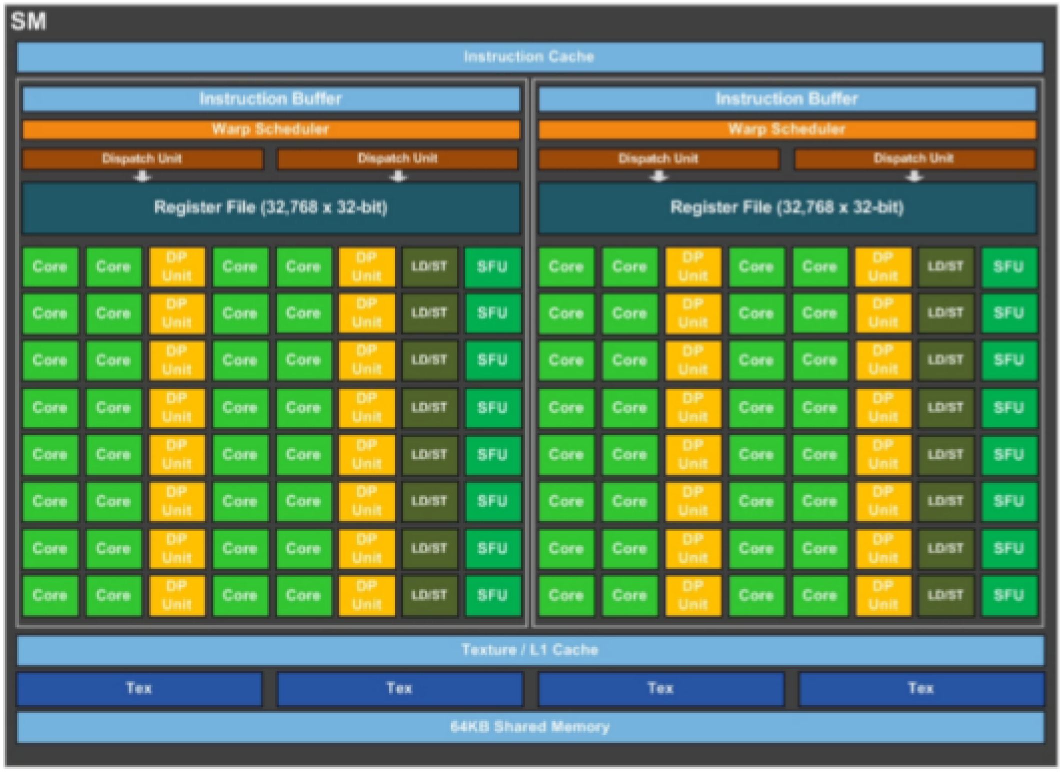

Figure 2.13 GP100 SM [31]

# CHAPTER 3. METHODS AND PROCEDURES

## 3.1 The GPGPU-Sim Simulator

The simulation platform used for this project is GPGPU-Sim, chosen because of its configurability, and familiarity within the GPU architecture research community. At the top level GPGPU-Sim is organized in a manner similar to the Tesla architecture. The model is composed of an array of clusters. These clusters are connected to memory partitions interfacing graphics GDDR DRAM through an on-chip connection network. Each of these pieces—the cores, the connection network, memory partitions, and DRAM—all have their own clock domains. These domains are independent and can have any arbitrary value. A visual representation of the overall architecture can be seen in Figure 3.1. [1]

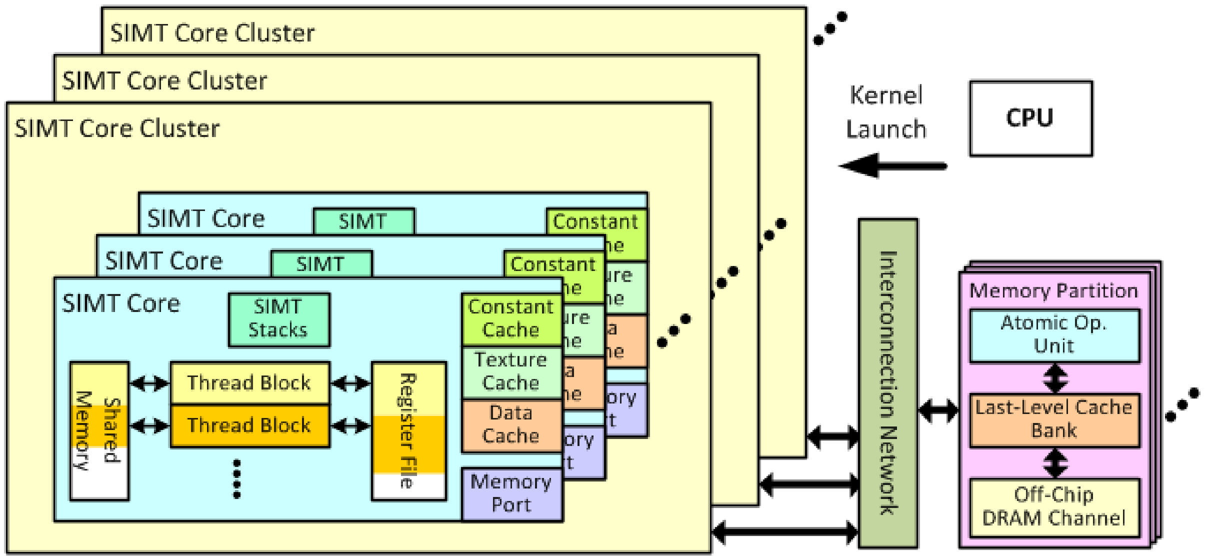

Figure 3.1 GPGPU-Sim Top Level Organization [1]

The layout of the clusters is shown in Figure 3.2. Each cluster is comprised of multiple

Single Instruction Multiple Thread, or SIMT, cores and a response FIFO. Each of these cores in the cluster shares a common port, the injection port, which interfaces to the on-chip connection network. The injection port handles all memory requests made by each core. The memory requests are then placed in the injection port buffer which is shared across all cores. The response FIFO holds packets returned in response to memory requests, ejected from the connection network, bound for either a core's instruction cache or its memory pipeline. These packets will be sent to their intended core in FIFO fashion. If a core cannot accept the packet at the head of the FIFO the FIFO is stalled. [1]

