# Power And Hotspot Modeling For Modern Gpus

Md Mainul Hassan
*University of Mississippi*

# POWER AND HOTSPOT MODELING FOR MODERN GPUS

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Md Mainul Hassan

May 2015

## ABSTRACT

As General Purpose GPUs (GPGPU) are increasingly becoming a prominent component of high performance computing platforms, power and thermal dissipation are getting more attention. The trade-offs among performance, power, and heat must be well modeled and evaluated from the early stage of GPU design. This necessitates a tool that allows GPU architects to quickly and accurately evaluate their design. There are a few models for GPU power but most of them estimate power at a higher level than architecture, which are therefore missing hardware reconfigurability. In this thesis, we propose a framework that models power and heat dissipation at the hardware architecture level, which allows for configuring and investigating individual hardware components. Our framework is also capable of visualizing the heat map of the processor over different clock cycles. To the best of our knowledge, this is the first comprehensive framework that integrates and visualizes power consumption and heat dissipation of GPUs.

## ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude and sincere appreciation to my advisor Dr. Byunghyun Jang for unprecedented support, relentless encouragement, and providing me with an outstanding atmosphere for doing research. I have been blessed with his excellent guidance, caring, patience, availability and immense knowledge of computer architecture and GPU computing that helped me at all points of my graduate study and research.

I would like to thank Dr. Wilkins for her continuous support not only in academic problems but also in non academic problems. I also thank Dr. Rhodes for being kind enough to be in my committee. I am fortunate to have such professors whose knowledge in algorithm, data structure, image processing, graphics and architecture helped me to shape up my young scientist mind.

I thank my department chair, Dr. Conrad Cunningham and the rest of our amazing faculties and staffs. Thank you for your support and the wonderful job for the department. I thank all my fellow HEROES at the Heterogeneous Systems Research Lab: David Troendle, Kyoshin Choo, Mengshen Zhao, Tuan Ta, Michael Ginn, Zhaohua Yi, Esra, Orevaoghene Addoh, Stephen Adams for your assistance with my research and stimulating discussions during our meetings. This work is supported by National Science Foundation (NSF) grant CCF-1337138.

Finally, to my wonderful parents and younger sister, thank you for all your advice, sacrifice, encouragement, patience and support throughout my life. I am fortunate and I could not ask for a better family.

TABLE OF CONTENTS

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

GPGPU is getting more widely adopted across different platforms such as mobile devices, desktops, and servers as well as supercomputers as a mean to achieving high performance. As more and more parties are striving to drive GPU performance up, more transistors are put inside a GPU device. At architecture level, they are built upon several billions of transistors which consume more than 200 watt. The direct consequence of their high power consumption is growing heat dissipation and more expensive cooling solutions. The widespread adoption of GPGPU in almost all devices makes analyzing and modeling the power consumption of GPUs a priority.

DVFS (Dynamic Voltage and Frequency Scaling) and specialized circuit techniques have been main strategies in low-power design for several decades. Unfortunately, these techniques alone are not enough; higher level strategies for reducing power consumption are increasingly crucial. As a result, a robust power model is necessary for processor architects to build power efficient computer systems. A robust power model must satisfy three requirements to be useful for computer architecture research. It must be (1) *configurable*, (2) *cycle level*, and (3) *strongly validated* against existing processor architectures using a rigorous methodology [14]. In the past couple of decades, a great deal of effort has been put into developing such robust CPU power models (e,g., Wattch [7] and McPAT [15]), which brought a surge of power-related research in computer architecture community. However, relatively few works are found regarding GPU power modeling and analysis. They are based on statistical estimation and empirical measurement from specific GPU architectures. These models are inadequate for generic design space exploration.

The difficulty of developing a GPU power model is due to insufficient information available to the research community about GPU micro-architecture and implementation. Moreover, GPU micro-architecture is somewhat different from CPU's as GPUs include new components such as local memory and thread scheduler. GPUs also specialize some components such as register files. Even though there exist several cycle-accurate architecture simulators which are currently used by architects to analyze performance and make design decisions, these tools do not provide power and heat consumption data for GPU.

In this thesis, we develop a GPU power model framework for contemporary GPGPU architectures which satisfies all the aforementioned criteria using a cycle accurate architectural simulator called Multi2Sim [25], a low-level power modeling framework called McPAT [15] and a heat estimation tool called HotSpot [23]. Our framework is able to estimate multiple characteristics of a GPU architecture such as chip area, peak dynamic power, as well as simulate the power consumed during the execution of GPGPU workloads. This framework also predicts heat consumption of GPU components as well as heat dissipation over time across space. With this framework, computer architects can evaluate the power and heat characteristics of their design early, and GPGPU programmers gain an effective way to investigate their GPGPU code to optimize power consumption from a software perspective. In summary, this thesis makes the following contribution:

- To the best of our knowledge, this is the first work that uses a cycle accurate architectural simulator, Multi2Sim, for GPU power modeling. We implement necessary modifications to this simulator to generate necessary data needed for power simulation.

- We update a well known CPU power modeling framework, McPAT, to model multiple GPU components (e.g., L1 cache, L2 cache, local memory, register files, functional units etc). Specifically, we model AMD's latest GPU architecture code-named Southern Island.
- With the integration of HotSpot, our model also estimates heat consumption as well as heat dissipation. It helps to identify hot spots in the device.
- We develop a graphical interface that displays power data and graphs for different GPU components at different cycles. This interface also visualizes the heat dissipation over time across different components.

# CHAPTER 2

# BACKGROUND

‘

In this chapter, we provide the technical background on GPU. Section 2.1 discusses GPU and GPU computing and Section 2.2 gives an overview of GPU hardware architecture. We describe previous work on GPU power model in Section 2.3

## 2.1 GPU and GPGPU

GPU (Graphics Processor Unit) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to screen [27]. The principal motivation behind building GPU was to efficiently and quickly perform mathematically- and computationally intensive computer graphics and image processing tasks. GPU's highly parallel structure makes them more effective than general-purpose CPUs for algorithms that process large blocks of data in parallel. Graphics chips have been developed by several manufacturers since the 1970s. But the term GPU was popularized by NVIDIA, who, in 1999, marketed their GeForce 256 as “the world's first GPU.” Their GPU at that time could process 10 million polygons per second and it had over 22 million transistors. GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. In personal computers, GPU can be present on discrete video card, or it can be embedded on the motherboard. Nowadays it is popular that GPU and CPUs are fabricated on a single die sharing a memory subsystem. It is known as heterogeneous architecture.

The primary motivation behind graphic processing units was, and still is, to efficiently render the graphical components desired by the computer application and related software which requires thousands of simultaneous calculations to produce the accurate image frame according to the demands of the software (e.g., graphics pipeline). This highly parallel computing architecture eventually brings forth the idea of using GPU for other non-graphics tasks. This computing paradigm is known as GPGPU computing. GPGPU (General Purpose computing on GPU) refers to using graphics processing units to accelerate non-graphics problems. GPGPU computing offers unprecedented application performance by offloading compute-intensive portions of an application to the GPU. A simple way to understand how GPU accelerates a compute intensive application more than a CPU is to compare how they process tasks. CPU consists of a few cores optimized for sequential serial processing while GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.

Figure 2.1 shows the comparison between CPU cores and GPU cores. It is evident from the figure that with a thousand of cores, GPUs can perform more computations in parallel which in turn gives a performance boost over CPU. GPU not only increases performance, but also improves power efficiency. They are inherently more energy efficient than other ways of computation because they are optimized for throughput and performance per watt and not absolute performance. Even though GPGPU computing provides unmatched benefits in performance and energy consumption, it comes with a grain of salt. To offload compute-intensive tasks to GPU, we need to transfer data to GPU memory. This data copying is done through a slow PCI bus. This can be a performance bottleneck for GPGPU. Moreover, GPU computing cannot be used to improve performance of all irregular applica-

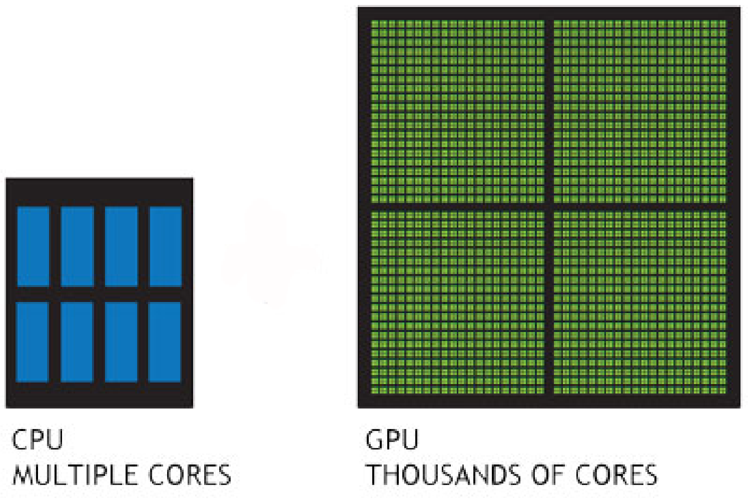


Figure 2.1. CPU cores vs GPU cores.

tions. To benefit from GPU computing an application has to have certain characteristics such as.

- It must have a large computational requirement so that it can hide the memory transfer latency. GPU is built to do a large number of computations in less time thanks to its thousands of cores. But the data required for the computations have to be transferred to GPU memory. This operation takes considerable amount of time. If computations are not much, then GPU cannot hide this memory latency and the total execution time would be slow.

- The algorithm should be parallel or parallelizable. This means there should not be any data dependency between multiple tasks. A simple example of this type of computation is vector addition. In vector addition, we sum up one element of an array with an element at the same index in another array and put the result in a third array at the same index position. There is no data dependency between different addition operations. We can add elements at index position 3 without waiting for the result of the addition operation at index position 2. On the contrary, there are algorithms where a task is dependent on the result of another task. In such cases, a sequential execution is necessary for correctness. These types of algorithms are not data independent and cannot be used in GPGPU. There is ongoing research to accelerate such irregular applications using GPU [8]. Irregular applications are those where the control flow and memory access patterns are data-dependent and statically unpredictable (e.g., binary-search-tree implementation). In a binary-search-tree implementation, the values and the order in which they are processed affect the control flow and memory references. Processing the values in sorted order will generate a tree with only right children whereas the reverse order will generate a tree with only left children, thus exercising a different control flow path. Graph-based applications in particular tend to be irregular and hence are not suitable at all for GPU computing.

To fully utilize the GPU's parallel architecture, we need a different programming model. Open Computing Language (OpenCL) is an industry standard programming model for GPU computing [19]. It is a heterogeneous computing framework for writing portable programs that run on different platforms including CPU, GPU or FPGA etc.

