Summer 2019

A configurable general purpose graphics processing unit for power, performance, and area analysis

Garrett Lies

A Configurable General Purpose Graphics Processing Unit

for Power, Performance, and Area Analysis

by

Garrett Joseph Lies

A Creative Component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Very-Large Scale Integration)

Program of Study Committee:
Joseph Zambreno, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

Thank you to Dr. Joseph Zambreno, my major professor, who guided me throughout this project. You encouraged me to learn, and gave me the help I needed to complete this project. Thank you to all of my colleagues working on their own creative components under the supervision of Dr. Zambreno. Your feedback and references made this work possible in the amount of time we had.

ABSTRACT

Graphics Processing Units (GPUs) have become increasingly popular for graphical and general purpose applications. Researchers rely on simulators to evaluate potential hardware improvements. Existing GPU simulators simplify details of the architecture which leads to inaccuracies at low levels of abstraction. To properly evaluate the effects on power, performance and area, more accurate simulators are required. In this paper we present a new configurable design created with a hardware description language to model the Nvidia Tesla architecture at the register transfer level. Parameters and modularity allow for new hardware designs to be generated quickly. This new design is capable of simulating a subset of parallel thread execution instructions. Using our new design, we provide insight in how scaling the number of processors affects performance and power per computation. A synthesizable design modeled after the Tesla architecture allows users to better evaluate the effectiveness of architectural changes by comparing commercial simulation and synthesis results.

CHAPTER 1. INTRODUCTION

1.1 Motivation

Power, performance, and area (PPA), are three of the most important metrics to a hardware engineer. Hardware engineers aim to design systems which meet performance requirements while minimizing power and area overhead. These three metrics are tightly connected, and their trade-offs are well analyzed in the design of computing platforms. For example, a performance increase may come with an increase in area. This increase in area can result in fewer processors on a chip, and end in a net loss of performance. Power often scales with area, and is a growing concern as transistor sizes continue to shrink. GPUs contain a lot of duplicate instances, so a minor change can have a large impact on a design. To effectively evaluate changes to hardware, engineers need to evaluate the effects those changes have on PPA.

In 2001 most GPUs took advantage of a single instruction multiple data (SIMD) execution model [2]. GPUs processed vectors of data in parallel. In 2006, Nvidia released the Tesla architecture which changed the execution model from SIMD to single instruction multiple thread (SIMT) [1]. Newer architectures such as the Pascal architecture [3], released in 2016, expand upon the hardware, but share many similarities with its 2006 predecessor. Both use the compute unified device architecture (CUDA) to execute threads in parallel [4]. Although these architectures share many similarities, specific details are left unknown to researchers. To fill knowledge gaps and better understand the hardware, researchers and engineers rely on simulation tools to identify possible improvements.

Several GPU simulation tools exist to help software developers find inefficiencies and suggest ways to improve their code [5, 6, 7]. To evaluate hardware changes, users require more detailed designs to simulate [8, 9]. Abstract simulators struggle to provide accurate power and area estimations. Synthesizable designs give more accurate power and area results, but require more detail to model the same architectures. Current designs containing this level of detail deviate from the

models they try to replicate or are not readily available. Improper use of these designs can lead to incorrect conclusions [10]. There is a need for more accurate designs which can be simulated to give further insight into the GPU design space.

1.2 Main Contributions

In this paper we present a new design for exploring the GPU design space. We present a highly configurable and scalable general purpose GPU (GPGPU) modeled after the Nvidia Tesla architecture. This model can be synthesized and used in cycle accurate simulation to estimate power, performance, and area trade-offs using commercial software. The model supports a subset of parallel thread execution (PTX) instructions to support a wide range of applications.

We demonstrate how our design can be used to determine the effectiveness of hardware changes. A set of configured parameters is given to provide an implementation of the Tesla architecture. Using our tool, we provide insight into how many streaming processors may be appropriate for a single multiprocessor. We give a detailed analysis on how the number of streaming processors can impact performance and reduce the amount of power required per computation. There are many other configurations left unexplored which may provide further insight into positive hardware changes.

Other papers, which will be discussed in Chapter 3, have presented similar designs modeling the Tesla architecture. Our design differs by offering increased configuration support for easier exploration. Some previous approaches have made simplifications which deviate from the design they attempt to replicate. We try to model the Tesla architecture as accurately as possible with the information that is available. Our design is fully synthesizable, allowing for accurate power and area analysis. The source code for our model is available for anyone to use or expand upon [11].

1.3 Overview

Since our model is based off of an existing architecture, we will begin by describing the Tesla architecture in Chapter 2. Chapter 3 will present additional information in the field which may

be beneficial to the reader. We will then discuss the details of our design, and how it relates to the Tesla architecture in Chapter 4. Lastly, in Chapter 5 we go into the results of our simulation, and demonstrate how our design can be used to properly explore the GPU design space. There we present a Tesla configuration and hardware exploration before closing with a conclusion and possible future work.

CHAPTER 2. BACKGROUND

In this chapter we discuss important background information which may be beneficial for readers unfamiliar with GPU architecture or architecture design. A brief introduction about GPUs and the CUDA programming model is given. We then explain some details about the Tesla architecture to serve as a comparison for our own implementation.

2.1 GPUs and CUDA

GPUs have been quickly growing in popularity since the introduction of general purpose applications [12]. The highly parallel architecture is well suited for problems which can be broken into several independent pieces. GPUs have grown from focusing on rendering objects to a screen, to solving complex differential equations and implementing artificial intelligence [13]. Since the release of CUDA, Nvidia GPUs have taken advantage of a SIMT execute model. Threads executing the same functions run in parallel to reduce the total execution time. SIMT models benefit from sharing an instruction cache and decode unit. Additionally, SIMT models remove some responsibilities from the programmer such as scheduling and divergence [1].

2.1.1 CUDA Programming Model

The CUDA programming model partitions GPU programs, called kernels, into grids and blocks [4]. These grids and blocks can be one, two, or three dimensional. A Grid contains an array of blocks which are distributed to streaming multiprocessors to execute threads within a block. Blocks are partitioned further by the hardware into warps which contain thirty-two threads. Programmers specify grid and block dimensions, and can influence how warps are partitioned. Threads belonging to a warp are grouped together, and execute in parallel with one another. Memory access latency caused by cache misses are hidden by zero overhead swaps to other warps [14].