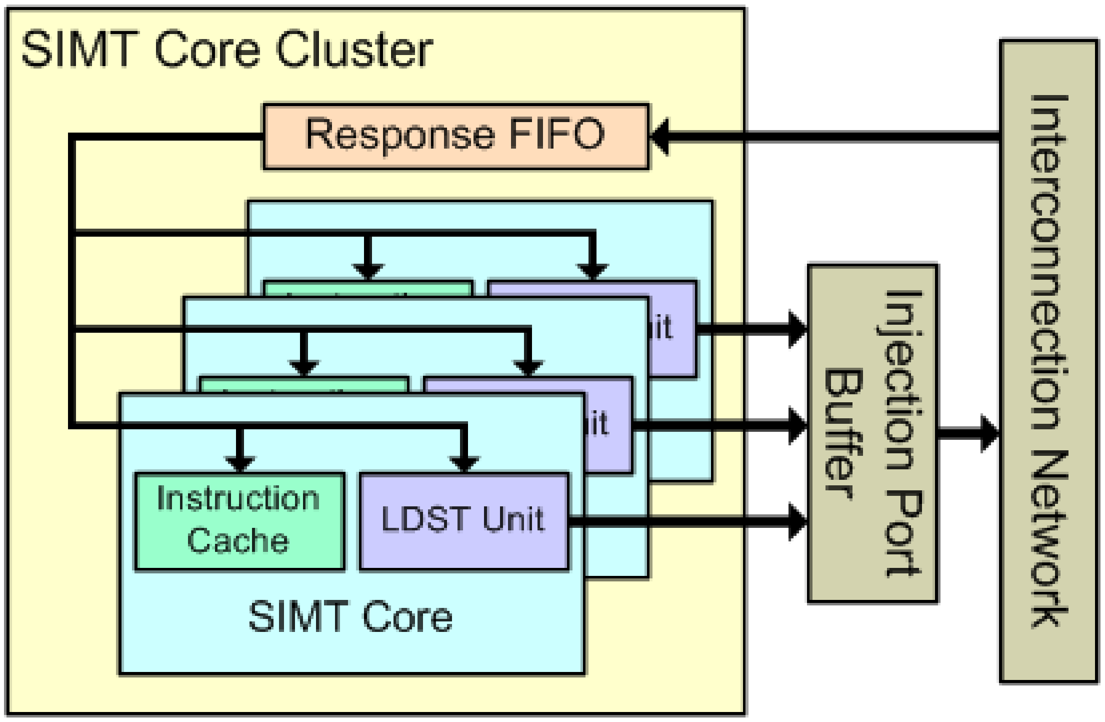

Figure 3.2 Cluster Organization [1]

Each SIMT core, shown in Figure 3.3, is a highly multithreaded and pipelined, Single Instruction Multiple Data processor designed to model an SM. The core is split up into four major parts: the front end, register access and operand collector, the ALU pipelines, and the memory pipelines. The front end consists of the instruction fetch and decode, instruction issue, scoreboard, and the SIMT stack. An operand collector, something proprietary to NVIDIA, is a set of buffers and arbitration logic along with multiple banks of single ported RAMs, that provides the illusion of a multiported register file. The ALU pipelines consist of two types of functional units: the SPs and the SFUs. The SPs execute all ALU instruction save for trancendentals, and the SFUs handle only trancendentals. Each core contains one of each unit. The memory pipeline contains four different types of level 1 on-chip memories. The memories are: shared memory, data cache, constant cache, and texture cache. Though they are modelled as separate physical structures they all share the same writeback stage, due to being components of the pipeline. [1]

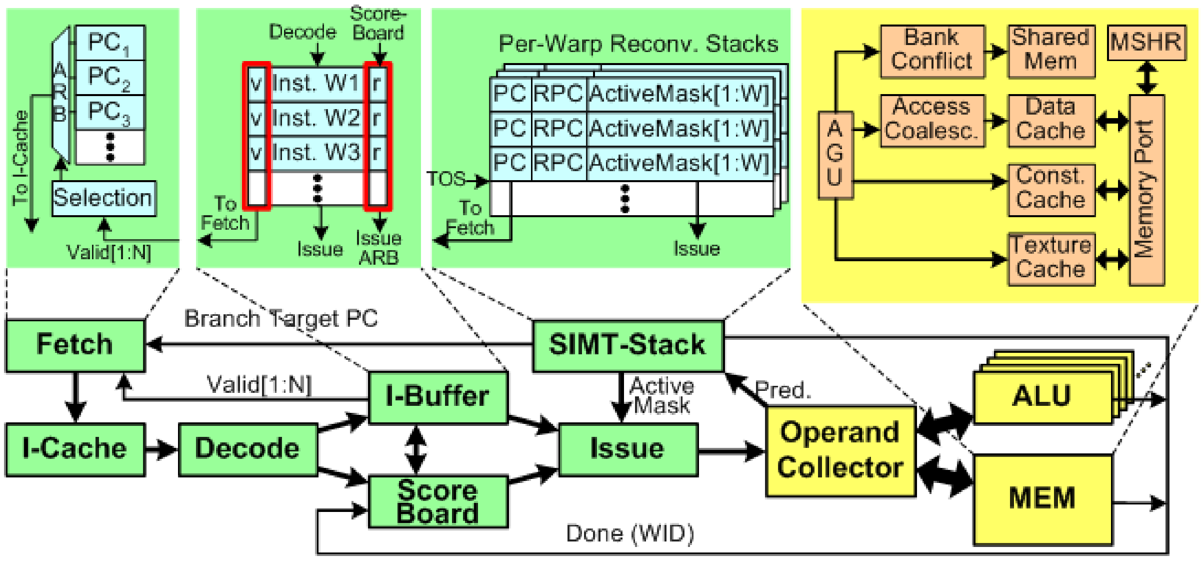

Figure 3.3 SIMT Core Organization [1]

Almost all of the pieces and components mentioned or shown above have been made to be configurable by changing values in the configuration file `gpgpusim.config`.

## 3.2 Creating the Model

The first step was to compile literature on the new Maxwell architecture. Unfortunately, not a lot of such literature is available, especially for the specific GPU that is to be modelled, the GTX Titan X. Due to this, some of the information regarding the Maxwell architecture and the GTX Titan X had to be acquired from less academic sources, such as graphics card review websites. After compiling the liturature, a comparison between the Maxwell architecture and the Fermi architecture could be made.

The differences between the GTX Titan X and the GTX 480 can be seen in Table 3.1. Information regarding the GTX Titan X was compiled from [27, 28, 16, 15, 2, 37], and information for the GTX 480 was compiled from [24, 14]. Note that this is not a complete description of the differences between the two cards, as there were changes in the organization of the streaming multiprocessors, cache configurations, shared memory organization, and clock domains, as detailed in Chapter 2.

In the new organization of streaming multiprocessors each Maxwell SM is divided into four processing blocks whereas a Fermi SM is not divided in any way. A good way to illustrate this change is to see the two SMs side by side, as in Figure 3.4.

Maxwell takes a departure from both Kepler and Fermi in that shared memory and L1 cache no longer share resources. In Maxwell L1 cache is now combined with texture cache, and shared memory is a dedicated unit.

Starting with the Tesla architecture, NVIDIA made the decision to have the SMs in the chip run at twice the speed of the rest of the processor. This was done to improve performance without increasing the area of the chip. A change in clock domain happened with the introduction of the Kepler architecture. Now, the SMs run at the same clock speed as the rest of the chip. Another clock feature that Maxwell retained from Kepler is the idea of a boost clock. Modelling this concept was not attempted for this project.

Two different options for modelling the GTX Titan X were considered. One model treated the SMM as though it had no clusters, meaning that all four schedulers can see all the different execution blocks in the SMM. This is the same configuration that is used for Fermi in GPGPU-Sim. Another, more conservative, model treats each processing block inside the SMM as a smaller SM, and the resources that it normally shares with the other processing blocks will be reduced accordingly. For both GTX Titan X models considered, it is assumed that the instruction latencies are not any worse than the latencies in the model for the GTX 480, i.e., the number of cycles taken to execute an instruction in the Maxwell architecture is not larger than the number of cycles taken to execute the same instruction in the Fermi architecture. Ultimately, the conservative model was used for this project. The configuration file `gpgpusim.config` was updated with the values needed for the desired model. The configuration options, along with their previous and updated values, are shown in table 3.2.