## 2.2 GPU Hardware Architecture

Over the last 15 years, GPUs have continually evolved and are on the cusp of becoming an integral part of the computing landscape. The first designs employed special purpose hardware with little flexibility. Later designs introduced limited programmability through shader programs, and eventually became highly programmable throughput computing devices that still maintained many graphics-specific capabilities. As we use AMD's Southern Island GPU architecture in this thesis, we shall have a brief discussion of its architecture.

Southern Island Architecture is based on AMD's new Graphics Core Next (GCN) architecture. GCN represents a fundamental shift for GPU hardware and is the architecture for future programmable heterogeneous systems. GCN is carefully optimized for power

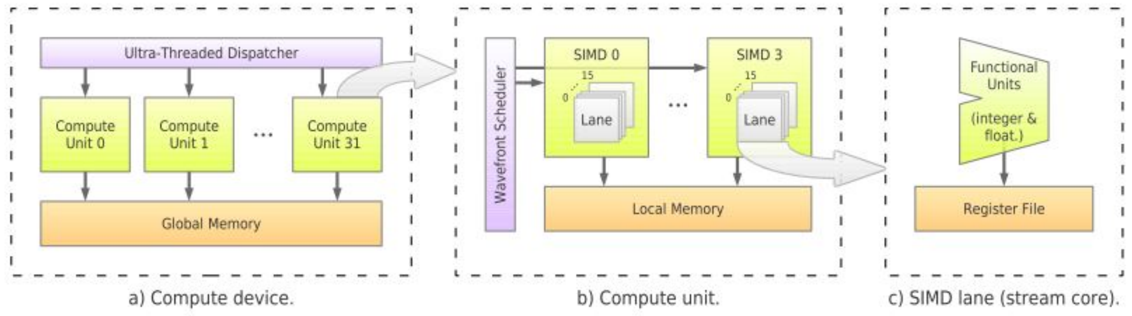


Figure 2.2. A simplified look of Southern Island Architecture.

and area efficiency at the 28nm node and will scale to future process technologies in the coming years. Compute Units (CU) are the basic computational building block of the GCN architecture. These CUs implement an entirely new instruction set that is much simpler for compilers and software developers to use and delivers more consistent performance than previous designs [2]. Previously, AMD GPUs consist of a number of SIMD engines which comprised 16 ALUs. Each ALU could run a bundle of five to six instructions in a VLIW (*Very Long Instruction Word*) format. This architecture is not suitable for general purpose computing. This is because in general purpose computing the underlying data format is not uniform and sometimes unpredictable which makes it difficult to consistently finding four to five independent instructions to execute in parallel in each cycle. GCN introduces vector processing in SIMD units. Each of these units can run a single operation on multiple threads simultaneously.

Figure 2.2 displays a simplified architecture of the Southern Island (SI) GPU architecture. At top level, the device consists of 32 compute units, a thread dispatcher and global memory. The thread dispatcher schedules the work groups to CUs whenever they are available. In OpenCL programming model, work groups are a collection of work items or threads. Each CU runs one or more work groups at a time. 64 work items of the same work group is called a wavefront. All work items in a wavefront execute the same instruction at a time. Global memory is accessible to all threads being executed on all CUs. Each CU has a wavefront scheduler that schedules wavefront from running work groups to SIMD units. Each SIMD unit consists of 16 light-weight cores. Each core includes integer and floating point functional units. Low latency, high bandwidth *local memory* is an on-chip memory that resides in each CU. All work items inside a work group can access and share data through local memory. L1 cache is private to each CU whereas L2 cache is shared across all CUs. Coherency amongst these caches is crucial for GCN performance. The cache coherency protocol shares data through L2 cache, which is significantly faster and more power efficient than using off-chip graphics memory. We shall discuss more on L1 cache and L2 cache structure in Section 3.3

## 2.3 Related Work

Power modeling of CPU has been widely studied and documented. Wattch [7] was the first seminal work on architecture level power model for CPU. An advanced and integrated power, area, and timing modeling framework named McPAT [15] was developed in HP Labs later. McPAT has certain benefits over Wattch. For example, McPAT models not only dynamic power but also static and short-circuit power. McPAT supports multicore or manycore processors. It handles technologies that can no longer be modeled by the linear scaling assumptions used by Wattch. Other than these two architecture level modeling frameworks, there are a few statistical models. Joseph and Martonosi [12] and Isci and Martonosi [11] used performance counters to examine power-relevant events and to present per-unit power estimates, respectively. Wu et al. [29] searches the component unit power of a processor by using a K-means-based method to correct inaccuracy resulting from manual tuning using empirical data.

While the literatures is rich enough for modeling CPU power consumption, power modeling for GPU has received little attention due to the scarcity of documented information for GPU hardware architecture. There are few GPU power models proposed based on statistical estimation and empirical measurement from specific GPU architectures. GPU power modeling works can be divided into two approaches: *statistical* approach and *architecture* approach. Initial works on GPU power model framework was inspired by the statistical modeling methodologies of CPU power. Ma et. al [18] present a scheme to statistically analyze and model the power consumption of a mainstream GPU (NVIDIA GeForce 8800GT) by exploiting the innate coupling among power consumption characteristics, runtime performance, and dynamic workloads. Based on the recorded runtime GPU workload signals, their trained statistical model is capable of robustly and accurately predicting power consumption of the target GPU. Hong and Kim [10] propose an integrated power and performance model for GPU. Their model predicts execution times to calculate dynamic power events. Then they uses the outcome of their model to control the number of running cores. They also model the increases in power consumption that resulted from the increases in temperature. Song et al [24] propose a power model based on performance counter. They use the hardware performance counter and physically measured power data as input to an Artificial Neural Network (ANN) to train the model. This model is then used to predict power consumption. Zhang et al. [9] utilized random forest model for ATI GPUs and analyzed the power consumption along with performance. Nagasaka et al. [20] used the linear regression method by collecting the information about the application from performance counters.

Most of the work on GPU power modeling is based on statistical modeling. The fundamental drawback of these models is that they are not configurable. That serves as the motivation behind our modeling a power estimation framework at the architecture level. Very few works have been done at the architecture level and all of them model power based on NIVIDIA GPU architecture, mostly Fermi architecture. PowerRed [21] is one of the architecture level power estimation framework. It combines both analytical and empirical models. They used Wattch to build various components of GPU. Similar work was done by Li [16]. Leng et. al [14] proposed a robust power model for GPU using McPAT as the power estimation tool and GPGPUSim [6] as performance simulator. This model is configurable and capable of cycle-level calculations. A bottom-up methodology and abstract parameters from the micro architectural components are used as the model's inputs. A rigorous suite of 80 micro benchmarks are used to bound any modeling uncertainties and inaccuracies. Another architecture level power model is proposed by Lim et. al [17]. Their work is similar to GPUWattch. They use McPAT for power modeling. But instead of GPGPUSim, they use MacSim to acquire performance statistics as inputs to McPAT. The major difference with their work and GPUWattch is that GPUWattch only measures power but Lim et. al include Hotspot [23] to detect hot spots.

# CHAPTER 3

# GPU POWER MODEL

This chapter describes our modeling methodologies of GPU power consumption. Section 3.1 gives an overview of generic power calculation and an overview of our power model. We describe our modeled interface framework between McPAT and Multi2Sim in Section 3.2. Section 3.3 describes the modeling of different GPU components using McPAT.

## 3.1 Overview

Power is one of the key design constraints for computing systems, and processors (i.e., CPUs, GPUs, and other processing units). With an increasing number of transistors in GPU, power consumption has become a major concern. A power model that estimates power consumption at architecture level can help design architects to explore different designs and measure power and also help them make design decisions. In this thesis, we build a power model for AMD's Southern Island (SI) GPU architecture. In this process, we model several SI GPU components using basic architectural blocks available in McPAT.

Power consumption in transistor devices can be divided into two parts: dynamic power and static power as shown in Equation 3.1.

$$Power = Dynamic_{power} + Static_{power} \tag{3.1}$$

Dynamic power is from switching of transistors, so it is determined by runtime events. Static power is mainly determined by circuit technology, chip layout and operating temperature. Total dynamic power of a GPU device can be computed as Equation 3.2

$$Power_{total} = \sum_{i=1}^{n} Power_i \tag{3.2}$$

where $i$ represents each component. The dynamic power of each component is calculated in a bottom up fashion. This means that McPAT calculates the power at a basic circuit level and sums it up to get the component's power. Dynamic power dissipation of CMOS circuits is computed by:

$$Power = \alpha C V_{dd} \Delta V f_{clk} \tag{3.3}$$

where C is the total load capacitance, $V_{dd}$ is the supply voltage, $\Delta$V is the voltage swing during switching, and $f_{clk}$ is the clock frequency. C depends on the circuit design and layout of each IC component; McPAT calculates it using analytic models for regular structures such as memory arrays and wires, along with empirical models for random logic structures such as ALUs. The activity factor $\alpha$ indicates the fraction of total circuit capacitance being charged during a clock cycle. McPAT calculates $\alpha$ using access statistics from architectural simulation together with circuit properties. This depends on access count of a component, total hamming distance and cycle count for a particular simulation period. The Hamming distance is the total number of flipped bits for two consecutive accesses. McPAT assumes that all bits are flipped per cycle. McPAT also has the ability to reason about activity factors for components as long as the basic statistics information is provided by the performance simulator.

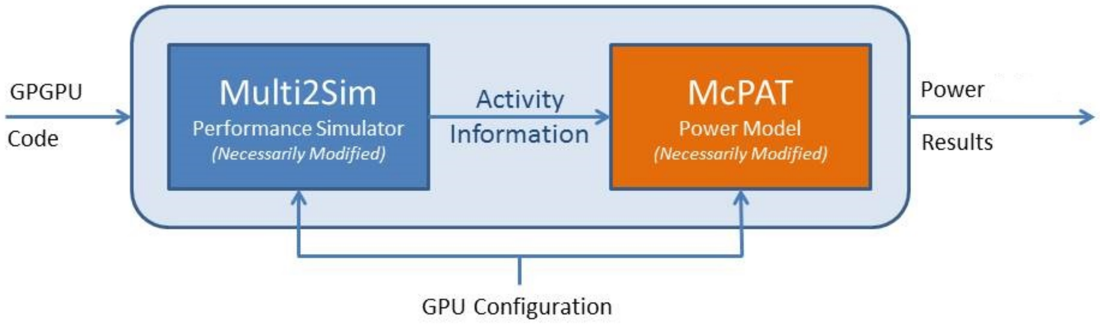


Figure 3.1. Power modeling framework.