The Tesla architecture, as with all Nvidia GPU architectures released after Tesla, support the CUDA programming model. Newer architectures continue to use CUDA as the programming model scales well with decreasing transistor sizes [15]. In a SIMT execution model, all threads in a warp execute the same instruction with that threads particular data. Thread indices, unique to the thread in a block are used along with the block index to access specific data for that thread. The SIMT model is perfect for GPUs, as only one instruction cache and decode unit is required for parallel execution on a streaming multiprocessor. This shared hardware reduces overhead and allows more processors to fit onto a single die.

2.2 Nvidia Tesla

This paper presents a GPGPU design to evaluate the effects of hardware changes on PPA. Since our implementation is modeled after the Tesla architecture, it is important to know some details about the hardware. In this next section we discuss these details, and provide some background information in GPU related hardware problems.

2.2.1 Basic Architecture

Figure 2.1 shows one of eight texture processing clusters (TPCs) on a Tesla GPU [1]. Blocks are distributed to the streaming multiprocessors (SMs) on the TPC units. These blocks are further split into warps. Each SM contains a scheduling unit which issues warps to the pipeline for execution. The Tesla architecture contains eight streaming processors (SPs) per multiprocessor. These processors share two special function units (SFUs). Each multiprocessor contains a shared memory for fast memory accesses among threads in a block.

The overall pipeline to execute threads in a GPU is similar to a simple CPU. A shallow pipeline depth along with a simpler instruction set architecture (ISA) allow for less complicated and smaller hardware. Several streaming processors exist in parallel on a streaming multiprocessor in order to execute threads belonging to the same warp in parallel. With a warp size of thirty-two threads, the warps are split into four execution rows which execute one after the other. Since only one warp

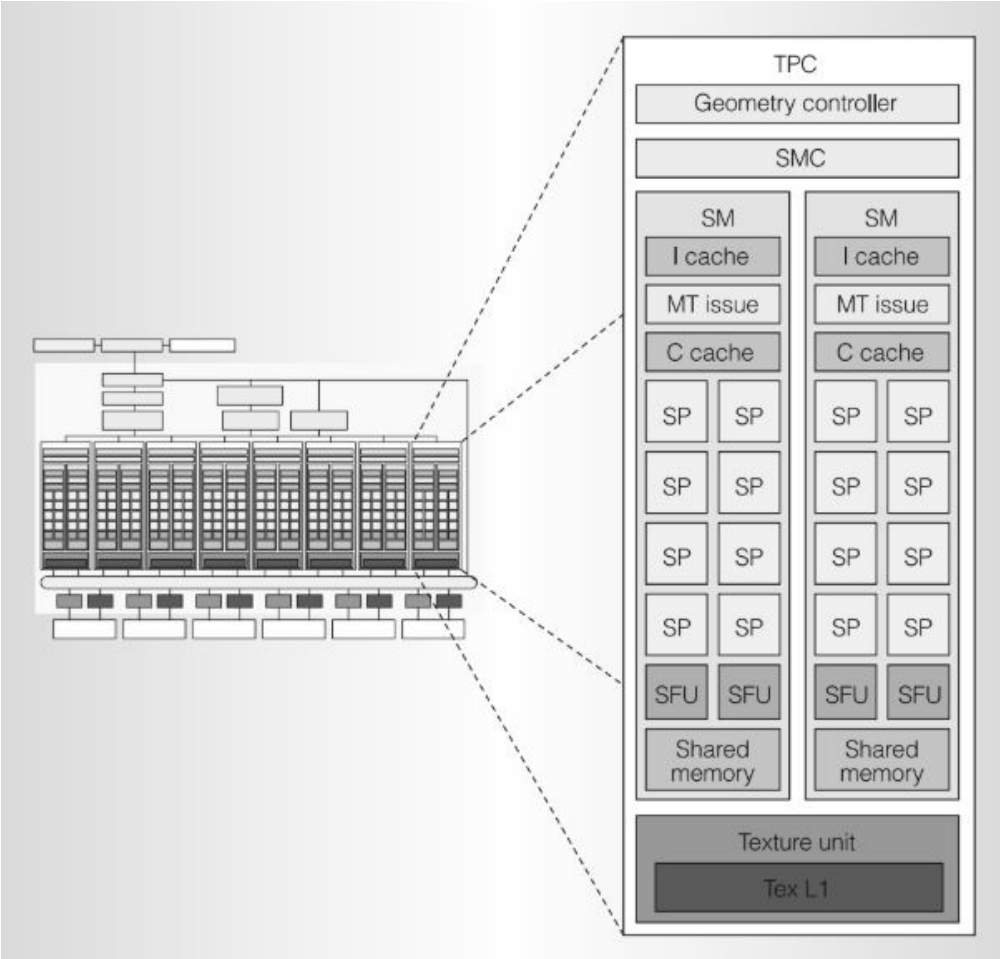

Figure 2.1 Nvidia Tesla Texture Processing Cluster from [1]

is provided at most every four pipeline cycles, the Tesla warp scheduler uses a clock at one fourth the frequency to reduce power consumption. More recent architectures contain enough streaming processors to execute all threads in a warp simultaneously, removing the need for execution rows.

Unlike CPUs, register files in a GPU can exceed the size of the L1 caches. GPU register files need to be large to support zero overhead warp switching. Each thread reserves its own registers in the register file. Newer architectures can support as many as 2048 active threads in a single multiprocessor. To reduce the latency of accessing these register files, the register file is partitioned to processor specific groups. Base register addresses are tracked by each warp to provide the indexing into the register file for the executing threads. This index is the same for all threads in a

warp execution row, as each thread indexes a separate partition of the register file. The operands of the register file are then passed to the respective streaming processors to process the data before writing results to memory or back to the register file.

2.2.2 GPU Related Problems

Two major problems which GPUs face in comparison to single core CPUs are thread divergence and non-coalesced memory accesses. Software developers must be aware of both of these issues to maximize the performance of their GPU applications. Multicore CPUs use a multiple instruction multiple thread or MIMT execution model. Thread divergence is a problem unique to the SIMT execution model. Additionally, memory access contention on a multicore CPU can often be solved with additional memory banks. These memory banks scale poorly with increasing parallel cores, so alternative solutions are implemented on GPUs.

In a SIMT execution model, each thread currently executing, is issued the same instruction. Thread divergence occurs when these threads branch to separate execution paths. Each path can no longer be executed at the same time, as they require different instructions to be read. Supporting additional read ports to instruction memory is expensive, and results in fewer processors per die. Instead, during thread divergence, only one path is executed at a time. When the current path reaches the designated convergence point, the other path executes. After both have reached the convergence point, the paths merge, returning to normal execution. Threads not currently on the executing path are masked, resulting in no data changes. This causes a reduction in performance as not all processors are fully utilized. Since the pascal architecture release in 2016, threads in the same warp are capable of executing separate paths simultaneously, reducing the performance loss.