Table 3.1 The differences between the Fermi(GF100) and Maxwell(GM200) architectures

| **Top level differences** | **GTX 480** | **GTX Titan X** |
|---|---|---|
| Number of SMs | 15 | 24 |
| Clock Domain(core/shader) in MHz | 700/1400 | 1000/1000 |
| Memory clock (in MHz) | 924 | 1753 |
| L2 cache size(in MB) | 1.5 | 3 |
| Number of DRAM chips | 12 | 24 |
| **SM level differences** | | |
| processing blocks per SM | 1 | 4 |
| Registers per SM | 32768 | 65536(16384 per block) |
| Warps per SM | 48 | 64 |
| CTAs per SM | 8 | 32 |
| SPs per SM | 32 | 128 |
| LD/ST units per SM | 16 | 32 |
| SFUs per SM | 4 | 32 |
| L1 cache size | 16/32KB | 24KB |
| Warp schedulers | 2 | 4 |
| Dispatch units per scheduler | 1 | 2 |
| Shared memory size | 16/32KB | 96KB |

## 3.3 Testing the Model

To test the GTX Titan X model a selection of benchmarks from the Rodinia Benchmark suite, along with a few programs from the NVIDIA CUDA SDK, were used. The programs chosen are enumerated in Table 3.3. Each of the programs has a minimum of one kernel that will be executed on the GPU. To get an accurate assessment of the performance of the model, the parameters of the programs were chosen such that the majority of the kernels in each program would run for a sufficiently long time for the model to reach a steady state. The results from any kernel that did not reach the steady state were discarded.

The benchmarks were first run on the GPGPU-Sim GTX Titan X model. While the simulator was running the benchmarks, we ran the same benchmark programs one after another on a different machine that had an actual GTX Titan X card. Each benchmark program running on the GTX Titan X card was run through the NVIDIA profiler to profile the hardware's

Table 3.2 Options for GPGPU-Sim that were updated

| **Configuration Option** | **GTX 480** | **GTX Titan X** |
|---|---|---|
| -gpgpu_n_clusters | 15 | 96 |
| -gpgpu_clock_domains | 700.0:700.0:700.0:924.0 | 1000.0:1000.0:1000.0:1753.0 |
| -gpgpu_shader_registers | 32768 | 16384 |
| -gpgpu_shader_core_pipeline | 1536:32 | 512:32 |
| -gpgpu_pipeline_widths | 2,1,1,2,1,1,2 | 1,1,1,1,1,1,2 |
| -gpgpu_num_sp_units | 2 | 1 |
| -gpgpu_cache:dl1 | 32:128:4,L:L:m:N:H,A:32:8,8 | 16:128:4,L:L:m:N:L,A:32:8,8 |
| -gpgpu_shmem_size | 49152 | 24576 |
| -gpgpu_cache:dl2 | 64:128:8,L:B:m:W:L,A:32:4,4:0,32 | 256:128:8,L:B:m:W:L,A:32:4,4:0,32 |
| -gpgpu_tex_cache:l1 | 4:128:24,L:R:m:N:L,F:128:4,128:2 | 16:128:24,L:R:m:N:L,F:128:4,128:2 |
| -gpgpu_const_cache:l1 | 64:64:2,L:R:f:N:L,A:2:32,4 | 128:64:2,L:R:f:N:L,A:2:32,4 |
| -gpgpu_operand_collector_num_units_sp | 6 | 4 |
| -gpgpu_operand_collector_num_units_sfu | 8 | 4 |
| -gpgpu_num_reg_banks | 16 | 8 |
| -gpgpu_max_insn_issue_per_warp | 1 | 2 |
| -inter_config_file | config_fermi_islip.icnt | config_maxwell_islip.icnt |
| -gpgpu_dram_return_queue_size | 116 | 160 |
| -gpgpu_n_mem_per_ctrlr | 2 | 4 |
| -gpgpu_dram_buswidth | 4 | 2 |
| -gpgpu_num_sched_per_core | 2 | 1 |

execution of the program. Since IPC is the metric used by [5] to test for performance parity between the simulator and the hardware, it was the same metric that we used. Output of the benchmarks run on the simulator were stored in log files, while the output of the NVIDIA profiler was stored in binary files that are readable by NVIDIA's Visual Profiler program. The Visual Profiler was then used to view the output of the profiler and export the results to a comma separated value file.

GPGPU-Sim calculates IPC over the entire GPU where the number of instructions being executed is incremented per thread. NVIDIA's profiler program calculates the IPC of the execution of a program run on a GPU by using the formula:

$$ipc = (inst\_executed/\#SM)/elapsed\_clocks \tag{3.1}$$

which is found in [26]. Note the $\#SM$ in the above equation; this means that the IPC value generated by the profiler is per SM as opposed to the IPC of the overall GPU. Also, as noted in [45], the value of *inst_executed* is incremented per warp as opposed to per thread. In order to get IPC values for the hardware that are comparable to the values calculated by GPGPU-Sim,