## 3.2 Power Modeling Framework

Our proposed power modeling framework consists of two main parts, as visualized in Figure 3.1. First, a cycle-accurate GPGPU simulator simulates the given kernel and thereby generates utilization information and activity factors for modeled components of the GPU architecture. Second, a chip representation with a power model for each component uses the activity information from the simulator to produce power numbers for a particular kernel. From the chip representation, statistics about area and peak, leakage, and short-circuit power are inferred as well. For the cycle-accurate GPGPU simulator, we employ a modified version of Multi2Sim [25] that has been altered to produce access counts and other hardware activity information for all parts of the simulated architecture which are required for the next steps. Multi2Sim has been developed to simulate several different architectures. In our thesis, we focus on the Southern Island Architecture.

The chip representation and power model are provided by a heavily modified variant of McPAT [15]. McPAT integrates three different modeling tiers hierarchically to provide a flexible and highly accurate power model for CPUs: The architectural tier, where a processor is broken down into major components such as cores, caches, and memory controllers, the circuit tier, where the architectural components are mapped to basic circuit structures such as arrays or clocking networks, and the technology tier, which provides the physical parameters, such as current densities and capacitances, of the circuits. Besides this hierarchy, a unique advantage of McPAT is its combination of analytical and empirical models for individual components. We embrace the hierarchical nature of McPAT and develop a McPAT-based model for GPU components. On one hand, this requires many modifications to McPAT, as multiple components that are present in the CPU architecture model cannot be reused for GPUs, and various core components of GPU architectures, such as local memory, are not present in CPUs. On the other hand, using McPAT enables us to utilize all the integrated low-level technological information, e.g. to scale the GPU power model for a specific manufacturing process node.

To build the power model framework, we considered multiple approaches. Our first approach was to integrate McPAT into Multi2Sim. This approach enables Multi2Sim to call McPAT at each cycle with performance statistics to generate power results. We found a major problem in this approach. The problem resides in the platform incompatibility of the latest release of Multi2Sim (version 4.2) and McPAT. To be able to continue with this approach, we later used the latest build of Multi2Sim instead of latest release. The latest build of Multi2Sim is built with a combination of two different programming platforms. We tried to integrate this and McPAT by modifying the building process of Mult2Sim. This solves one problem of compilation but generated several others. The dependency of several files was broken.

This motivates us go for our second approach. In this strategy, we keep Multi2Sim and McPAT completely separated. This enables modularity between the two systems. We can modify both systems without hampering the other one. Simulation speed is important for this kind of framework. Having both tools being separated, we can run simulation using both tools in parallel or at different times. This gives the user an added level of flexibility. To pass information between the two systems we use XML files. Our modified Multi2Sim provides required performance statistics in multiple XML files which are then passed to McPAT to compute energy.

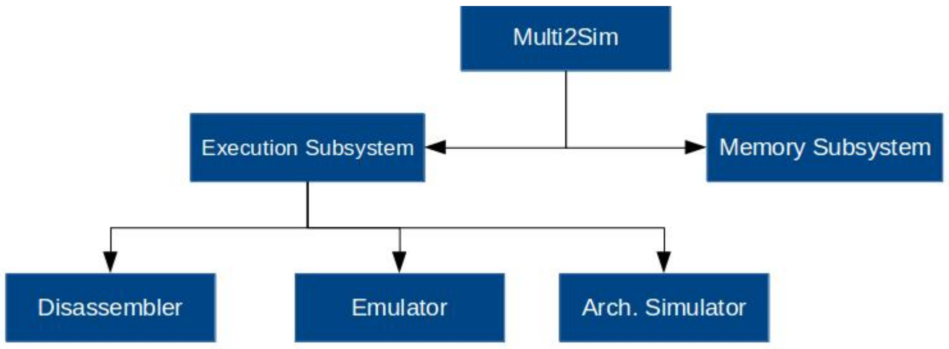


Figure 3.2. Multi2Sim architecture.

### 3.2.1 Generating Cycle Level Data

Multi2Sim is a layered system. The upper layer has a pointer to functions in the lower layer where different architectural components are simulated. Each component has three modules: a disassembler, a functional simulator and a detailed (or timing) simulator (Figure 3.2). Based on the arguments passed, functions of any module of any component can be called from the upper layer. As we are interested in the performance statistics of Southern Island structure, functions from the timing simulator for Southern Island micro architecture are called in the simulation process. Multi2Sim gives performance data for the entire execution of a program. In our thesis, we need performance statistics at the cycle level to compute power in a time domain.

The GPU simulation in Multi2Sim starts in function `SIGpuRun()` in gpu.c file. This function assigns work groups to available compute units and calls `si_compute_unit_run()` for each compute unit. `si_compute_unit_run()` simulates the work of a compute unit for each cycle until all the work groups are simulated. All compute units(CU) are called sequentially one by one in each cycle. We introduce necessary methods at this level to generate performance statistics of several components of a compute unit. Then we added another method to dump performance data into an XML file for all compute units. We decided to dump data at every 925 cycles. We chose this number because this is the clock frequency of Southern Island architecture. In this way, we can collect data at nanosecond scale. This can help us analyze the power consumption of each GPU component over time domain. Another point to be noted is that the number of compute units used in each cycle varies which depends on available work groups. This makes it difficult to format the XML correctly at this stage. We overcome the problem by adding another method to restructure the XML at an upper stage in the simulation process.

Other than compute unit statistics, we also need memory statistics. Memory information is not available from inside compute unit simulation stage in Multi2Sim. Multi2Sim simulates memory in a separate subsystem called *mem-system*. Memory is accessed as a module using the function `mod_access()`. Memory handlers are simulated in this subsystem. Different cache control protocols are also simulated in this system and are defined in `mod_handler_nmoesi_load()`. We modified Multi2Sim and wrote wrapper methods that include the complete memory subsystem inside GPU simulation process. Using these wrapper methods, we can easily access the memory modules from inside GPU simulation and produce performance statistics of each memory components at cycle the level. We also added a process to output these statistics in an XML format.

### 3.2.2 Initialization and Parsing

To successfully generate power data for an architecture using McPAT, it shoud be configured for that architecture first. The way to do that is to provide all the necessary architectural configuration information in an XML file. We maintain a master XML file that contains all necessary information for the initial configuration of McPAT for the Southern Island architecture. This file is loaded at the initialization phase of McPAT and all hardware configurations are set at this phase.

As we model GPU hardware components in McPAT, we need to pass their configuration information, too. We also do this using an XML file. It is also mandatory to update the performance statistics of the corresponding GPU hardware blocks inside McPAT. We collect these performance statistics from Multi2Sim. We modified McPAT in a way that it reads all statistics from XML data files for each sampled cycle and computes power of individual components. We must read and store all performance statistics across different sampled cycles in order by their cycle numbers. This is necessary because Multi2Sim generated cumulative statistics and not statistics specific to a range of cycles e.g., data for cycle 1850 consists of performance statistics for cycle range from 0 to 1850 and not for cycle range from 925 to 1850. We implemented a separate class to keep track of the previous cycle data and we subtract this value while loading the data into McPAT hardware blocks. We cannot produce information for particular cycle range from inside Multi2Sim because there is no way we can keep track of the previous cycle values in Multi2Sim.

## 3.3 Power Modeling of Architectural Components

This section explains the process of building the GPU power model using McPAT. In this process, we model several GPU components inside McPAT. The major difficulty of modeling GPU power is lack of detailed information about GPU hardware architecture. We model our components based on Southern Island GPU architecture. We use publicly available resources that describe the microarchitecture of GPU and rely on McPAT to model the micro-architectural blocks. Table 3.1 showcases our modeled components and the basic structure used to model these components.

The configurable input parameters to McPAT are largely composed of general technology parameters (i.e., technology node, clock frequency, etc.) and component-level values such as cache-line size, decoder width etc. These parameters are adjustable through Multi2Sim and McPAT interface. Table 3.2 lists some input technology parameters used by McPAT to define device-level characteristics of the chip and the GPU specific values assigned to these parameters in this thesis.

Table 3.1. Modeled components.

| Description | Microarchitectural Components | Basic Structures |
|---|---|---|
| Register Files | Register file banks, Operand collectors, Collection network | SRAM, Cross bar |
| Local Memory | Local memory banks, local memory network | Array, Cross bar |
| Caches | L1 cache, L2 cache | Array |
| Execution Units | ALU, FPU | Base functional units |

Table 3.2. Technology parameters.

| Parameter | Available Options | GPU |
|---|---|---|
| Clock frequency | in unit of Hz | 925MHz |
| Feature size | 16nm to 180nm | 28nm |
| Core type | Out-of-Order, in-order | In-order |
| Embedded | True, false | False |
| Interconnect projection | Aggressive, conservative | Aggressive |
| Component type | Core, uncore | Core |
| Device type | High-performance type, low standby power type, low operating power type | High-performance type |
| Homogeneous cores | 1 means all cores are the same, 0 means heterogeneous | 0 |
| Number of cache levels | Number of cache levels | 2 |
| Number of cores | Number of cores | 32 |

### 3.3.1 L1 Cache

Southern Island architecture has two kinds of L1 data caches. One cache (L1D1) is shared across four compute units and another cache (L1D2) is private to each compute unit. The L1 cache is 16KB and 4-way set associative with 64B lines. It follows the LRU replacement policy. The L1 cache has a write-through, write-allocate design with a dirty byte mask. Both caches are implemented as array structure in the McPAT. Both scalar and vector caches have the same structure except that they work on single data and multiple data respectively. We modeled both of them as data cache in McPAT.

Before describing the modeling process of cache, we discuss the cache structure. Cache

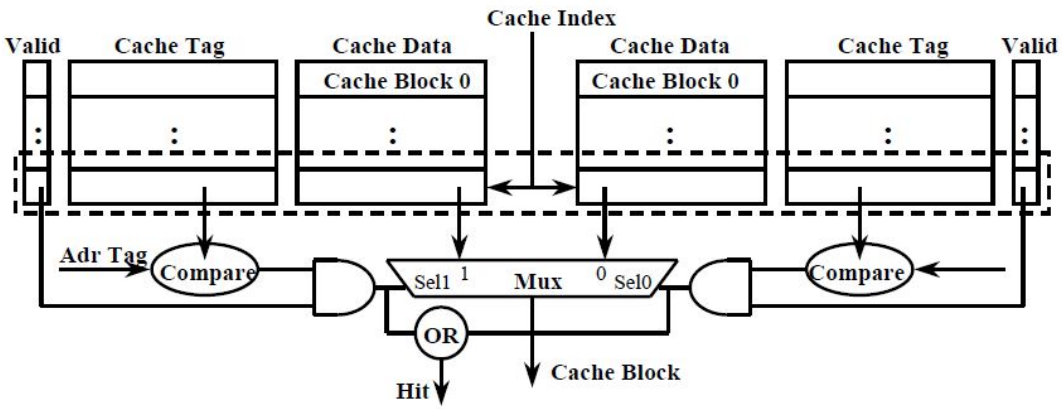


Figure 3.3. A 2-way set associative cache architecture.

is a fast on chip memory. It is on the top of the memory hierarchy. Figure 3.3 shows a 2-way set associative cache architecture. As we can see, a cache includes two major components: *tag array* and *data array*. Whenever data is needed, information is read from the tag array first. It is then compared to the tag bits of the address. In an N-way set-associative cache, N comparators are required for this. The results of the N comparisons are used to drive a valid (hit/miss) output as well as to drive the output multiplexers. These output multiplexers select the proper data from the data array and drive the selected data out of the cache. The tag and data arrays are large enough that it is inefficient to implement them as single large structures. So they are partitioned horizontally and vertically [26].