A common solution for thread divergence is to use a stack to track information of various paths [16]. The stack tracks a convergence PC, the PC of the other path, which threads are on the other path, and a state. The taken path is usually executed first. When the path reaches the convergence point, the not taken path begins to execute. Once both paths reach the convergence point, the threads in both paths merge back together, and the stack entry is popped. The warp stack can

handle any combination of nested divergence. In the worst case, each thread diverges onto its own path. This will require a warp stack with as many entries as there are threads in a warp.

Non-coalesced memory accesses occur when threads executing in parallel issue a memory request which take additional cycles to service [17]. CPUs often use banks for expanding the number of read ports on the cache. GPUs partition the cache into banks, similar to how the data of a cache line is separated into blocks and indexed by a block offset. There are as many bank partitions in the cache as there are streaming processors on a multiprocessor. This allows for all executing threads to perform reads and writes simultaneously as long as they dont request different addresses from the same bank. Same bank accesses are allowed if the data requested is the same. If two or more threads request access to the same bank with different addresses, then a non-coalesced memory access occurs. Since the requests cant be serviced in parallel, non-coalesced memory accesses add to the total time of execution.

To distribute load data and identify non-coalesced memory accesses, hardware is needed to detect and distribute data based on the thread requests. One way to handle a non-coalesced memory access is to use a first in first out (FIFO) queue to request loads and stores to memory. The benefit of this approach is that non-coalesced accesses can be dealt with while other warps execute. Additional hardware is necessary in the condition that both a memory request and another warp in write back need to write results to the register file at the same time.

CHAPTER 3. RELATED WORK

Nvidia introduced the Tesla GPU architecture in 2006 [1]. This architecture unified the vertex and pixel processors to enable more active hardware across a range of applications. A more recent example is the Pascal architecture released in 2016 [3]. Both of these architectures are built around the compute unified device architecture (CUDA) [4]. CUDA is a scalable architecture which takes advantage of shrinking transistor sizes so more devices can be fitted onto a single die [15]. The newer architectures share many similarities between their predecessors [18]. These GPUs execute graphics and general purpose applications by launching kernels from a host CPU [14, 12]. Unfortunately, specific hardware details and design decisions are kept secret. This presents a problem to newcomers looking to learn, and researchers wanting to suggest hardware improvements.

In order for researchers to effectively explore the design space, they rely on simulators with various levels of abstraction. By simulating trusted benchmarks [19, 20, 21, 22], researchers can discover ways to improve the hardware. Abstract, but faster, simulators exist [7, 23, 5, 6] which are useful for researching hardware and software improvements. Careful use of these tools are important as misuse can lead to incorrect results [10]. More accurate simulators can be run on FPGAs [24, 25, 26, 27], and operate on many workloads when connected to a host CPU [8, 28]. These implementations can differ from the desired hardware, so the correct device needs to be used. Statistical and visual approaches have also been taken to help identify bottlenecks, and suggest improvements to compute systems or code [17, 29, 30].

GPUSim [7] provides a detailed simulator for executing CUDA applications. This model is highly configurable and closely models modern graphics architecture. In addition to identifying performance bottlenecks in code, the configurability can provide insight for potential hardware improvements. GPUSim contains a ported version to gem5 as well [23]. These models are not synthesizable, and cannot provide detailed power and area analysis. GPUSim can be used for

suggesting possible hardware improvements, but more detailed analysis is required to confirm the value of these improvements.

Flexgrip offers a configurable GPU modeled after the Tesla architecture [9]. This model can be run on a FPGA using a subset of parallel thread execution (PTX) instructions to execute CUDA Kernels. While Flexgrip is synthesizable, it is not open source or available to all researchers. Flexgrip was designed to accelerate GPU applications on a FPGA. The design contains some configurable parameters to support multiple FPGA designs, but not enough to easily explore the entire GPU design space.

Nyami is another synthesizable design which models general purpose GPUs on a FPGA [8]. Nyami contains full compiler support to execute trusted benchmarks. Since the design is modular, it can be used to evaluate hardware changes. Nyami resembles a MIMT execution model to remove the hassle of thread divergence. As a result, the concept of warps does not exist. Since the model differs from modern GPU architecture, users need to be careful when evaluating potential hardware improvements.

Current GPU architectures run many threads in parallel, and swap between threads to hide memory latency [31, 32, 33]. Topics such as thread divergence [34] and warp scheduling [16] are examples of problems now solved by hardware. Various implementations have been compared to explore the strengths and weaknesses between hardware architectures [35, 36]. Software applications are an important consideration when designing hardware. The CPU and GPU need an interconnect for communicating workloads [37, 38, 39]. Data movement from CPU to GPU is a growing concern as GPU applications may execute faster, but the time it takes to move the data can result in more execution time overall [40]. Additionally, the hardware must support various programming models and applications [41, 42].

Without careful consideration, results from existing designs can lead to incorrect conclusions. To reduce research time and provide more accurate results, a new design is beneficial. Nvidias most recent architectures share many similarities with the Tesla architecture. A synthesizable design modeling the Tesla architecture would allow users to better understand current GPU hardware.

To make this design efficient for research, many desirable parameters should be easily configurable. We propose an implementation of this design to serve as an effective educational and research tool for exploring the GPU design space.

CHAPTER 4. DESIGN DETAILS

This chapter covers the design details and decisions for our configurable GPGPU. We first present a top level model of our design, and how it compares to the Tesla architecture. The details of the pipeline are discussed next. Then we describe how we solved GPU specific problems such as thread divergence and non-coalesced memory accesses. We end discussion of the design by describing the parameters which can be configured, and how to properly simulate the model.