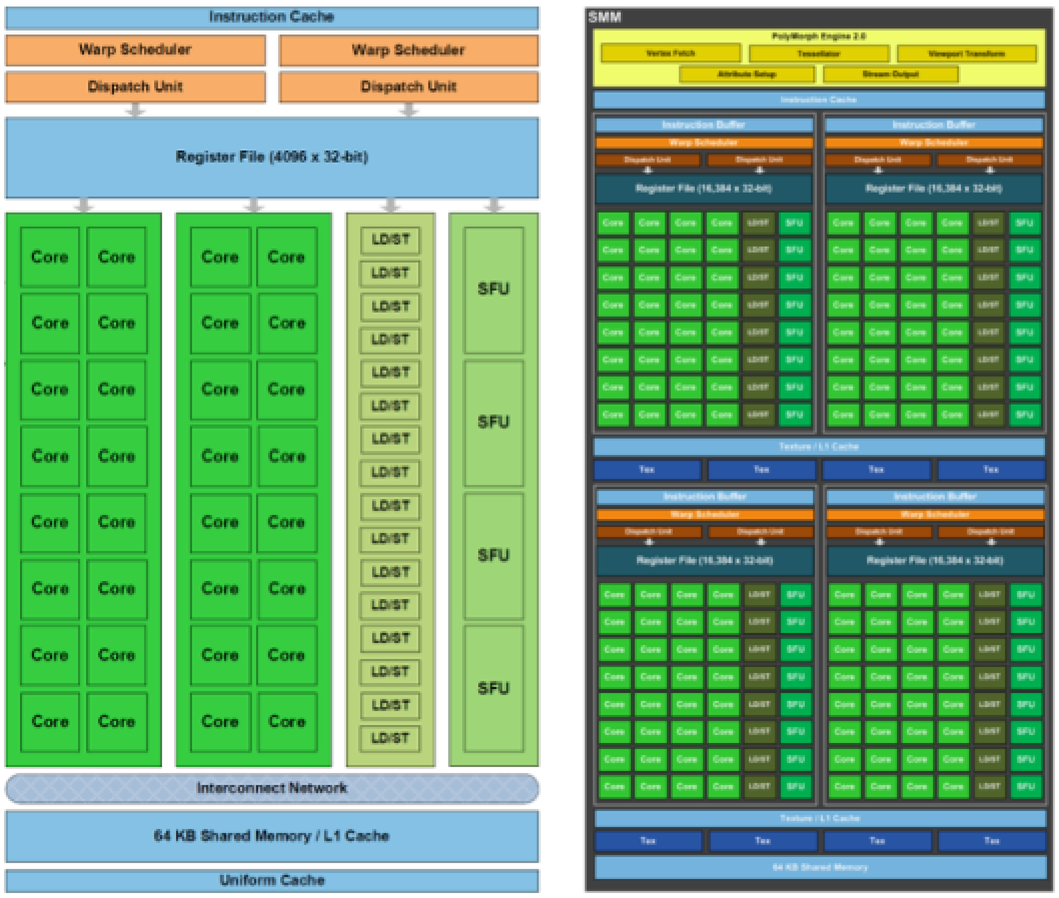

Figure 3.4 The SM in Fermi(left) vs. the SM in Maxwell(right) [14, 28]

the IPC value returned by the profiler will need to be modified in the following manner.

$$thread\_level\_ipc = ipc \cdot 32 \cdot \#SM \tag{3.2}$$

The IPC values obtained from the profiler were converted to thread-level IPC values using the above formula.

Table 3.3 The benchmarks chosen to test the model

| **Software Suite** | **Program Name** |
|---|---|
| Rodinia | backprop |
| | b+tree |
| | hotspot |
| | kmeans |
| | LUD |
| | nearest neighbor |
| | pathfinder |
| | srad |
| CUDA SDK | BlackSholes |
| | scalarProd |
| | histogram |

# CHAPTER 4. RESULTS

This chapter summarizes and analyzes the results from running the benchmarks. In particular, major discrepancies between the results from the simulation and results from the hardware are noted and possible explanations for the discrepancies are given.

## 4.1 Results from the simulator

The results from running the selection of programs from both the Rodinia Benchmark suite and the NVIDIA CUDA SDK on the simulator are shown in Table 4.1. Results are given for each individual kernel that is executed for each program. The result is the number of instructions executed per cycle at the thread level, over the entire GPU, by the simulator.

Table 4.1 IPC of the benchmark programs ran on GPGPU-Sim

| **Program** | **IPC** |
|---|---|
| nn | 306.6863 |
| backprop-1 | 2331.9951 |
| backprop-2 | 638.3974 |
| hotspot | 1079.0985 |
| pathfinder | 1887.0366 |
| srad-1 | 897.3112 |
| srad-2 | 831.0049 |
| lud-3 | 402.0011 |
| b+tree-1 | 797.889 |
| b+tree-2 | 842.8557 |
| kmeans-1 | 13.4951 |
| kmeans-2 | 501.9459 |
| Blacksholes | 1007.2 |
| scalarProd | 307.3682 |
| histogram-1 | 1291.2040 |

## 4.2 Results from the GTX Titan X

The results from running the selection of programs from both the Rodinia Benchmark suite and the NVIDIA CUDA SDK on the GTX Titan X are shown in Table 4.2. Again, results are given for each individual kernel that is executed by each program. The result is the calculated number of instructions executed per cycle at the thread level by the entire GPU using Equation 3.2 provided in Chapter 3.

Table 4.2 IPC of the benchmark programs ran on the GTX Titan X

| **Program** | **overall IPC(calculated)** |
|---|---|
| nn | 792.576 |
| backprop-1 | 2405.376 |
| backprop-2 | 269.568 |
| hotspot | 703.488 |
| pathfinder | 1863.168 |
| srad-1 | 1271.808 |
| srad-2 | 1294.08 |
| lud-3 | 2090.496 |
| b+tree-1 | 690.432 |
| b+tree-2 | 767.232 |
| kmeans-1 | 39.936 |
| kmeans-2 | 514.56 |
| Blacksholes | 589.056 |
| scalarProd | 415.488 |
| histogram-1 | 1443.072 |

## 4.3 Comparison of the data

The results above are normalized by dividing each IPC value by the IPC value calculated for the GTX Titan X for that kernel. This results in a normalized IPC value of 1 for all GTX Titan X kernel executions, making it easy to compare it to the corresponding normalized GPGPU-Sim IPC value. Comparing the normalized results from the simulated model with the results from the GXT Titan X, as is done in Figure 4.1, we can see that the model is indeed a conservative model. Nine of the fifteen kernels executed ran slower on the model than on the GTX Titan X, with three of these performing much slower than the hardware. Six out of the

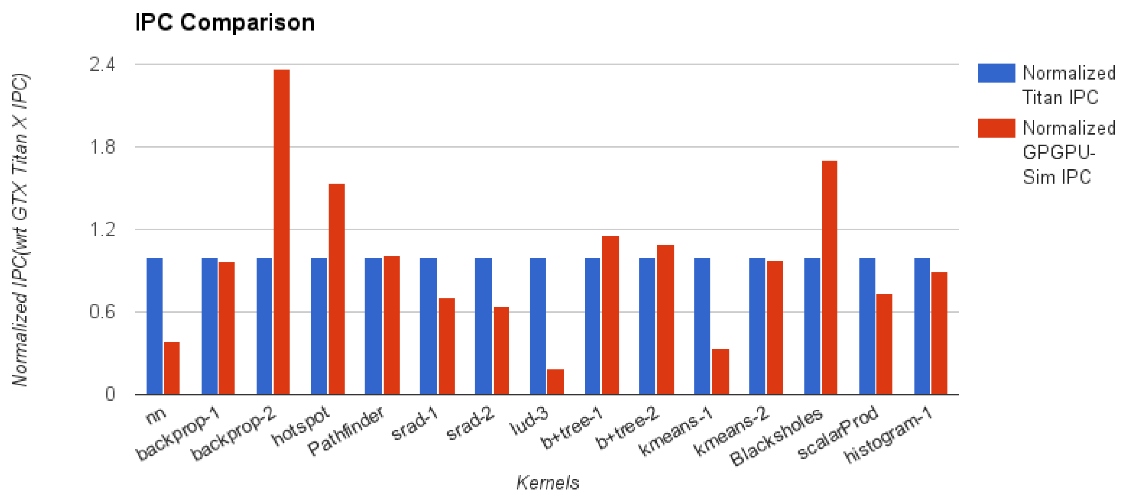

Figure 4.1 Normalized IPC Comparison

fifteen kernels achieved a model IPC value that was within 16% of what the hardware achieved. There are three kernels for which the model performed much faster than the GTX Titan X, namely hotspot, backprop-2, and BlackScholes. Each of these performed at least 50% faster in the model than on the hardware. There were also three kernels for which the model performed much slower than the hardware, namely kmeans-1, lud-3, and nn, achieving a model IPC that was less than 40% of what the hardware achieved.