We model the cache using basic SRAM component from CACTI [28]. CACTI is a power and timing estimation tool for memory architectures. It models the basic circuit block that can be used to model cache architectures. CACTI is also integrated inside McPAT. This SRAM includes both tag array and data array. Multiple arrays are used to build a cache. Each of these is called a bank. The advantage of using banks is that they can be accessed simultaneously. A data or tag array is divided into a number of sub-arrays to reduce the delay. A collection of four sub-arrays are called a Mat. A cache block is scattered across a rows of Mats. The row number corresponding to a particular block is determined based on the block address. Each row (of mats) in an array is referred to as a sub-bank. Figure 3.4

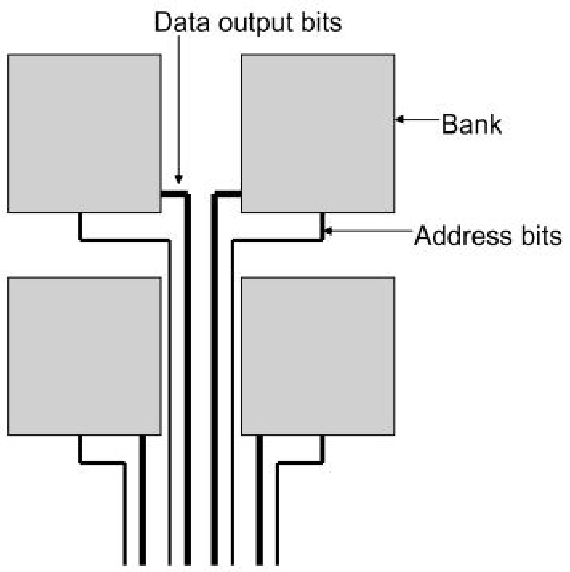


Figure 3.4. A physical cache architecture.

displays the physical structure of a cache.

Once we have modeled the cache, we also need to model the cache controller. The cache controller module consists of three components:

- *Miss buffer:* Cache stores the missed memory requests into the miss buffer and continue working on later requests while waiting for memory to supply previous misses.
- *Fill buffer:* Fill buffer accumulates parts of the cache line while they are coming from other levels of memory hierarchy.
- *Prefetch buffer:* This holds the prefetched data.
- *Write back buffer:* WBB is used to store the evicted dirty cache lines. The evicted lines are streamed out to the lower level memory opportunistically.

The write back buffer is only needed if the cache has write back policy. All of these components are modeled with Array or SRAM structure component from CACTI [28].

Once we have the cache architecture modeled, we can implement the method to compute the power for the cache components. Power is computed in two stages. In the first stage we compute the peak power for the cache component based on architectural configurations. The second stage computes the runtime dynamic power. In both cases, to get total energy consumption of the L1 cache we need to sum the individual energy consumption of each cache component.

The runtime energy consumption depends on the cache access statistics which includes cache read hit and miss as well as cache write hit and miss. We get this information from our performance simulator. We update the data cache statistics directly from the performance simulator output. To update cache controller's statistics we need to consider the cache policy and data cache statistics. Table 3.3 describes the different statistics of the cache controllers based on cache policy.

Table 3.3. Cache controller statistics.

| Cache controllers | Parameters |
|---|---|
| Write back | Cache write miss |
| Write through | Cache read miss |

### 3.3.2 L2 Cache

L2 cache is the central point of coherency in the GPU. L2 cache is shared across all 32 compute units. The L2 cache is 16 way associative, with 64B cache lines. It has the LRU replacement policy. In Southern Island architecture, it is 768KB in size. It is a write-back and write allocate design, so it absorbs all the write misses from the L1 data cache. The L2 cache is physically partitioned into slices that are coupled to each memory channel.

To model L2 cache we use the Array structure of McPAT. As this cache is shared across all compute units, we add a L2 cache component object at the device class. At the initialization of the device class we initialize the L2 cache structure. This is done in two steps. At the first step, we create a set of input parameters for the cache structure. The input parameters are passed in an XML configuration file. We need the following configuration to set the structure of L2 cache.

Table 3.4. L2 cache configuration.

| Configurations | SI value |
|---|---|
| Capacity | 768 |
| Block_width | 64 |
| Associativity | 16 |
| Cache policy | 1 (write-back with write-allocate) |

Once we have the shared cache configured, we need to pass the performance statistics from Multi2Sim to calculate runtime energy. The runtime energy is computed using the formula in Eq 3.4.

$$\begin{aligned} Power_{l2cache} = {} & ReadHit * per\_access\_read\_energy + ReadMiss * tagarray\_read\_energy + \\ & WriteMiss * tagarray\_read\_energy + WriteAccess * per\_access\_write\_energy \end{aligned} \tag{3.4}$$

## 3.3.3 Local Memory

Local memory is scratch-pad memory specific to a work-group; accessible only by work-items belonging to that work-group. Using local memory (known as local data store, or LDS) typically is an order of magnitude faster than accessing host memory through global memory. The LDS is a heavily banked design that supports intensive parallel accesses.

AMD Southern Island architecture (a.k.a GCN architecture) has 64KB of on chip local memory with 16 or 32 banks for each computing unit [2]. In our McPAT implementation of local memory we used 32 banks. Each bank contains 512 entries which are 32-bit wide. A bank can read and write a 32-bit value across an all-to-all crossbar. Typically, the LDS will coalesce 16 lanes from two different SIMDs each cycle, so two wavefronts complete every 4 cycles. An instruction which accesses different elements in the same bank takes additional cycles to finish.

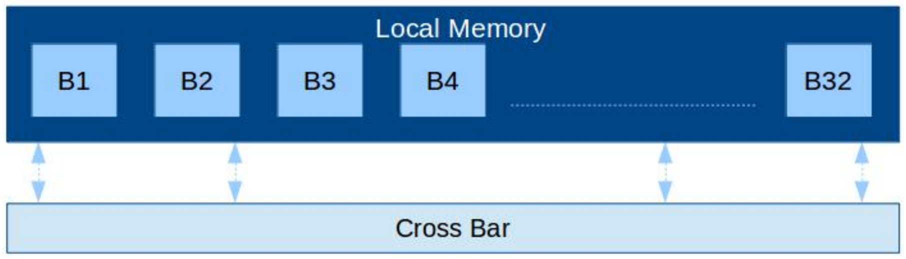


Figure 3.5. Local memory model.

McPAT does not have local memory component. We added this GPU specific component to mimic GPU hardware architecture. To implement local memory we need on chip memory structure and and interconnect module. Our implemented model is displayed in Figure 3.5.

We leveraged array structure component of McPAT along with a 64×32 crossbar. As with other components, first, we need to configure the local memory and then we need to add compute methods. The process starts from adding elements in the McPAT configuration files and adding methods to parse this. Local memory is an on chip memory and it has similar architecture as data cache. We are using Array structure to build the local memory structure. In the initialization phase, we set the parameters as we have done with data cache. The main difference with data cache and local memory is that local memory communicates with the execution unit through an interconnection network. To build this interconnection network we use a basic module from CACTI called crossbar.

A crossbar has every node of the network connected to every other node (Figure 3.6). Because any node can send simultaneous messages to every other node in the system without conflicts, this network topology is nonblocking. Before the initialization of the local memory, we build a new crossbar network. To build a cross bar we need four configuration parameters. Table 3.5 shows the configurations needed to build a cross bar network.

After local memory and crossbar are configured and initialized, we write the method to compute the energy consumed by these components. The computation is done in two

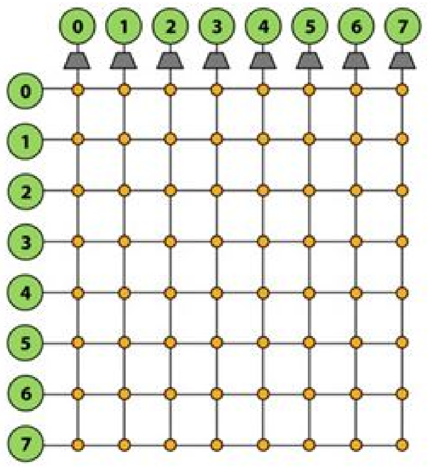


Figure 3.6. Crossbar switch.

Table 3.5. Crossbar configurations.

| Configurations | SI value |
|---|---|
| Number of input | Number of functional units |
| Number of output | Number of functional units |
| Bandwidth | 32 |

steps. First, based on the performance statistics, we compute the power consumption of local memory. The formula we use here is tantamount to that of L1 cache. Second, we calculate the power consumption of crossbar. There is no separate performance statistics for crossbar because its statistics completely depend on the statistics of local memory. Crossbar is accessed every time local memory is accessed. So, to compute the power of a crossbar we need to compute the total access of local memory and multiply it with the per access energy for crossbar. The formula is shown in Eq 3.5.

$$Power_{crossbar} = BasePower_{crossbar} * (ReadHit_{localmemory} + WriteHit_{localmemory}) \tag{3.5}$$

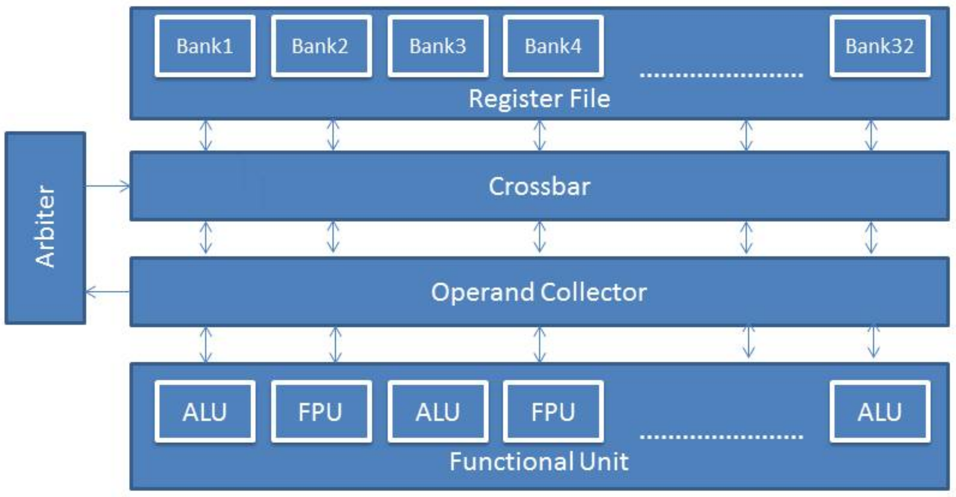


Figure 3.7. Register file architecture.