4.1 Design Comparison

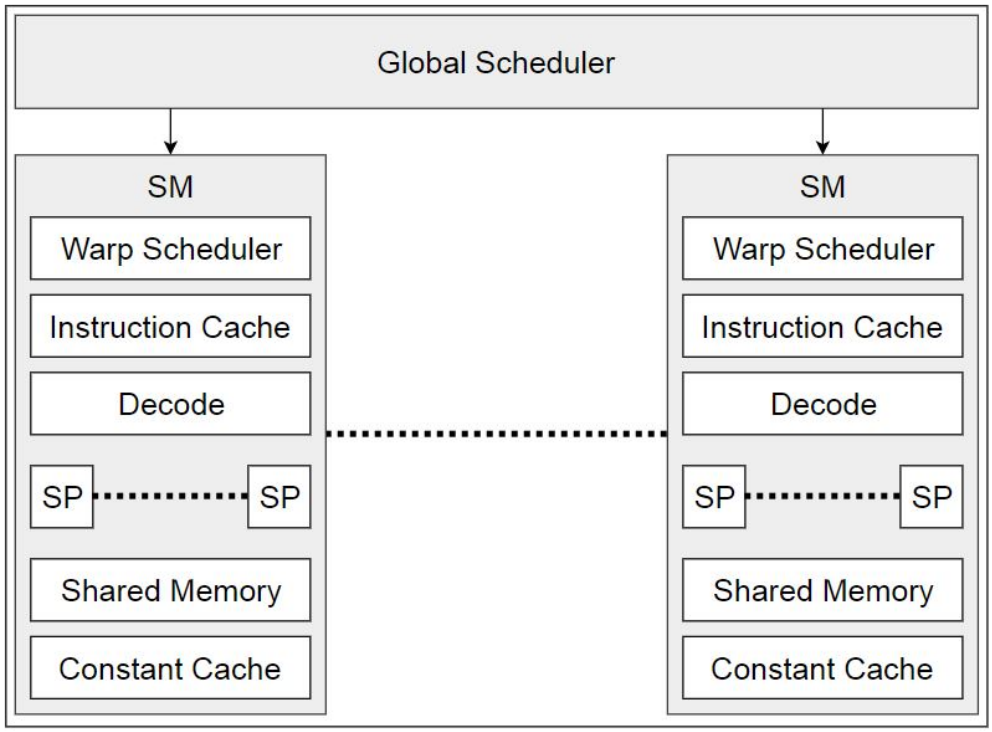

Figure 4.1 Configurable GPGPU Model

Figure 4.1 shows the top level model of our configurable GPGPU. The most notable differences between our design and the Tesla model is the removal of the texture processing cluster, special functional units, and the cache hierarchy. We wanted to provide a base implementation to improve upon, and therefore we decided to remove these features. With the removal of the cache hierarchy, threads will not miss on the shared memory cache. These features may be a good starting point for future work.

Several streaming multiprocessors exist in parallel. Each of the multiprocessors contain a single warp scheduler, instruction, constant and shared memory cache, as well as a single decode unit. Streaming processors exist on multiprocessors to execute threads in a warp. The global scheduler serves as the interconnect between the CPU and GPU, and is responsible for distributing blocks to the multiprocessors.

We chose to implement a global scheduler as an easy way to distribute blocks to the streaming multiprocessors. This relieves the user from the burden of having to do it manually. Instead the user provides information about the kernel, the data to process, and the rest is taken care of automatically. When blocks are sent to the multiprocessors to initialize, the warps are created automatically as well. We do not recommend including the global scheduler during synthesis. Instead we recommend only to synthesize one multiprocessor to reduce total synthesis time.

4.2 Pipeline

The pipeline was broken into four major stages, five including scheduling. These stages can be seen in Figure 4.2. Blocks and warps are first created in the warp unit. This unit then passes the warps to the rest of the pipeline for execution. All threads of a warp currently in the pipeline execute the same instruction on their specific register values. The results are then written to memory or back to the register file for further processing.

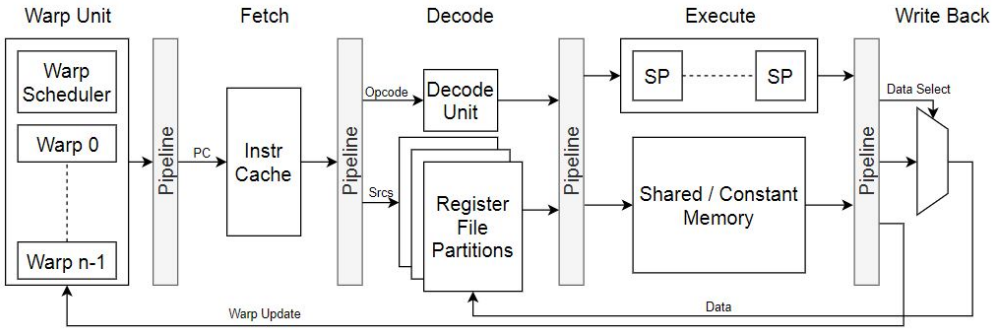

Figure 4.2 Configurable GPGPU Pipeline for a streaming multiprocessor

4.2.1 Warp Unit

The warp unit contains all the information about the blocks and warps currently active on the multiprocessor. Warps track information about the current PC, thread mask, base register address, warp state, and warp ID. A warp scheduler selects which warp should enter the rest of the pipeline. The selected warp will then pass its information to the fetch stage.

Warps are broken into rows of execution depending on the number of threads per warp and the number of processors available on a multiprocessor. The Tesla architecture requires four rows of execution as there are thirty-two threads in a warp, but only eight streaming processors. The warp scheduler only needs to provide a new warp when all rows of execution have completed for the previous warp. For configurations where the warp size matches the number of streaming processors, the concept of warp rows is removed.

We chose to implement a round robin scheduling algorithm for the warp scheduler. This is different than the Tesla architecture which prefers a prioritized greedy algorithm which only swaps warps on stalls or if higher priority warps are ready. The benefits of a round robin scheduler is the ability to keep the pipeline busy, while completely removing the need for forwarding logic. Without forwarding logic, the same threads of a warp cannot exist in multiple locations of the pipeline simultaneously.

4.2.2 Fetch

Since the same threads cannot occupy different stages of the pipeline, a shorter pipeline consisting of four main stages was chosen. Fetch is the first stage encountered after the warp is selected by the scheduler. This stage simply fetches the instruction pointed at by the PC of the selected warp. Additionally, if multiple rows of threads exist in a warp, an offset may be added to the warps base register address to properly index into the register file.

4.2.3 Decode

Decode is the next stage of the pipeline. In the decode stage, the fetched instruction is decoded to produce the required control signals. Since the control signals are not necessary for retrieving source operands from the register file, the operands are also gathered in the decode stage. The register file was partitioned for each streaming processor that exists on a multiprocessor. The splitting of the register file allows for each thread currently executing to retrieve or store operands in parallel.

Each thread belonging to the same row of a warp, index the same address in their own partition of the register file. A base register address is provided by the warp scheduler and offset by the source and destination bits of the instruction. The base register address allows for each thread to reserve a variable amount of registers, the same as the Tesla architecture. The total number of registers a thread can reserve is limited by the source and destination widths of the instruction. This value can be configured to provide for more registers if needed.