### 4.3.1 Analysis of Outliers

Using the NVIDIA profiler, with options set to collect data regarding instructions executed, and analysing the code of the kernels that performed much faster or much slower on the model compared to the hardware, provides insight regarding their extraordinary performance on the simulator. We will analyze each kernel individually.

#### 4.3.1.1 backprop-2

We will start with discussing the second backprop kernel's performance. From the source code, shown in Listing 4.1, and its disassembled version, it is clear that the main computation of the kernel is comprised of single precision floating point operations. However, running the program with the profiler shows the mix of computation and memory instructions displayed in Table 4.3. Note that there were no single precision instructions executed. This means that the GTX Titan X executed the main computation as double precision floating point operations. As mentioned in [15], the double precision floating point performance of Maxwell is 1/32 that of single precision. This is eight times slower than the double precision performance of Fermi, which is 1/4 of single precision. Hence, there are two possible explanations for the discrepancy. One is that all floating point instructions in the kernel are executed as double precision on the hardware, whereas the model executes them as single precision. Another possibility is that both the hardware and the model execute the floating point instructions as double precision. However, the model uses the instruction latencies of the Fermi architecture, so it could evaluate more instructions per cycle than the hardware does.

```
__global__ void bpnn_adjust_weights_cuda(float * delta,
                                          int hid,
                                          float * ly,
                                          int in,
                                          float * w,
                                          float * oldw)
{

   int by = blockIdx.y;

   int tx = threadIdx.x;
   int ty = threadIdx.y;

   int index =  ( hid + 1 ) * HEIGHT * by + ( hid + 1 )
               * ty + tx + 1 + ( hid + 1 ) ;
   int index_y = HEIGHT * by + ty + 1;
   int index_x = tx + 1;

   w[index] += ((ETA * delta[index_x] * ly[index_y])
               + (MOMENTUM * oldw[index]));
   oldw[index] = ((ETA * delta[index_x] * ly[index_y])
               + (MOMENTUM * oldw[index]));



   __syncthreads();


   if (ty == 0 && by ==0){
   w[index_x] += ((ETA * delta[index_x])
               + (MOMENTUM * oldw[index_x]));
   oldw[index_x] = ((ETA * delta[index_x])
               + (MOMENTUM * oldw[index_x]));
   }

}
```

Listing 4.1 bpnn_adjust_weights_cuda

Table 4.3 Percentage of Instructions Executed by the bpnn_adjust_weights Kernel

| Memory inst % | FP32 inst % | FP64 inst % | Int inst % |
|---|---|---|---|
| 20.31250573 | 0 | 10.9374993 | 42.18747453 |

#### 4.3.1.2 hotspot

The next kernel for which the model performed much faster than the GTX Titan X is the hotspot temperature calculation kernel. It is possible that the discrepancy here is the result of a situation similar to the one encountered in the backprop kernel analyzed previously. Reading the source and the disassembled object code, none of the computational instructions were double precision instructions. Running the program with the profiler yields the mix of computation and memory instructions shown in Table 4.4. The disassembled object code, a snippet of which is shown in Listing 4.2, produced 197 instructions, with 18 single precision floating point instructions. Using the executed instruction mix in the table as a reference, approximately 7 of the 18 single precision floating point instructions are executed as double precision instructions. The cumulative number of floating point multiply and multiply-adds is

7. It is likely that the GTX Titan X executes these instructions as double precision instructions to preserve the precision of the output.

Table 4.4 Percentage of Instructions Executed by the calculate_temp Kernel

| **Memory inst %** | **FP32 inst %** | **FP64 inst %** | **Int inst %** |
|---|---|---|---|
| 12.80139185 | 5.452961578 | 3.797714708 | 45.29189201 |

```
... /*Floating point operations start here*/
/*0228*/         FSET.C1 o [0x7f], |g| [0x10], c [0x1] [0x1], GT;
/*0230*/         MOV R1, g [0x14];
/*0238*/         MOV R2, g [0x10];
/*0240*/         MOV R11 (C0.NOT_SIGN), R124;
/*0248*/         FMUL R1 (C1.NEU), g [0x14], c [0x1] [0x3];
/*0250*/         FMUL R2 (C1.NEU), g [0x10], c [0x1] [0x3];
/*0258*/         RCP32 R2, R2;
/*025c*/         FMUL32 R12, R1, R2;
/*0260*/         PBK 0x558;
/*0268*/         MOV R2, g [0x13];
/*0270*/         MOV R3, g [0x12];
/*0278*/         IADD32I R17, R7, 0xffffffff;
/*0280*/         IADD32I R18, R7, 0x1;
/*0288*/         MOV R4, g [0x11];
/*0290*/         IADD32I R14, g [0x4], 0xffffffff;
/*0298*/         MOV R1, R124;
/*02a0*/         I2I.U32.U16 R13, R0L;
/*02a8*/         RCP32 R15, R2;
/*02ac*/         RCP32 R16, R3;
/*02b0*/         IMAX.S32 R17, R17, R8;
/*02b8*/         IADD32I R2, R13, 0xffffffff;
/*02c0*/         IADD32I R3, R13, 0x1;
/*02c8*/         IMIN.S32 R18, R18, R9;
/*02d0*/         RCP R4, R4;
/*02d8*/         IMAX.S32 R19, R2, R11;
/*02e0*/         IMIN.S32 R20, R3, R10;
/*02e8*/         IADD32I R2, R1, 0x1;
```

```
/*02f0*/        ISET.S32.C0 o [0x7f], R2, R13, GT;
/*02f8*/        SSY 0x4a8;
/*0300*/        MOV R3, R124;
/*0308*/        BRA C0.NE, 0x4a8;
/*0310*/        IADD32I R21, -R1, 0xe;
/*0318*/        ISET.S32.C0 o [0x7f], R21, R13, LT;
/*0320*/        BRA C0.NE, 0x4a8;
/*0328*/        ISET.S32.C0 o [0x7f], R2, R7, GT;
/*0330*/        BRA C0.NE, 0x4a8;
/*0338*/        ISET.S32.C0 o [0x7f], R21, R7, LT;
/*0340*/        BRA C0.NE, 0x4a8;
/*0348*/        ISET.S32.C0 o [0x7f], R11, R13, GT;
/*0350*/        BRA C0.NE, 0x4a8;
/*0358*/        ISET.S32.C0 o [0x7f], R10, R13, LT;
/*0360*/        BRA C0.NE, 0x4a8;
/*0368*/        ISET.S32.C0 o [0x7f], R8, R7, GT;
/*0370*/        BRA C0.NE, 0x4a8;
/*0378*/        ISET.S32.C0 o [0x7f], R9, R7, LT;
/*0380*/        BRA C0.NE, 0x4a8;
/*0388*/        SHL R3, R7, 0x4;
/*0390*/        SHL R23, R18, 0x4;
/*0398*/        IADD R21, R13, R3;
/*03a0*/        SHL R22, R17, 0x4;
/*03a8*/        IADD R24, R13, R23;
/*03b0*/        R2A A2, R21, 0x2;
/*03b8*/        IADD R23, R13, R22;
/*03c0*/        R2A A3, R24, 0x2;
/*03c8*/        IADD32 R22, R3, R20;
/*03cc*/        IADD32 R21, R3, R19;
/*03d0*/        R2A A1, R23, 0x2;
/*03d8*/        MOV R3, g [A3+0x16];
/*03e0*/        R2A A4, R22, 0x2;
/*03e8*/        R2A A3, R21, 0x2;
/*03f0*/        F2F.F32.F32 R21, g [A2+0x16];
/*03f8*/        FADD R22, g [A1+0x16], R3;
```

```
/*0400*/        MOV R3, g [A4+0x16];
/*0408*/        ADA A1, A2, 0x458;
/*0410*/        FADD R23, R21, R21;
/*0418*/        F2F.F32.F32 R22, R22;
/*0420*/        FADD R3, g [A3+0x16], R3;
/*0428*/        F2F.F32.F32 R24, R16;
/*0430*/        F2F.F32.F32 R26, g [A1+0x0];
/*0438*/        FADD R25, -R23, R22;
/*0440*/        F2F.F32.F32 R22, R3;
/*0448*/        FADD R3, -g [A2+0x16], c [0x1] [0x4];
/*0450*/        FMAD R24, R24, R25, R26;
/*0458*/        FADD R23, R22, -R23;
/*0460*/        F2F.F32.F32 R22, R4;
/*0468*/        FMUL R3, R15, R3;
/*0470*/        FMAD R22, R22, R23, R24;
/*0478*/        F2F.F32.F32 R3, R3;
/*0480*/        FADD R22, R3, R22;
/*0488*/        F2F.F32.F32 R3, R12;
/*0490*/        FMAD R3, R3, R22, R21;
/*0498*/        R2G.U32.U32 g [A2+0x216], R3;
...
```