### 3.3.4 Register file

The next component modeled is the register file. A register file is an array of processor registers. It is used to stage data between memory and the functional units on the chip. Register files are multi-banked which allows multiple access simultaneously. The register files in GPU are different from that of CPU in that there are tens of thousands of registers to hold the register state of all the threads running on an compute unit. Moreover, the register file has to support many concurrent accesses. GPUs have a multi-banked register file architecture to avoid the area and power overhead of a multi ported register file. Crossbar networks and operand collectors are used to transport operands from different banks to the appropriate SIMD execution lane. Arbiter is responsible for serializing the register file access, and the operand collectors are used to buffer the data already read from the register file. The register files bandwidth and size determines the number and size of banks. Each compute unit contains 65536 32-bit registers [25], which are shared by all threads executed on the same compute unit. We model the register file as 32 dual-ported (one read and one write port) banks (Figure 3.7).

We estimate the power consumption for these memory arrays using CACTI. We model a crossbar interconnection network. This is used to transfer operands from register file banks to operand collectors. The crossbar interconnection network model is used while computing the power of our register-file power. The crossbar network's parameters are determined by the number of register-file banks. Each wavefront operand accesses the crossbar and arbiter once. So, we divide the total register access by 64 to get the access count of the crossbar and arbiter. For the Southern Island architecture, we model a crossbar network with 32-input $\times$ 32-output. The operands read from a register file bank are routed to the proper operand collector. We model each operand collector as an Array structure and initialize in a similar way as cross bar. Operand collector statistics is equal to register file access statistics.

### 3.3.5 Functional Units

Each compute unit consists of an execution unit. The execution unit consists of two units: *scalar unit* and *vector unit.* The major difference between these two execution units is the the number of values they work on. The Scalar unit works on a single value while vector or SIMD units work on multiple values [2]. Another difference is their instruction formats and encodings [3].

- **Scalar Unit:** The Scalar unit consists of an ALU(functional unit) that is responsible for executing scalar arithmetic-logic operations. The full integer ALU acts as an address generation unit (AGU) to read from the data cache. The operations performed on this unit consist of 32-bit integer arithmetic. This ALU can also perform operations on the Program Counter. AMD has defined 17 instructions to be executed on this ALU including signed and unsigned integer operations.

- **Vector Unit:** A Vector unit consists of two functional units ALU and FPU. ALU is responsible for integer operations whereas FPU is allocated to perform double precision floating point operations with full speed denormals and all rounding modes. The functional units can execute a single precision multiply-add or integer operation. The integer multiply-add is particularly useful for calculating addresses within a work-group [2].

Both scalar and vector units consist of only two different types of functional units: *ALU* and *FPU*. To model these components in McPAT, we leverage McPAT's provided logical units [15]. We model two functional units inside McPAT to mimic Southern Island execution units. They are : *ALU* and *FPU*.

The first challenge to model the functional units in McPAT is to get the proper statistics from our performance simulator that are required to generate power from McPAT. We require certain performance statistics to compute power of our modeled functional units using McPAT. Table 3.6 describes the statistics required.

Table 3.6. Performance statistics for functional unit.

| Parameter | Definition |
|---|---|
| *ialu_accesses* | Number of instruction issued to the ALU |
| *fpu_accesses* | Number of instruction issued to the FPU |

As our performance simulator Multi2Sim [25] does not provide these statistics, we need to modify the source code as necessary to generate statistics that comply with McPAT. Multi2Sim gives the total number of scalar unit instructions and the total number of vector unit instructions. In the initial implementation, we used scalar instructions as *ialu_access* as the scalar unit only works with integers. Then we modeled vector instruction as *fpu_access*. But this was not an effective implementation as in this way, some integer instructions are using FPU in McPAT. This motivates us for further modification of Multi2Sim. Our modifications comprise two steps- implementing a mechanism to generate required statistics for each functional unit and output them for each compute unit at cycle level.

To generate functional unit specific instructions and count their numbers we need to identify each instruction at the microcode level. Multi2Sim implements any structure in three segments: *timing segment*, *emulator segment* and *assembly segment*. The first segment defines the architecture and collects performance statistics, the second level emulates the operation of the architecture and the last segment holds the instruction set and the format of different instructions. We need to modify the first two levels. First, we modified the compute unit component and added required statistics. To update the statistics counter we modified the emulator segment as well as the timing segment. In the timing segment, Multi2Sim works sequentially for each cycle and for each compute unit. At each cycle, for each compute unit, we modified the issue stage of the compute units. In this stage, the micro instructions are decoded. We check each microinstruction's type and format. We compare the microcode instruction type with instruction type defined in assembly segment. Based on the result of our comparison, we update corresponding statistics of the compute unit. Multi2Sim increases the vector instruction count once for each wavefront. This does not give us the total number of instructions. We changed the wavefront emulation method and added statistics for the wavefronts. For each compute unit, while counting the vector instruction count we check the work item count of the assigned wavefront and multiply our instruction count with the work item count. This way, we get the total number of instructions. One point should be noted here, we differentiate the scalar ALU integer operations and vector ALU instructions. We do this because they have different latency and power depends on the latency. At each cycle in the fetch state, Multi2Sim emulates the instructions. We modified the emulator component. At each cycle of the emulator execution phase, based on the micro instruction type, name and format we update the emulator statistics. Once we have the statistics update mechanism implemented, we put that information in our previously implemented method to dump this information in a XML file format for each compute unit.

The next step is to model the functional units inside McPAT. We have added three type of functional units to our compute unit's execution unit. In the initialization phase of the functional units, we specify the type of the functional unit: *ALU* or *FPU*. We also need to pass the number of functional units for each compute unit, compute unit id, operating voltage and latency. Based on the technology node and operating voltage, McPAT gives us a per access energy of the logical unit. We use this to calculate the energy for each functional unit.

$$runtime\_power = access\_count * per\_access\_energy * coefficient * latency \tag{3.6}$$

Equation 3.6 describes how the power is calculated for functional units. Coefficient depends on the technology node. McPAT has fixed coefficients for each technology node from ITRS road map [5]. That coefficient is then multiplied by a scaling factor if the technology node is not implemented in McPAT. The latency information is fixed and we get this from the Southern Island architecture configuration file of Multi2Sim.

## 3.4 Hotspot and HeatMap

Power dissipation is spatially non-uniform across the chip which leads to hot spots and spatial gradients that can cause timing errors or even physical damage. Heat map or heat distribution over time to identify hotter segments in an architecture is an important segment of this thesis and its future improvement. We need a thermal model to accurately characterize current and future thermal stress, temporal and spatial non-uniformities, and application-dependent behavior. To evaluate architectural techniques for managing thermal effects a model of temperature is needed. Developing a new thermal model is out of the scope of this thesis. Fortunately, there exists a well established thermal model and hot spot detection tool named HotSpot [23].

HotSpot works by converting a power model to equivalent RC circuit model. This is based on the duality [13] principal between heat transfer and electrical phenomena. Heat flow can be described as a current passing through a thermal resistance, leading to a temperature difference analogous to a voltage. The rationale behind this duality is that current and heat flow are described by exactly the same differential equations for a potential difference. Thermal capacitance is also necessary for modeling transient behavior, to capture the delay before a change in power results in the temperature reaching steady state. Lumped values of thermal R and C can be computed to represent the heat flow among units and from each unit to the thermal package.

$$R = \frac{t}{k.A} \tag{3.7}$$

where t is the thickness of the block, A is the area and k is the thermal conductivity of the material per unit volume.

$$C = c.t.A \tag{3.8}$$

where c is the thermal capacitance per unit volume of an element.

The thermal Rs and Cs together lead to exponential rise and fall times characterized by thermal RC time constants analogous to the electrical RC time constants. In the thermal-design community, these equivalent circuits are called compact models, and dynamic compact models if they include thermal capacitors. This duality provides a convenient basis for an architecture-level thermal model. For a micro-architectural unit, heat conduction to the thermal package and to neighboring units are the dominant mechanisms that determine the temperature.

Hotspot assumes a package as a three layer die: *Silicon Die*, *Heat Spreader* and *Heatsink*. The silicon die layer holds the block design. Hotspot tool assumes to have registers and capacitance between each block and each layer.

Figure 3.8 shows a sample HotSpot RC model. The RC model consists of a vertical model and a lateral model. The vertical model captures heat flow from one layer to the next, moving from the die through the package and eventually into the air. For example, $R_{v2}$ accounts for heat flow from Block 2 into the heat spreader. The lateral model captures heat diffusion between adjacent blocks within a layer, and from the edge of one layer into the periphery of the next area e.g.,$R_1$ accounts for heat spread from the edge of Block 1 into the spreader, while $R_2$ accounts for heat spread from the edge of Block 1 into the rest of the

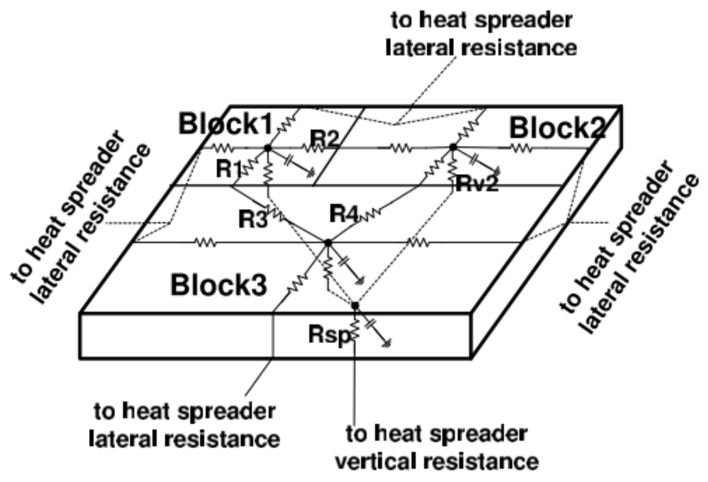

to heat spreader lateral resistance
Block1
R2
Block2
R1
Rv2
R3
R4
to heat spreader lateral resistance
to heat spreader lateral resistance
Block3
Rsp
to heat spreader lateral resistance
to heat spreader vertical resistance

Figure 3.8. HotSpot RC model for a floor plan with three architectural units [23].

chip. At each time step in the dynamic simulation, the power dissipated in each unit of the die is modeled as a current source at the node in the center of that block.