4.2.4 Execute

After decode, the operands are sent to the execution stage. Here each of the streaming processors are instantiated, each receiving their respective operands. Supported instructions can be found in Table 4.1. These instructions were chosen from a subset of PTX compute capability 2.0 instructions [4]. Currently, only integer operations are supported. Some instructions were added or modified to better implement a general purpose application.

In addition to streaming processor specific hardware, some hardware in the execution stage is shared among all processors. This hardware includes the L1 shared memory and constant caches. A branch and load store evaluation unit is also provided for determining if there is thread divergence or non-coalesced memory accesses. These issues will be addressed in Section 4.3.

Table 4.1 Supported Instructions

Instruction	**Representation**	**Description**
NOP	000000-xxxxx-xxxxx-xxxxx-xxxxx-000000	No operation
MAD	000000-ddddd-11111-22222-33333-000001	Multiply add
MUL	000000-ddddd-xxxxx-22222-33333-000010	Multiply
ADD	000000-ddddd-11111-22222-xxxxx-000011	Add
SUB	000000-ddddd-11111-22222-xxxxx-000100	Subtract
SLT	000000-ddddd-11111-22222-xxxxx-000101	Set if less than
SGT	000000-ddddd-11111-22222-xxxxx-000110	Set if greater than
SHR	000000-ddddd-11111-22222-xxxxx-000111	Shift register right
SHL	000000-ddddd-11111-22222-xxxxx-001000	Shift register left
NOT	000000-ddddd-xxxxx-22222-xxxxx-001001	Logical NOT
AND	000000-ddddd-11111-22222-xxxxx-001010	Logical AND
OR	000000-ddddd-11111-22222-xxxxx-001011	Logical OR
XOR	000000-ddddd-11111-22222-xxxxx-001100	Logical XOR
MV	000001-ddddd-11111-xxxxx-xxxxx-xxxxxx	Move register to register
MVI	000010-ddddd-xxxxx-iiiii-iiiii-iiiiii	move immediate to register
LD	000011-ddddd-11111-xxxxx-xxxxx-xxxxxx	Load from shared memory
LDC	000100-ddddd-11111-xxxxx-xxxxx-xxxxxx	Load from constant memory
LDS	000101-ddddd-11111-xxxxx-xxxxx-xxxxxx	Load from special register
BEQ	000110-xxxxx-11111-llllll-llllll-llllll	Branch equal to zero
BNE	000111-xxxxx-11111-llllll-llllll-llllll	Branch not equal to zero
ST	001000-xxxxx-11111-22222-xxxxx-xxxxxx	Store to shared memory
SSY	111101-xxxxx-xxxxx-llllll-llllll-llllll	Set synchronous point
BAR	111110-xxxxx-xxxxx-xxxxx-xxxxx-xxxxxx	Barrier synchronization
EXIT	111111-xxxxx-xxxxx-xxxxx-xxxxx-xxxxxx	Exit warp

It is important to note that memory accesses to L1 or the constant cache occur in the execution stage as well. To prevent latency issues, address offsets must be computed in separate cycles using the add instruction. This decision was made to reduce the number of pipeline stages from five to four. The stage reduction reduces the amount of wasted cycles if no other warps are ready to enter

the pipeline. This allows the same warp currently executing to re-enter the pipeline in fewer cycles. With four pipeline stages, two active warps are enough to fully utilize the pipeline hardware.

4.2.5 Write Back

The final stage of the pipeline is the write back stage. In this stage, the control signals choose which data to store into the register file. If a thread is masked by the warp mask then no value is written back for that thread. Additionally, the write back stage signals the warp unit to update. The warp unit updates the warp that was in the pipeline with the next PC to execute, changes to the thread mask, and the state of the warp.

For configurations where the warp executes in rows, the warp executing is only updated when the last row is in the write back stage. This was done to prevent warp rows from taking separate paths, say if one branched but the other did not. In this case we would not want the first row to re-enter the pipeline as it would be executing the incorrect instructions. Furthermore we did not want the rows to track their own PC as the Tesla architecture did not support this feature.

4.3 Thread Divergence and Memory Accesses

This next section is about the problems faced in GPUs which are not common to CPUs. These problems include warp divergence and non-coalesced memory accesses. The hardware of the Tesla architecture was taken into consideration in the solution to each of these problems. Solutions to these problems provided by other papers were also taken into consideration [8, 9].

4.3.1 Thread Divergence

Thread divergence occurs when threads belonging to the same warp take a different path of execution. A branch is a common example where one thread may take the branch, but another thread may not. Thread divergence is an issue for SIMT architectures as multiple threads execute the same instruction in parallel. If the paths of the threads diverge, then multiple instructions are required for the threads to execute simultaneously. The cost to provide these instructions can

quickly become large. A simple solution used by the Tesla architecture is to execute the diverging paths one at a time.

A set synchronous point instruction (SSY) is used to mark a convergence point where the diverging threads will merge back to the same path. Whenever a potentially divergent instruction is encountered, a SSY instruction is required prior to the divergent instruction to setup a convergence point. A warp stack is used to keep track of diverging thread information. Each warp contains a warp stack which tracks the PC set by the SSY instruction, a PC for the threads on the path not currently executing, a thread mask to mark threads on the other path, and a state. The state determines which path is being executed.

When a branch causes threads to diverge, an entry is pushed onto the warp stack. The threads on the taken path execute first. When these threads reach the convergence point marked by the SSY PC, the warp switches to execute the not taken path. The current thread mask will be swapped with the thread mask on the warp stack to enable and disable the appropriate threads. When the not taken path reaches the convergence point, both paths merge to execute in parallel again. This is done by ORing the current thread mask and the thread mask on the warp stack. When merging, the top entry of the warp stack is popped.

4.3.2 Non-Coalesced Memory Accesses

Another common problem with GPUs is non-coalesced memory accesses. In order for all threads to load or store data to memory in parallel, the shared memory cache is split into banks. One bank exists for each streaming processor. These banks split up the cache in the same way that a block offset splits entries in a cache line. The least significant bits of the address determine the bank to access. Contention for one of the banks can occur when two or more threads need to access the same bank on the same cycle. If the threads accessing the same bank are not using the same address, then a non-coalesced memory access occurs. As a result, it will take multiple cycles for all threads to perform the desired request on the memory.