Listing 4.2 Snippet of the calculate_temp CUDA Assembly

#### 4.3.1.3 BlackScholes

The last of the kernels that performs much faster in the model than on the GTX Titan X is BlackScholes. Similar to the previous two kernels, all computations on the data (as opposed to index calculations) use single precision floating point instructions. Execution of the program with the profiler yields the instruction mix shown in table 4.5. Unlike the previous two kernels, none of the single precision instructions were executed as double precision instructions.

Looking more closely at the data collected by the profiler, it is noted that some of the memory instructions were local loads and local stores. Consulting the source code did not show any such operations. Upon disassembling the program, it is revealed that there are two versions

Table 4.5 Percentage of Instructions Executed by the BlackScholesGPU Kernel

| **Memory inst %** | **FP32 inst %** | **FP64 inst %** | **Int inst %** |
|---|---|---|---|
| 23.43096234 | 37.40081659 | 0 | 17.57322176 |

of the kernel assembly code. One version is for the sm_10 architecture, and the other is for the sm_20 architecture. The sm_10 architecture version of the kernel performs only global loads and global stores, whereas the sm_20 architecture version performs global and local memory operations. The local memory operations seem to be used for storing intermediate values in a calculation. Based on this information, it safe to assume that the GTX Titan X executed the sm_20 version of the kernel. The output file of the model simulation makes it clear that it executes the sm_10 version of the assembly code. While not taking as long as a global memory operation, a local memory operation has a longer latency than reading intermediate values from a register. Hence the sm_20 architecture version can be expected to run slower due to storing intermediate values in local memory.

#### 4.3.1.4 kmeans-1

Now, the kernels that performed much slower in the model than on the GTX Titan X will be analyzed. The invert_mapping kernel, referred to previously as kmeans-1, will be analyzed first. Reviewing the source code and the disassembled object code, a block of code has been identified, shown in Listing 4.3, where memory operations are performed in rapid succession. One possible reason for why the model does not have performance parity with the hardware is that although the size of the L2 cache in the model has been increased to match that of the hardware, the bandwidth to handle memory transfers between the cores and the memory partition has not been increased accordingly.

```
...
/*0088*/          GLD.U32 R5, global14 [R0];
/*0090*/          IADD32I R3, R3, 0x1;
/*0098*/          GST.U32 global14 [R1], R5;
/*00a0*/          ISET.S32.C0 o [0x7f], R3, R4, NE;
```

```
/*00a8*/          IADD32I R0, R0, 0x4;
/*00b0*/          IADD R1, R2, R1;
/*00b8*/          BRA C0.NE, 0x88;
...
```

Listing 4.3 Snippet of the invert_mapping CUDA Assembly

#### 4.3.1.5 lud-3

The next kernel that performed much slower in the model than on the GTX Titan X is the lud-3 kernel. The lud_internal kernel suffers from an instruction execution block similar to the one seen in the kmeans-1 kernel. This block of code is shown in Listing 4.4. Here, rather than a global load followed almost immediately by a global store, it contains two global loads followed by two stores to shared memory. The same reasoning as above applies to the global loads, i.e., the bandwidth to handle memory transfers has not been increased to match that of the hardware. Another possible reason that could cause such a slowdown in the model is that the model has a reduced shared memory size. This might cause more shared memory bank conflicts than in the hardware, contributing to the slowdown.

```
...
/*00d0*/          GLD.U32 R4, global14 [R4];
/*00d8*/          R2A A1, R5, 0x6;
/*00e0*/          R2A A2, R1, 0x2;
/*00e8*/          IADD R1, g [0x4], R6;
/*00f0*/          GLD.U32 R1, global14 [R1];
/*00f8*/          R2G.U32.U32 g [A2+0x108], R4;
/*0100*/          R2G.U32.U32 g [A2+0x8], R1;
/*0108*/          BAR.ARV.WAIT b0, 0xfff;
...
```

Listing 4.4 Snippet of the lud_internal CUDA Assembly

#### 4.3.1.6 nearest neighbor

Finally, the last kernel that performs much slower in the model than on the hardware is the nearest neighbor kernel. Reading the source code and the disassembled object code, a block of code similar to what was seen in the previous two kernels is present in the euclid nearest neighbor kernel. This block is shown in Listing 4.5. Though not shown in the listing, the variable latLong is loaded from global memory. In the disassembled object code, a global load is performed for each element of the struct. The loads are performed one after the other, which is similar to the other two kernels. Therefore, it is not surprising that the nearest neighbor kernel performs much slower in the model.

```
...
if(globalId < numRecords){
        float *dist=d_distances+globalId;
        *dist = (float)sqrt((lat-latLong->lat)
                *(lat-latLong->lat) +(lng-latLong->lng)
                *(lng-latLong->lng));
}
...
```

Listing 4.5 Snippet of the euclid Kernel

There are other possible explanations for such drastic differences in performance for the last three kernels. One is that the memory hierarchy of the Fermi architecture is preserved in the model, which is not the same as the memory hierarchy in Maxwell. [16] Another possible explanation is that, as stated in [27], the memory bandwidth has been increased in Maxwell. Also worth noting, as stated in [28], is that Maxwell has improved memory compression algorithms to reduce bandwidth demands on DRAM. Whether or not this is used for computation workloads is uncertain, however.