Adding HotSpot to our power-performance framework consists of two steps. First, initialization information is passed to HotSpot. This consists of configuration information and an adjacency matrix describing the floor plan and an array giving the initial temperatures for each architectural block. Then at runtime, the power dissipated in each block is averaged over a user-specified interval and passed to HotSpots RC solver, which returns the newly computed temperatures. Figure 3.9 demonstrates the steps involved in generating the heat map.

- Configuration and floor plan: We configure the HotSpot tool with our our device's frequency. We also set the ambient temperature. Other configurations include the thermal conductivity and resistance of chip and heat sink. Next we generate a floor plan using floor plan generator. Floor plan is the die design of the GPU HW. This includes all the components of a GPU HW. We pass it to the HotSpot tool. A floor plan description file holds all the information needed to be used by the HotSpot tool.

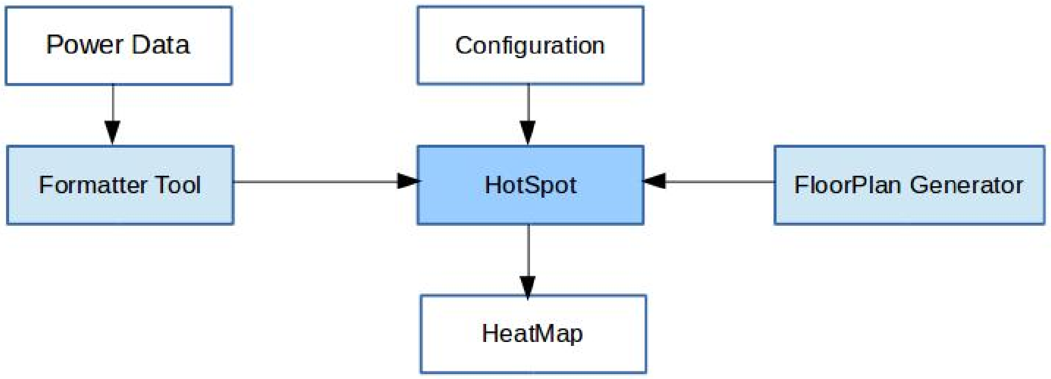


Figure 3.9. Flowchart for generating heat map.

The file follows the following format:

Table 3.7. Floor plan file structure.

| Unit-name | Width | Height | Left-x | Bottom-y |
|---|---|---|---|---|

All the units are in meters. One thing has to be noted is that the (x,y) value is relative to the lower left most corner of the die. This means the (0,0) point is on the left-bottom corner of a die. We modified the base floor plan generator methods to produce a floor plan for our GPU HW. Throughout all the benchmark tests, we consider this floor plan fixed. HotSpot provides a tool to convert the floor plan file (*.flp*) file into a visual design and writes it into a *.pdf* file.

- Power Data: To get thermal data and heat map, we need to pass power data for each individual component of the GPU HW. We get the data from our power-performance framework. We have developed a tool to convert the power data from our power-performance framework to HotSpot in an acceptable format.

Once HotSpot has the floor plan and power data component it computes the R and

Table 3.8. Power trace file format.

| Cycle | Component 1 | Component 2 | ....... | Component N |
|---|---|---|---|---|

C value and then generates a RC circuit. The differential equation to solve a RC circuit is as follows:

$$I = C.\frac{dV}{dT} + \frac{V}{R} \tag{3.9}$$

$$\Delta V = \frac{I}{C}.\Delta t + \frac{V}{RC}.\Delta t \tag{3.10}$$

Based on the duality [13], equation 3.10 can be written as follows [22]:

$$\Delta T_i = \frac{P_i}{C_i}.\Delta t + \frac{T_i}{R_i C_i}.\Delta t \tag{3.11}$$

RC circuit of equation 3.11 is then solved based on fourth order Runge-Kutta (RK4) method.

# CHAPTER 4

# VISUALIZATION

It is important to display the data we get from our power model. Our power model framework thus includes a graphical user interface with multiple options that display the energy consumption we get from our power model. We display power consumption not only in a time domain at nanosecond level but also in the space domain. This shows the heat dissipation across different architectural components over time due to the power consumption of different components. This chapter describes several techniques that we followed to visualize and display power data in a space time domain.

## 4.1 Excel

At the beginning, we thought of having an interactive excel sheet to display data and graph. The benefits of using excel is:

- Portable. It can be transferable to any platform.
- Easy to manipulate. If someone wants different type of graph, one can easily do that as Excel provides lots of different graph options.

We can use macro or C++ Excel interface to program interactive Excel sheets to parse and display data.

The first approach requires using of open Excel development interface for C++. We installed `libreoffice-dev`. This holds all the necessary libraries needed to build a libreoffice application using C++. The main challenge of developing an application with open office is the lack of enough documentation on open office library.

The next approach was to use Excel macro (written in Visual Basic) to make Excel sheets interactive as an application. We did this in two steps. First, we counted the number of cycles present in the output data and drew multiple GPU blocks equal to the total number of cycles using macro code. Each GPU block contains all the components modeled using McPAT. Second, for each cycle, power consumption data is parsed from the output files and values for different components of a GPU block are collected and stored in a data structure. These values are then compared against a predefined threshold and the corresponding component of each GPU block in the excel is colored based on the difference between the actual value and the threshold value.

A major challenge of this approach is writing the macro to parse input data and format excel cells. This approach has several disadvantages and is not suitable for our framework.

- Excel cells do not have a good visualization. It is hard to have a proper gradient color values for the cells to demonstrate heat.
- Excel is not scalable as there are limitations on the number of columns and we have less control over the cells.

## 4.2 OpenCV

We also developed a tool to display our data using OpenCV [1]. OpenCV is a cross platform library of programming functions mainly aimed at real-time computer vision. It has built in

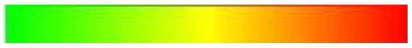


Figure 4.1. Power data gradient.

functionalities to display and manipulate images. The advantage of using OpenCV is that we can draw geometric shapes and write text on a image using OpenCV.

Our display tool includes a class for `PowerData`. This class includes the all modeled GPU components. Each component consists of three items: area, peak dynamic and run time dynamic power. The system starts with counting the number of cycles in the output. Then we parse and store data into an array of `PowerData` objects. We create a large image matrix of size 725×210 pixels using OpenCV. The size was carefully taken so that all GPU components can be drawn in a single matrix. All GPU components are drawn on this image matrix. Our initial design includes 32 cores of size 40×30 pixels, 8 L1 cache of size 180×30 pixels and one L2 cache of size 715×30 pixels. After drawing the individual components, we color them based on their power data. At first, we normalized the power data into a range from 0 to 255 to use it in RGB color space. We want to have a gradient look (Figure 4.1) to have a better understanding of power transitions.

In Figure 4.1, green means no power is consumed and red means highest possible amount of power is consumed. We convert power data into RGB values using the following formula.

```
B = 0
if $0 < \text{power} \leq 127$ then
    $green \leftarrow 255$
    $red \leftarrow 2 * power$
else
    $red \leftarrow 255$
    $green \leftarrow 255 - 2 * (power - 128)$
end if
```

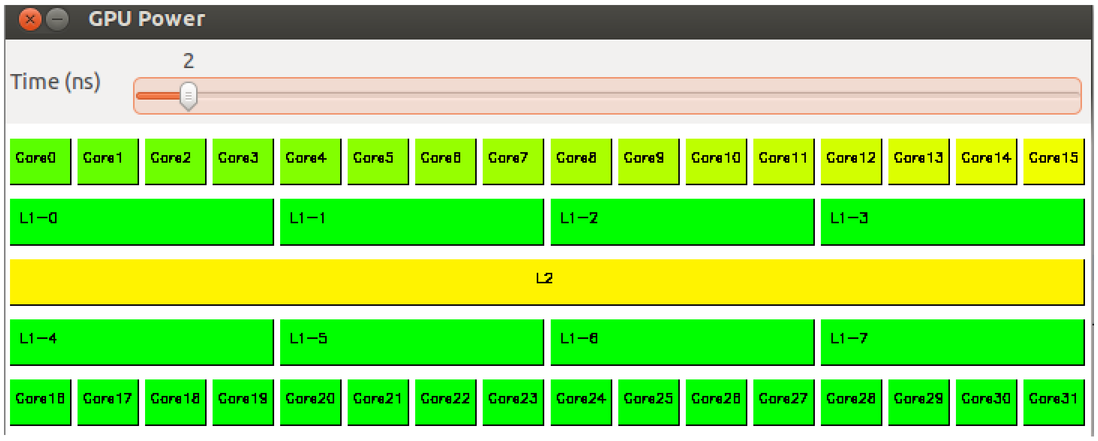


Figure 4.2. Power data visualization using OpenCV.

We apply this formula to each component to get their color based on the power consumed by that component. A region of interest (ROI) is selected with appropriate pixel positions and height and width. The color of ROI is set to the color we get from the above formula. It has to be noted that every time an object or text is drawn on an image matrix, it overwrites the previous image matrix values. As a result text has to be written again on ROI after coloring the ROI.

A movable track bar is added to the tool. It can move from zero to maximum number of cycles. This gives the facility to observe power data at different times of program execution. All GPU blocks are redrawn every time the cycle value is changed using the track bar with power data acquired from that cycle. Figure 4.2 displays the current GUI for power data visualization.

This solution still has some potential problems.

- We cannot pass user input to the application.
- It is not scalable and we cannot add elements to it.
- We cannot draw graphs using OpenCV.

To overcome the problems with the previous two tools, we developed our final tool using Qt framework.

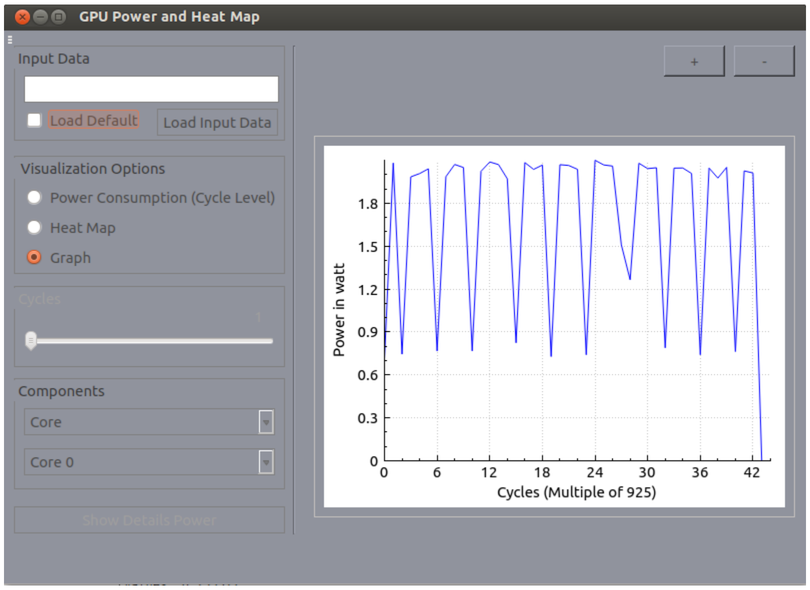
## 4.3 Qt Framework

Qt is a cross-platform application and UI framework used for device creation and application development supporting deployment to over a dozen leading platforms. The Qt framework comprises of modular cross-platform C++ class Qt libraries and an Integrated Development Environment with Qt Creator IDE. The visualization segment consists of two segments: *settings* and *display*. In the settings segment, the user can pass input data and select any of the three visualization options: *Power Consumption*, *Heat Map* and *Graph*.