A common solution to this problem is to use a FIFO queue to stagger the memory requests. This solution causes more issues as a load request may finish in the same cycle that another thread is trying to write back to the register file. The result is new contention for storing write back data or the memory loaded value into the register file. Again this could be solved by introducing another FIFO queue to stagger writes to the register file.

An alternative solution was chosen for the design of our GPGPU. We decided to utilize the warp stack to diverge threads during non-coalesced memory accesses. This would cause the threads that were unable to access the cache on the previous attempt, to do so next time the warp enters the pipeline. Once all threads have accessed the cache, then normal execution can resume.

The downside to this approach is that the shared memory cache may be idle when it could be handling a non-coalesced access. This approach however greatly reduces the amount of additional logic required to handle non-coalesced memory accesses. Additionally the warp stack which already exists offers a zero cost solution beyond detecting the access issue.

4.4 Configurability

This section outlines what parameters can be configured to implement various GPGPU designs. Table 4.2 shows all of the parameters which can be configured. By changing the associated value, the hardware will update automatically to match the desired configuration.

For more processing power the number of multiprocessors and streamining processors can be modified with the NUM_MP and SP_PER_MP parameters. Grid and block dimensions are separated into a DIM and DIM_WIDTH parameter. This is because the x, y, and z dimensions are packed into a single register value. The DIM_WIDTH parameter specifies the number of bits used to represent an individual dimension. By default these dimensions are large, but can be increased if needed. The number of active blocks and warps supported by a multiprocessor can also be changed.

All of the caches can be configured. The caches, as well as the register file partitions, were implemented using block rams. Only the instruction and register file data widths can be modified, as the constant and shared memory cache entries should fit into the register file. Information about

the instruction can also be easily modified to support new instructions without needing to worry about breaking already existing hardware. The source and destination widths are beneficial for allowing more or less registers to be allocated per thread.

Table 4.2 Configurable Parameters

Parameter	**Description**
SYNTHESIS	One when synthesizing, removes simulation specific hardware
GRID_DIM	Combined grid dimension width for x, y, and z dimensions
BLOCK_DIM	Combined block dimension width for x, y, and z dimensions
GRID_DIM_WIDTH	Individual grid dimension width
BLOCK_DIM_WIDTH	Individual block dimension width
NUM_PARAMS	Number of parameter registers
NUM_MPS	Number of streaming multiprocessors
SP_PER_MP	Number of streaming processors per multiprocessor
NUM_BLOCKS	Maximum number of active blocks per multiprocessor
NUM_WARPS	Maximum number of active warps per multiprocessor
WARP_WIDTH	Number of threads in a warp
I_DATA_WIDTH	Data width of instruction cache
I_ADDR_WIDTH	Number of bits in instruction address
R_DATA_WIDTH	Data width of register file
R_ADDR_WIDTH	Number of bits in register file address
C_ADDR_WIDTH	Number of bits in constant cache address
L_ADDR_WIDTH	Number of bits in shared memory cache address
CONTROL_WIDTH	Number of control signals
OP_WIDTH	Number of bits in opcode
FUNC_WIDTH	Number of bits in function Code
DEST_WIDTH	Number of bits in destination
SRC_WIDTH	Number of bits in source
IMM_WIDTH	Number of bits in immediate

Some limitations exist with these parameters. Most should be specified in powers of two for efficient hardware utilization. The L_ADDR_WIDTH specifies the total address width of the shared memory, not the address width of each bank. The R_ADDR_WIDTH on the other hand specifies the address width for each partition of the register file. The total space is the number of bits in the address multiplied by the number of streaming processors per multiprocessor.

4.5 Simulation

In order to simulate the design the user must initialize information about the kernel. Information about the kernel is gathered from the global scheduler by reading a kernel cache. Similar to the instruction cache, the kernel cache reads kernel specific instructions to decode information about the kernel. These instructions are listed in Table 4.3. The kernel cache should be initialized with the kernel information using a memory file. This can be done by modifying a mem file type with cache entries defined as hexadecimal values separated by white space.

Table 4.3 Kernel Instructions

Instruction	**Representation**	**Description**
GRID	0000-dd-zzzzz-zzzzz-yyyyy-yyyyy-xxxxx-xxxxx	Grid dimensions
BLOCK	0001-dd-zzzzz-zzzzz-yyyyy-yyyyy-xxxxx-xxxxx	Block dimensions
PARAM	0010-dd-ddddd-ddddd-ddddd-ddddd-ddddd-ddddd	Parameters
INSTR	0011-ii-iiiii-iiiii-iiiii-iiiii-iiiii-iiiii	Single instruction
CONST	0100-dd-ddddd-ddddd-ddddd-ddddd-ddddd-ddddd	Single constant
DATA	0101-dd-ddddd-ddddd-ddddd-ddddd-ddddd-ddddd	Single piece of data
WARP	0110-dd-ddddd-ddddd-ddddd-ddddd-ddddd-ddddd	Warps per block
REG	0111-dd-ddddd-ddddd-ddddd-ddddd-ddddd-ddddd	Registers per thread
START	1111-xx-xxxxx-xxxxx-xxxxx-xxxxx-xxxxx-xxxxx	Start distributing

The GRID and BLOCK instructions specify the grid and block dimensions of the kernel. The PARAM instruction is used to setup pointers to memory locations and is stored in special registers contained by the blocks. INSTR, CONST, and DATA instructions specify data to store to the respective caches. Only one thirty-two bit value can be stored at a time, and subsequent values should be given by multiple instructions. WARP specifies the number of warps per block. This instruction makes it easier to recognize how many warps need to be allocated for a block. REG specifies the number of registers reserved in the register file by each thread. START will tell the global scheduler to begin distributing blocks to multiprocessors ready to receive them.

When distributing blocks to the multiprocessors, warps begin execution as soon as they are initialized. The warp unit can initialize new warps belonging to the same block while other warps execute. Warps are initialized with the correct base register address, PC, thread mask, thread IDs,

warp, and block IDs. The block initialized holds information about the parameters, grid and block dimensions, and tracks the state of barrier synchronization.

CHAPTER 5. Results

In this chapter we present our synthesis results for various configurations of the GPGPU. We first present a configuration that resembles the original Tesla architecture. Next we show how re-configuring the number of streaming processors affects power, performance, and area. After analyzing the results we wrap up with a conclusion and possible future work.