# CHAPTER 5. RELATED WORKS

## 5.1 Research Performed using GPGPU-Sim

As mentioned previously, GPGPU-Sim is used for a wide variety of GPU architectural research and exploration. Though not exhaustive, a list of the areas in which GPGPU-Sim has been used is as follows: It has been used to validate the performance of multicore scaling on the GPU [11], to analyze GPU architectures with memory-level and thread-level parallelism awareness [18], to explore different scheduling algorithms and methods such as phase aware warp scheduling [4], or interleaving the execution of thread blocks from different kernels [3]. GPGPU-Sim is also used in various heterogeneous computing system simulators [34, 46], among other projects.

## 5.2 Other Simulators

Though this work is concerned with the GPGPU-Sim simulator, it is not the only general purpose computing GPU simulator or emulator. Among the other simulators and emulators are NVIDIAs own CUDA emulator, which was initially created to help with debugging CUDA programs. It became deprecated with the hardware debugging support of `cuda-gdb`. [25] GPU Ocelot, a just-in-time compilation framework for GPU compute applications that contains a PTX emulator [10], is one of the other emulators available. It has been used for analyzing PTX kernels [19], modeling heterogenous workloads and systems [20], and GPGPU application performance tuning [43], among other projects. ATTILA is a cycle-accurate simulator that simulates graphical workloads on a replica of modern architectures. [9] ATTILA has been used to analyse the effects of die stacking on GPU performance [13], to test different schemes of precision reduction in pixel shaders [33], to analyse predictive power gating techniques of shader processors to reduce power consumption [42], and to evaluate precision reduction of mobile graphics processing units and its effects on energy usage [32], along with other projects. Qsilver, a simulator that uses a cycle-timer to simulate data flow and computation through the GPU, is a similar project to ATTILA in that it deals only with graphics workloads. [35] It has been used to study thermal management for GPU architectures [36], and has been referenced by other works in the field of GPU architecture research. Another GPU simulator is Barra. It is an ISA-level functional simulator targeting the Tesla architecture. [7] These are not the only simulators, but they are ones that simulate only the GPU or functions of it.

There are other simulators that are available and used for research that use one of the aforementioned simulators to simulate a heterogeneous system. One such simulator is Gem5-gpu, a GPGPU heterogeneous CPU-GPU simulator developed at the University of Wisconsin-Madison. [34] It has been used to explore CPU-GPU coherence [39], and analyze microarchitecture effects on memory system behavior, [17] among other projects. Another very similar simulator to the Gem5-gpu is the FusionSim simulator developed at the University of Toronto. [46] Another heterogeneous simulator, Multi2Sim, is a combination of ISA-level functional simulation for the GPU and ISA-level simulation of superscalar, multi-threaded, and multicore processors. [41]

## 5.3 Benchmarking

To test the GTX Titan X model, a selection of programs from the Rodinia benchmark suite was used. [6] Other GPGPU or scientific computation benchmark suites include Parboil [38] and Scalable HeterOgeneous Computing, or SHOC. [8]

Benchmarking a system is a good way to get an idea of overall performance, but to get an idea of how a system performs at a lower level, a different kind of benchmarking is needed. Microbenchmarks are programs that are used to identify architectural features or program characteristics. To get a more accurate picture of how a GPU works, microbenchmarks are needed. They have been used to get a better understanding of Tesla-based GPUs. [44] A microbenchmark suite has been developed for AMD GPUs by [40].

# CHAPTER 6. CONCLUSION

## 6.1 Contributions

This paper began with a brief discussion of the advances in graphics architecture and graphics processors within the last ten years. Attention was directed to the fact that as these advances were made the simulator GPGPU-Sim was not advancing in tandem. To rectify this, an attempt was made to model a more modern graphics processor, specifically the GTX Titan X. This was accomplished though changes made in the configuration files that GPGPU-Sim uses. The updates to these files were derived from information about the GTX Titan X compiled from as many sources as possible, which included computer hardware review websites. With the configuration files for the most up-to-date architecture that GPGPU-Sim supports as a base, a configuration was assembled that yielded a model for the GTX Titan X. To test the accuracy of the model, a selection of programs, some of which have more than one kernel, from the Rodinia Benchmark suite and the CUDA SDK were ran on both the GTX Titan X and GPGPU-Sim. The metric used to measure the accuracy of the model is the overall IPC of the kernel, which was the same metric used by the creators of GPGPU-Sim to verify the accuracy of their models.

The results of the programs show that the model that was chosen was quite conservative, with 9 out of the 15 kernels that were considered running slower than the GTX Titan X. Out of the 15 kernels, only 6 achieved an IPC that is within 16% of the IPC achieved by the GTX Titan X. Of the 6 kernels that did not perform slower than the GTX Titan X, 3 performed at least 50% faster. These kernels had various reasons for performing faster. Two of those kernels had some, if not all, of the floating point calculations executed as double precision instructions on the hardware. Hence, the performance difference could be due to the fact that the instruction latencies assumed for the model are not correct, specifically that the newer Maxwell architecture has an increased latency for the double precision execution path. Another possibility is that the model executed those floating point calculations as single precision instructions, which are faster than double precision instructions. The other kernel that performed over 50% faster had a different version of the GPU code being executed on the hardware than on GPGPU-Sim. The version executed on the hardware had memory operations to local memory that were not present in the version that executed on the simulator. This caused the version that ran on the hardware to run slower. There were also kernels for which the model achieved an IPC less than 40% of what the GTX Titan X achieved. These kernels seem to follow an execution pattern where many loads or stores are performed in succession with a small amount of computations in between. Hence, one explanation is that the memory hierarchy and memory partition model of GPGPU-Sim is not sufficiently accurate to provide performance parity in such an execution pattern.

This paper concludes with a discussion of the different GPU simulators and emulators, such as GPGPU-Sim, NVIDIA's own emulator, Ocelot, ATTILA, Barra, and derivative simulators of these, and how they are used for research. The derivative simulators are Gem5-GPU and FusionSim. Finally, different benchmark suites and the different types of benchmarking, such as microbenchmarking, are also discussed.

## 6.2 Future Work

The work performed for this paper takes only the first steps toward creating a more accurate model of the GTX Titan X GPU. With this work, it has been shown that GPGPU-Sim in its current state is insufficient to accurately model the newer architecture. One area to work on is to perform microbenchmarks on the GTX Titan X to get accurate latencies for all the execution paths. Other areas that could be worked on include updating GPGPU-Sim's memory hierarchy, updating GPGPU-Sim's top level organization, and adding new clocking features, such as boost clock and different levels of clock gating. All areas just mentioned would involve making changes to the GPGPU-Sim source code. Updating the memory hierarchy would include separating the shared memory from L1 cache, combining L1 caching functionality with texture cache, and creating a separate unit for read-only data cache. Updating GPGPU-Sim's top level organization would include creating a new unit within the SIMT core that would work in the same manner as a processing block in the Maxwell architecture. Finally, adding new clocking features such as clock gating would require extra code in the different levels of organization to support the clocks of the cores or processing blocks; however, it is not clear whether adding such functionality would be practical.