We have modularized the visualization part so it is not dependent on the power performance tool. It is an stand alone application that only needs a text file including the power information. As it is based on Qt, it is portable, too.

The PowerData class works with the power data. It reads data from a text file and holds the data to show using the UI class.

Qt does not provide graph drawing facilities in the base package. We have integrated another package named QCustomPlot which provides a facility to draw different kinds of graphs. With the graph option selected, the user can have the option of selecting any HW component from a drop down list which will then show the power graph for that component over time. The heat map option shows the heat distribution over time across different components of the GPU HW. The power consumption option gives the user an option to see consumed power at a certain cycle. The user can move a slider to see power consumption of different components at different cycles. Figure 4.3 and Figure 4.4 displays screen shots of our graphical interface.


GPU Power and Heat Map

Input Data

Load Default

Load Input Data

Visualization Options

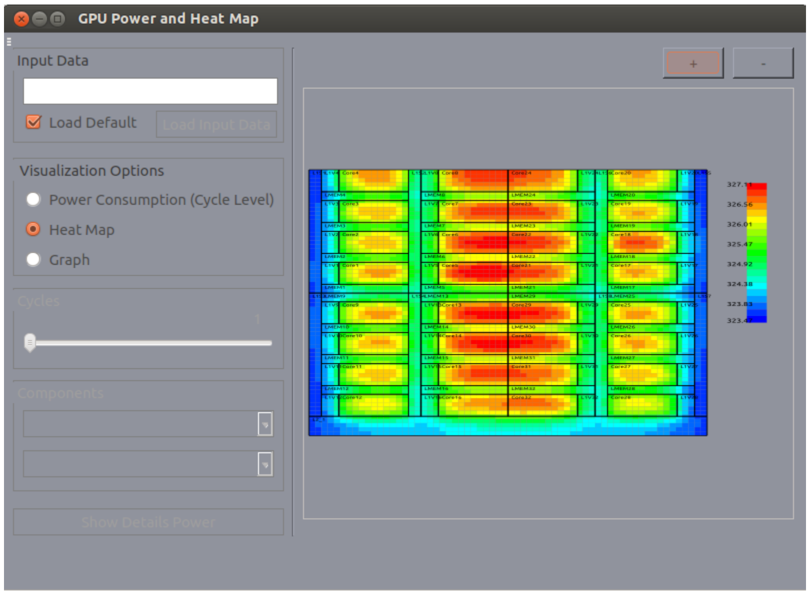
Power Consumption (Cycle Level) [✓ Graph] Heat Map

Components

Core

Core 0

Figure 4.3. Graphical display of power for a single component.


GPU Power and Heat Map

Input Data

[✓ Load Default]

Load Input Data

Visualization Options

Power Consumption (Cycle Level) [✓ Heat Map] Graph

Figure 4.4. Graphical display of heat consumption of each component for a program.

# CHAPTER 5

# EXPERIMENTS AND RESULTS

This chapter presents our experiments and their results. We also describe the benchmarks used and include the hot spot visualization. All experiments were simulated using our modified Multi2Sim for performance statistics, modified McPAT for power statistics and configured HotSpot for heat profiling. All power data is in watt (W) and temperature data is in Celsius for the graphs and Kelvin for the heat map.

## 5.1 Benchmark Programs

We use the following benchmark programs in our experiments to cover different power consumptions across different hardware components.

- **Matrix Multiplication:** This is the most widely used benchmark program for any GPU experiment. The algorithm is highly parallel and possesses interesting memory access pattern. The program simply takes two matrices as input and multiplies them. The result is stored in another matrix.

- **Bitonic Sort:** As the name suggests, this is a sorting algorithm. It was chosen because Bitonic Sort is well-known parallel algorithm. It works by creating bitonic sub-sequences in original array, starting with sequences of size 4 and continuously merging sub-sequences to generate bigger bitonic subsequences. Finally, when the size of this subsequence is the size of the array, it means the entire array is a bitonic sequence. Repeating the steps makes the array a part of a bitonic subsequence that is twice the size of the array. Thus, this part of the sub-sequence, the full array, is monotonic (sorted) [4].

- **Reduction:** It is another widely used parallel algorithm that highly leverages the parallel computation power of GPU. In this program, an array is divided into multiple blocks. Each block invocation of the kernel reduces the input array block to a single value; it then writes this value to output and reduces the block sums to a final result, which is sent to the host program.

## 5.2 Results

- **Matrix Multiplication:** The program takes two matrices as input and multiplies them. The result is stored in another matrix. We took two matrices of height 128 and width 128. The program takes 40594 cycles to complete on Multi2Sim. The program has total 16 work groups which is scheduled to 16 compute units of the GPU. This resulted in a power consumption only by the 16 compute units. Figure 5.1 shows the average power consumption across compute units. As 16 work groups are scheduled to only 16 compute units, other 16 compute units are idle and do not consume power. We can also see that the average power consumption of all these 16 compute units are similar. This is because the work group scheduler evenly distributes the works across all compute units. Thus all compute units have similar work loads and they consume similar amount of power. Figure 5.2 describes the power consumption of compute units across all cycle intervals. As expected, they have similar pattern. Figure 5.3 describes the average power distribution of each component in GPU hardware. Functional units

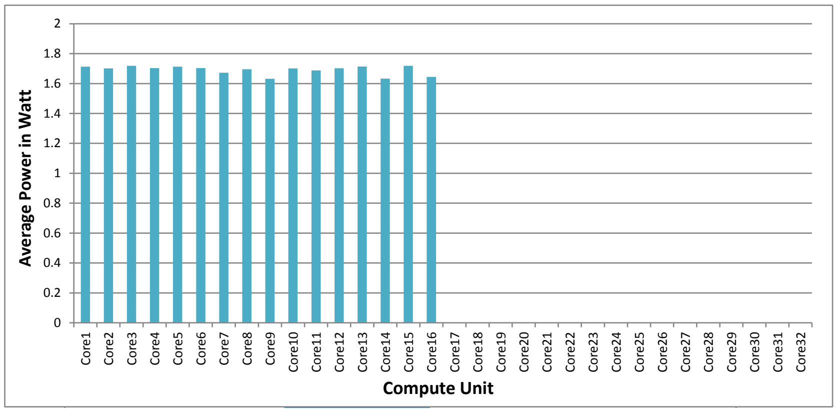


Figure 5.1. Average power consumption of different units.

and register files consume the most power (85%). Figure 5.4 demonstrates the temperature of the compute units over the total execution time period. HotSpot assumes the initial chip temperature is 45°C. 16 idle compute units have temperature close to this. Small temperature rise shown in Figure 5.4 is caused by the heat transfer from the active 16 compute units. Figure 5.5 shows the heat map for the matrix multiplication application. HotSpot uses localized heat information to color. If an area is colored red that means that area is relatively hotter than its surroundings. As the first 16 compute units and their vector caches are used in the simulation, they consume power and eventually dissipate heat to surroundings. This results in more temperature at the center of the compute units.

- **Bitonic Sort:** The next application tested is Bitonic Sort. We used 1000 elements to sort. This program runs in 67195 cycles and has a total 28160 work groups. All compute units are evenly distributed to compute units and as total work group number is more than 32, all compute units are used. We show the power consumption across cycle intervals in Figure 5.6. The temperature consumption is shown in Figure 5.8.

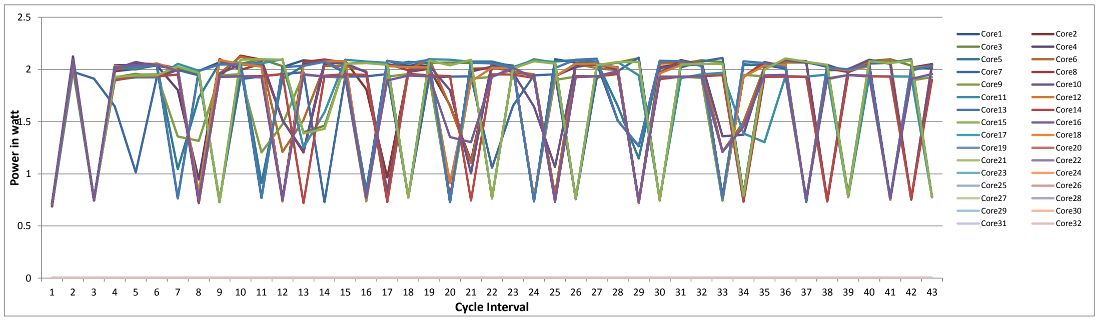


Figure 5.2. Power consumption of compute units across cycle intervals.

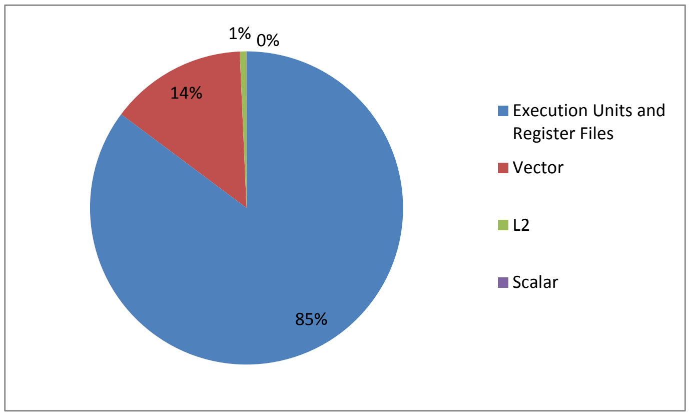


Figure 5.3. Power consumption distribution.