5.1 Tesla Configuration

The first configuration of our GPGPU closely resembles that of the Tesla architecture. The most important parameters and their configurations are shown in Table 5.1. The number of processors per multiprocessor was set to eight to match the Tesla architecture. We chose sixteen multiprocessors as the Tesla architecture contains eight TPC units, each with two multiprocessors. The maximum number of active blocks and warps on a multiprocessor was set to four and sixteen respectively. These values were chosen to allow for a maximum of 512 active threads per multiprocessor. The address widths of the caches and register files were reasonably chosen to support 512 active threads.

Table 5.1 Tesla Configuration

Parameter	**Value**
NUM_MPS	16
SP_PER_MP	8
NUM_BLOCKS	4
NUM_WARPS	16
WARP_WIDTH	32
I_ADDR_WIDTH	8
R_ADDR_WIDTH	9
C_ADDR_WIDTH	6
L_ADDR_WIDTH	10

The Tesla configuration was synthesized using Cadence Encounter with a 65nm process. We chose to use a clock frequency of 300 MHz for this experiment, but the design was valid for clock

frequency of up to 700 MHz. Table 5.2 shows some synthesis results for this design. The slack was a positive value meaning a faster clock could have been used. A total power consumption of 11.55W was observed. The lower power consumption is due to the lack of a cache hierarchy, special function units, and other specialized hardware we chose not to implement. The design consumed a total of 4.912 million cells from the 65nm library. These cells resulted in a total area consumption of 25.41 mm^2.

Table 5.2 Tesla Synthesis Results for 300 MHz Clock

Slack	**Power**	**Cells**	**Area**
2 ps	11.55 W	4.912 M	25.41 mm^2

These results appear to be reasonable given the limitations of the design. The critical path was observed to be through the multiply add unit as expected. Each module was synthesized individually prior to the synthesis of the Tesla configuration, and no major issues were observed. The next section demonstrates what can be learned from comparing the synthesis results of various configurations.

5.2 Scaling Processors

One of the questions we wanted to explore was at what point it becomes beneficial to scale the number of multiprocessors instead of the number of streaming processors. We ran synthesis on configurations of four, eight, sixteen, thirty-two, and sixty-four streaming processors per multiprocessor. Since register file partitions were instantiated for each processor, we set the register address width to maintain the same total number of registers. The synthesis results can be observed in Figure 5.1.

Each simulation had a positive or zero slack, with the values ranging from zero to seven picoseconds. The results were recorded for individual multiprocessors rather than all sixteen. We measured power, area, and the total number of gates. Gates here is the number of library cells instantiated. The total number of transistors will be larger. We again chose a clock frequency of

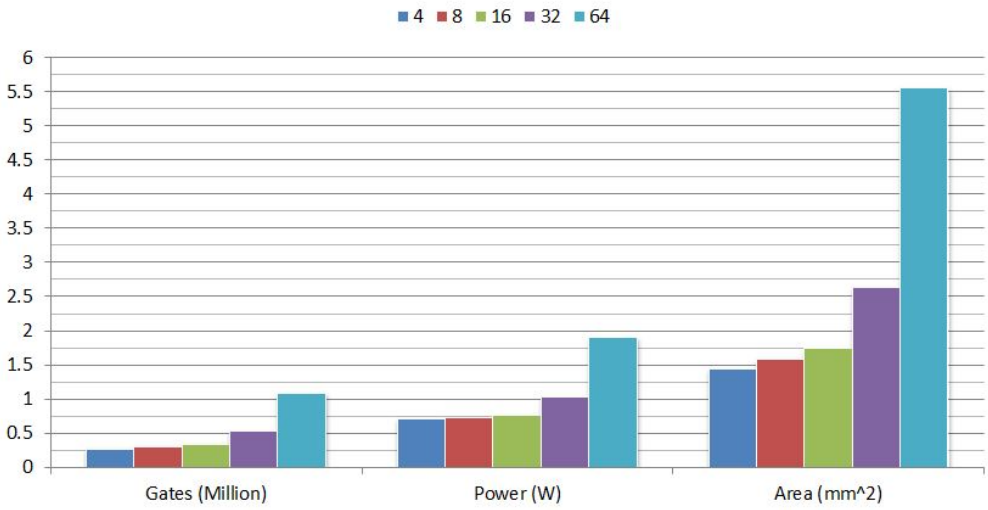

Figure 5.1 Synthesis results for variable number of streaming processors. Clock set to 300 MHz.

300 MHz to ensure fair synthesis results. We did not want the larger designs to close in on the maximum possible frequency as additional buffer logic could skew power and area results.

When the number of streaming processors is low, the overhead from the rest of the hardware makes up the majority of the streaming multiprocessor. As a result there is little power and area increase as the number of processors increase initially. From sixteen to thirty-two this amount has a more significant increase. The total area consumption for a single multiprocessor more than doubles as the number of processors increase from thirty-two to sixty-four. Even though the number of processors doubles, increasing the area beyond double means less than half the multiprocessors can fit in the same area as the previous die. This results in a net loss in performance since there will be fewer total streaming processors.

These results did not come as a surprise. When designing the hardware for detecting non-coalesced memory accesses, we noticed the amount of hardware increased quadratically with the number of processors. This is caused by needing to compare the requested shared memory addresses with each other, as well as requiring more banks. This theory was further supported when the critical path through the multiprocessor changed from the multiply add unit to the detection

hardware and shared memory cache. We also note that there was some additional overhead required to increase the warp size to sixty-four to support the sixty-four processors.

Because of this insight, we were able to determine that the optimal number of streaming processors per multiprocessor was thirty-two. After adjusting for the increase in area, we were able to determine the increase in the number of processors per unit area and the power reduction per processor. These results can be seen in Table 5.3. We assume that adding additional multiprocessors scales linearly. The reported values are in reference to the previous number of processors.

Table 5.3 Adjusted increase of number of processors and power reduction per processor for doubling processors on a multiprocessor

SP Increase	Increase in Processors Per Unit Area	Reduction in Total Power Per Processor
4 to 8	83%	49%
8 to 16	81%	47%
16 to 32	32%	32%
32 to 64	-5%	8%

Each doubling of the number of streaming processors shows an improvement in the number of processors per unit of area until doubling to sixty-four processors per multiprocessor. This is caused by the area consumption more than doubling. All doublings saw a reduction in power per processor in an equivalent amount of area. This is due to a larger ratio of power being spent on processor execution rather than the overhead from the rest of the multiprocessors. At thirty-two processors per multiprocessor, we recorded an improvement in the number of processors of 32% and a reduction in power per processor of 32%. Adding additional processors leads to a performance loss. For this reason, thirty-two processors per multiprocessors appears to be the best option.