Updating the memory hierarchy could be accomplished by the following changes to the source code: To create the new separate shared memory and read-only data cache, a few new variables would be added to the `shader.h` and `shader.cc` files to represent them. Combining the L1 caching functionality with the texture cache would require declaring the L1 data cache class as a friend class of the texture cache class. To utilize these changes, the load-store unit in the shader would need to be updated as well. The `cycle` function would need to be updated to reflect the changes made, such as adding the calls to the `cycle` functions for the new shared memory and read-only cache and having the L1 data functions be called from the texture cache.

To update the top level organization, a new processing block class containing the computational units, instruction buffer, register file, warp scheduler, and dispatch units would need to be created. This new class would exist inside the shader, and would connect to the other elements inside the shader through functions in that class.

# BIBLIOGRAPHY

[1] Aamodt, T. M., Fung, W. W. L., Singh, I., El-Shafiey, A., Kwa, J., Hetherington, T., Gubran, A., Boktor, A., Rogers, T., Bakhoda, A., and Jooybar, H. Gpgpu-sim 3.x manual. http://gpgpu-sim.org/manual/index.php/Main_Page, 2012.

[2] Angelini, C., and Wallossek, I. Geforce gtx titan x review: Can one gpu handle 4k? http://www.tomshardware.com/reviews/nvidia-geforce-gtx-titan-x-gm200-maxwell,4091.html, 2015.

[3] Awatramani, M., Zambreno, J., and Rover, D. Increasing gpu throughput using kernel interleaved thread block scheduling. In *Proceedings of the International Conference on Computer Design (ICCD)* (October 2013).

[4] Awatramani, M., Zhu, X., Zambreno, J., and Rover, D. Phase aware warp scheduling: Mitigating effects of phase behavior in gpgpu applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)* (October 2015).

[5] Bakhoda, A., Yuan, G. L., Fung, W. W., Wong, H., and Aamodt, T. M. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 163–174.

[6] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), IEEE, pp. 44–54.

[7] Collange, S., Defour, D., and Parello, D. Barra, a Parallel Functional GPGPU Simulator. working paper or preprint, Feb. 2009.

[8] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (New York, NY, USA, 2010), GPGPU-3, ACM, pp. 63–74.

[9] Del Barrio, V. M., González, C., Roca, J., Fernández, A., and Espasa, R. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on* (2006), IEEE, pp. 231–241.

[10] Diamos, G., Kerr, A., and Yalamanchili, S. Gpu ocelot: a binary translation framework for ptx. http://code.google.com/p/gpuocelot/, 2009.

[11] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on* (June 2011), pp. 365–376.

[12] Fung, W. W., Sham, I., Yuan, G., and Aamodt, T. M. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007), IEEE Computer Society, pp. 407–420.

[13] Fung, W. W., Sham, I., Yuan, G., and Aamodt, T. M. Dynamic warp formation: Efficient mimd control flow on simd graphics hardware. *ACM Transactions on Architecture and Code Optimization (TACO) 6*, 2 (2009), 7.

[14] Glaskowsky, P. N. Nvidia's fermi: The first complete gpu computing architecture. http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf, 2009.

[15] Hagedoorn, H. Nvidia geforce gtx titan x review. http://www.guru3d.com/articles-pages/nvidia-geforce-gtx-titan-x-review.html, 2015.

[16] Harris, M. Maxwell: The most advanced cuda gpu ever made. https://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/, 2014.

[17] Hestness, J., Keckler, S. W., and Wood, D. A. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on* (2014), IEEE, pp. 150–160.

[18] Hong, S., and Kim, H. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News 37*, 3 (June 2009), 152–163.

[19] Kerr, A., Diamos, G., and Yalamanchili, S. A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (oct. 2009), pp. 3 –12.

[20] Kerr, A., Diamos, G., and Yalamanchili, S. Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (New York, NY, USA, 2010), GPGPU 10, ACM, p. 3142.

[21] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (2008), 39–55.

[22] Moya, V., Gonzalez, C., Roca, J., Fernandez, A., and Espasa, R. Shader performance analysis on a modern gpu architecture. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on* (2005), IEEE, pp. 10–pp.

[23] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with cuda. *Queue 6*, 2 (Mar. 2008), 40–53.

[24] NVIDIA. Fermi: Nvidia's next generation cuda compute architecture. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[25] NVIDIA. Cuda toolkit 3.0 downloads. https://developer.nvidia.com/cuda-toolkit-30-downloads, March 2010.

[26] NVIDIA. Profiler user's guide. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf, 2012.

[27] NVIDIA. Nvidia geforce gtx 750 ti: Featuring first generation maxwell gpu technology, designed for extreme performance per watt. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014.

[28] NVIDIA. Nvidia geforce gtx 980: Featuring maxwell, the most advanced gpu ever made. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014.

[29] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110/210. http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf, 2014.

[30] NVIDIA. Cuda c programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2015.

[31] NVIDIA. Nvidia tesla p100: The most advanced datacenter accelerator ever built: Featuring pascal gp100, the world's fastest gpu. http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[32] Pool, J., Lastra, A., and Singh, M. Energy-precision tradeoffs in mobile graphics processing units. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on* (Oct 2008), pp. 60–67.

[33] Pool, J., Lastra, A., and Singh, M. Precision selection for energy-efficient pixel shaders. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, ACM, pp. 159–168.

[34] Power, J., Hestness, J., Orr, M., Hill, M., and Wood, D. gem5-gpu: A heterogeneous cpu-gpu simulator. *Computer Architecture Letters 13*, 1 (Jan 2014).

[35] Sheaffer, J. W., Luebke, D., and Skadron, K. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (New York, NY, USA, 2004), HWWS '04, ACM, pp. 85–94.

[36] Sheaffer, J. W., Skadron, K., and Luebke, D. P. Studying thermal management for graphics-processor architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* (March 2005), pp. 54–65.

[37] Smith, R. The nvidia geforce gtx titan x review. http://www.anandtech.com/show/9059/the-nvidia-geforce-gtx-titan-x-review/, 2015.

[38] Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and Hwu, W.-m. W. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing 127* (2012).

[39] Subramanian, G., Thakker, U., Haria, S., Shukla, R., and Lin, H. Exploring cpu-gpu coherence.

[40] Taylor, R., and Li, X. A micro-benchmark suite for amd gpus. In *2010 39th International Conference on Parallel Processing Workshops* (Sept 2010), pp. 387–396.

[41] Ubal, R., Jang, B., Mistry, P., Schaa, D., and Kaeli, D. Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2012), PACT '12, ACM, pp. 335–344.

[42] Wang, P. H., Chen, Y. M., Yang, C. L., and Cheng, Y. J. A predictive shutdown technique for gpu shader processors. *IEEE Computer Architecture Letters 8*, 1 (Jan 2009), 9–12.

[43] Wen-mei Hwu, e. a. *GPU Computing GEMS Jade Edition, 1st Edition*. Morgan Kaufmann, 2011, ch. 30.

[44] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 235–246.

[45] Ziegler, G. Instruction limited kernels. http://on-demand.gputechconf.com/gtc-express/2011/presentations/Inst_limited_kernels_Oct2011.pdf, 2011.

[46] Zkharenko, V. Fusionsim simulator. http://www.fusionsim.ca/home, 2012.