5.3 Conclusion

Power, performance, and area are important metrics to consider in the design of hardware. GPUs are massive and complex architectures which need detailed analysis to accurately determine hardware improvements. Our configurable GPGPU is based off an early CUDA architecture, but can be scaled to newer architectures as well. The design can easily be configured to realize various

hardware changes. Designs can be synthesized using commercial software to provide PPA results. The full source code is available for anyone to expand or try out their own designs [11].

Using this new configurable GPGPU, we were able to gain insight into GPU hardware. We provided a base configuration to model the Tesla architecture. Additionally we were able to observe how changing the number of processors per multiprocessor impacted the power, performance, and area of the GPGPU. This insight allowed us to make conclusions about the optimal number of streaming processors per streaming multiprocessors. Additionally, the synthesis results gave us a better understanding of what gains we can expect by scaling the number of processors.

5.4 Future Work

We have only provided a base model to explore the GPU design space. There are several areas which can be improved upon to more accurately represent current hardware. The major pieces missing are rendering specific hardware support and a cache hierarchy. Expanding the ISA to include special function units, floating point operations, or a more optimized execution pipeline would be a great contribution to the current design.

Only a shared memory cache exists on the current architecture. Multiple levels of cache, texture memory, and DRAM may be beneficial. While the current model can give insight into many applications, a proper cache hierarchy may be necessary for others. This would allow the model to properly simulate cache misses and lead to more accurate performance results. Additionally the global scheduler could be changed to initialize the DRAM instead of the caches specific to each multiprocessor. The DRAM could also serve as a location to store a frame buffer.

Our design was created to be easily configurable, but there is still room to simplify simulation setup. To simulate the design, the user has to manually specify kernel details in a memory initialization file. This can be tedious and prone to error. Compiler support would be a useful addition to automatically generating a binary which can be run on our model. This would allow for trusted benchmarks to be more easily ported and executed on the model.

BIBLIOGRAPHY

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008.

[2] E. Lindholm, M. J. Kilgard, and H. Moreton, "A user-programmable vertex engine," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 149–158, ACM, 2001.

[3] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[4] N. Corporation, "CUDA toolkit documentation v10.1.105," 2019.

[5] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "ATTILA: a cycle-level execution-driven simulator for modern GPU architectures," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 231–241, IEEE, 2006.

[6] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for GPGPU," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 351–360, IEEE, 2010.

[7] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, April 2009.

[8] J. Bush, P. Dexter, T. N. Miller, and A. Carpenter, "Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 173–182, IEEE, 2015.

[9] K. Andryc, M. Merchant, and R. Tessier, "Flexgrip: A soft GPGPU for FPGAs," in *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 230–237, IEEE, 2013.

[10] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "gem5, gpgpusim, mcpat, gpuwattch," your favorite simulator here" considered harmful," 2014.

[11] G. Lies, "A configurable GPGPU for PPA analysis: https://github.com/gjlies/configgpgpu," 2019.

[12] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[13] A. Bleiweiss, "GPU accelerated pathfinding," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 65–74, Eurographics Association, 2008.

[14] D. Luebke and G. Humphreys, "How GPUs work," *Computer*, vol. 40, no. 2, pp. 96–100, 2007.

[15] J. Y. Chen, "GPU technology trends and future requirements," in *2009 IEEE International Electron Devices Meeting (IEDM)*, pp. 1–6, Dec 2009.

[16] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308–317, ACM, 2011.

[17] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 152–163, ACM, 2009.

[18] T. Akenine-Moller, E. Haines, and N. Hoffman, *Real-time rendering*. AK Peters/CRC Press, 2018.

[19] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 335–344, IEEE, 2012.

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

[21] T. D. Han and T. S. Abdelrahman, "hi CUDA: a high-level directive-based language for GPU programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 52–61, ACM, 2009.

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

[23] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "Gem5-GPU: A heterogeneous CPU-GPU simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[24] K. Kim, S. Cho, S. Park, *et al.*, "Implementation of 3D graphics accelerator using full pipeline scheme on FPGA," in *2008 International SoC Design Conference*, vol. 2, pp. II–97, IEEE, 2008.

[25] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "SGRT: A mobile GPU architecture for real-time ray tracing," in *Proceedings of the 5th high-performance graphics conference*, pp. 109–119, ACM, 2013.

[26] L. Wu, L. Huang, and T. Xiong, "Designing a unified architecture graphics processing unit," in *The 7th International Conference on Computer Engineering and Networks*, vol. 299, p. 079, SISSA Medialab, 2017.

[27] W.-Y. Kim, B.-H. Lee, K.-Y. Lee, and J.-C. Kwak, "Design of a fully programmable shader processor for low power mobile devices," in *TENCON 2009-2009 IEEE Region 10 Conference*, pp. 1–5, IEEE, 2009.

[28] J. Bush, M. A. Khasawneh, K. Z. Mahmoud, and T. N. Miller, "NyuziRaster: Optimizing rasterizer performance and energy in the Nyuzi open source GPU," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 204–213, IEEE, 2016.

[29] C. Weaver, K. C. Barr, E. Marsman, D. Ernst, and T. Austin, "Performance analysis using pipeline visualization," in *2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 18–21, IEEE, 2001.

[30] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *2011 IEEE 17th international symposium on high performance computer architecture*, pp. 382–393, IEEE, 2011.

[31] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, pp. 50–59, March 2011.

[32] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, ACM, 2008.

[33] C. Nugteren, G.-J. Van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 37–48, IEEE, 2014.

[34] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 591–602, IEEE, 2013.

[35] Y. Zhang, L. Peng, B. Li, J. Peir, and J. Chen, "Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 205–215, Nov 2011.

[36] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, pp. 149–158, IEEE, 2011.

[37] M. Eldridge, "Designing graphics architectures around scalability and communication," *Unpublished doctoral thesis, Stanford University*, 2001.

[38] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and W. Wong, "Guppy: A GPU-like soft-core processor," in *2012 International Conference on Field-Programmable Technology*, pp. 57–60, Dec 2012.

[39] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-CPU, GPU and memory controller 32nm processor," in *2011 IEEE International Solid-State Circuits Conference*, pp. 264–266, IEEE, 2011.

[40] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 134–144, IEEE, 2011.

[41] H. Kasim, V. March, R. Zhang, and S. See, "Survey on parallel programming model," in *IFIP International Conference on Network and Parallel Computing*, pp. 266–275, Springer, 2008.

[42] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL," *arXiv preprint arXiv:1005.2581*, 2010.