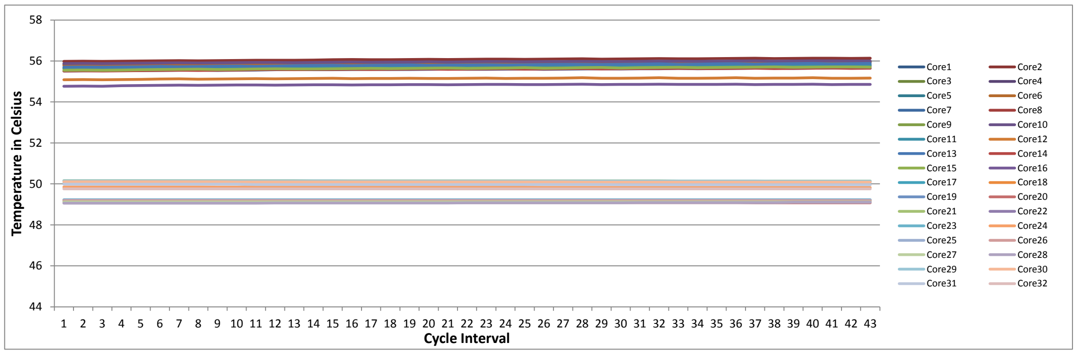


Figure 5.4. Temperature of compute units across cycle intervals.

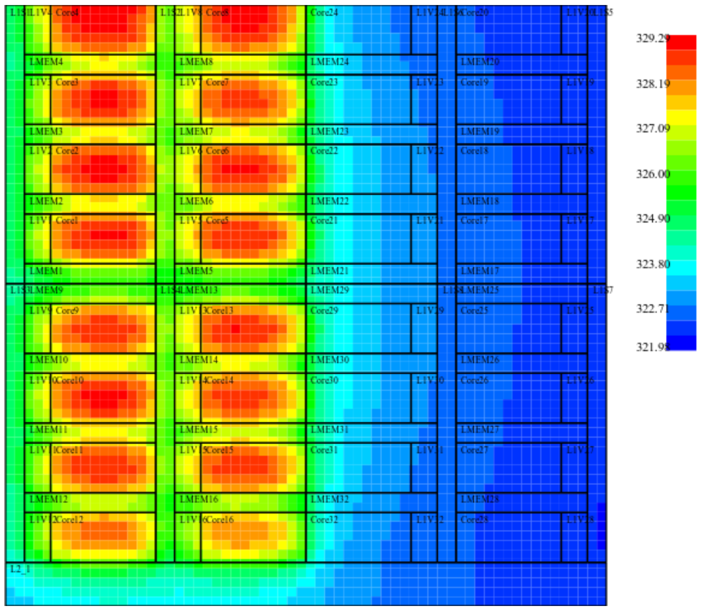


Figure 5.5. Heat map for Matrix Multiplication operation.

We can see that all 32 compute units are heated.

Figure 5.9 shows the heat map of the process. All compute units are marked red as they are most power consuming segment. We see that compute units in the middle are marked more red than the compute units on either side. There are caches in between the compute units on either side which consume less power and produce less heat. As a result they absorb some heat from the compute units. But for compute units in the

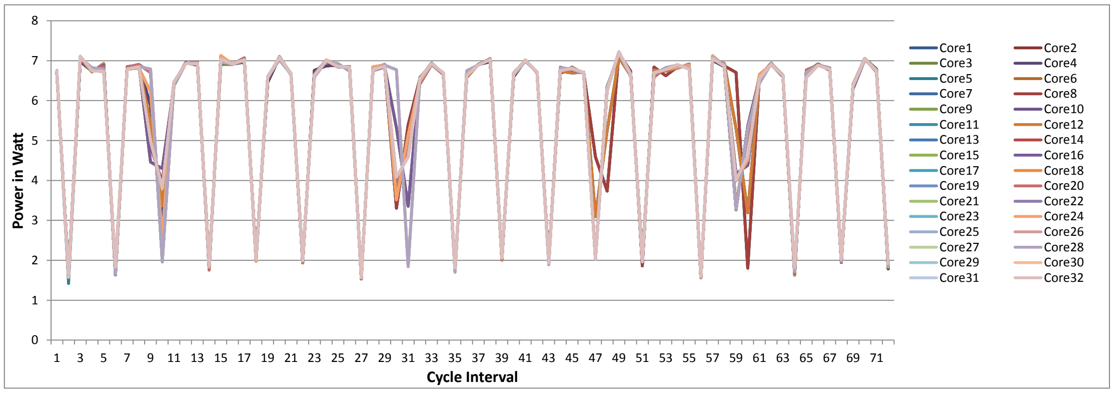


Figure 5.6. Power consumption of compute units across cycle intervals(Bitonic Sort).

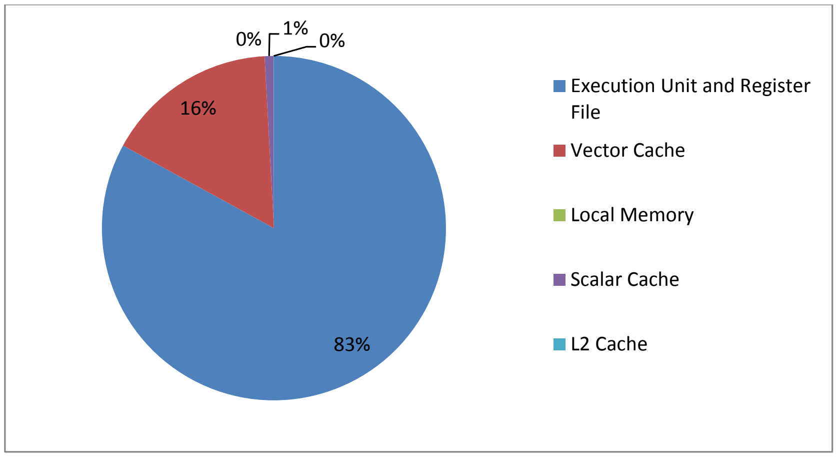


Figure 5.7. Average power consumption of different units(Bitonic Sort).

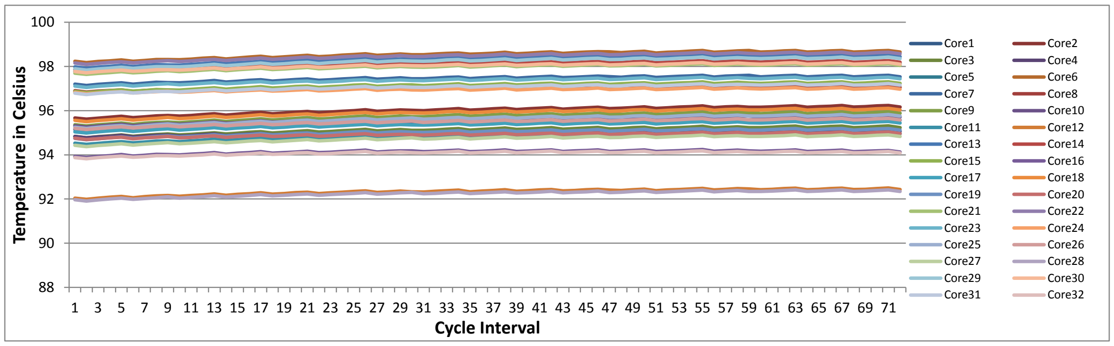


Figure 5.8. Temperature of compute units across cycle intervals (Bitonic Sort).

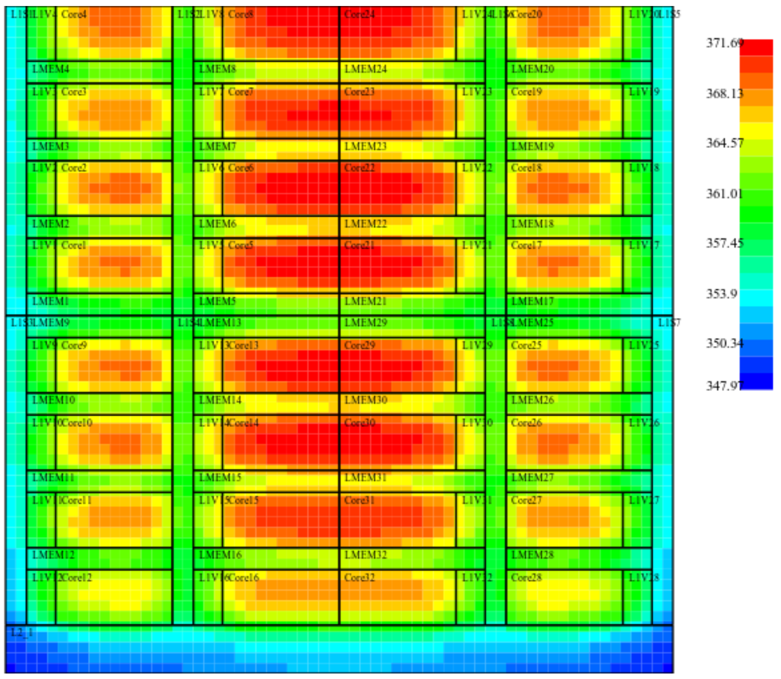


Figure 5.9. Heat map for Bitonic Sort process.

middle, there is no segment with less power consumption between the compute units in the middle which can work as a heat sink to absorb heat. So, heat from compute units in the middle make them more heated compared to the other compute units.

- **Reduction:** This small application takes 1000 elements as input. In this case, we only have one work group and the program takes 7720 cycles to complete. Only one work group is scheduled to one compute unit. As a result only one compute unit will be used for execution and only that compute unit consumes power. This behavior is shown in Figure 5.10. We can see only one line which represents one active compute unit. Temperature graph (Figure 5.11) also demonstrates the similar pattern where only one compute unit is hot and the other compute units are in initial chip temperature. Figure 5.12 shows the heat map. Only one compute unit is marked red as that is the only compute unit that has been used during entire program execution.

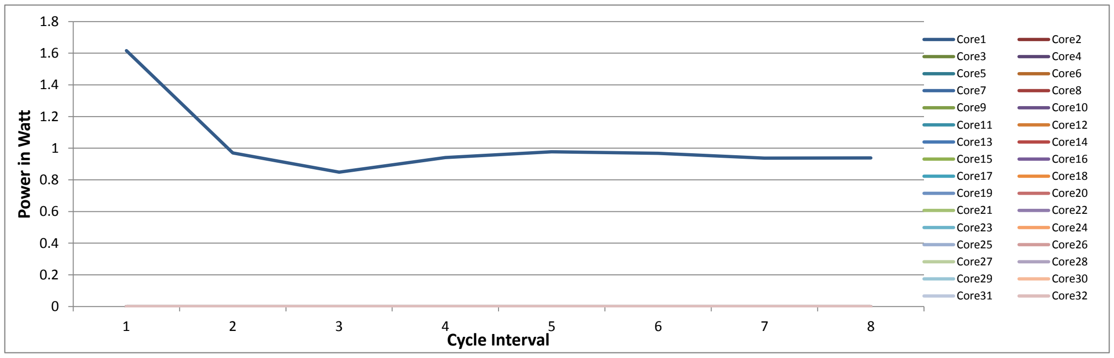


Figure 5.10. Power consumption of compute units across cycle intervals(Reduction).

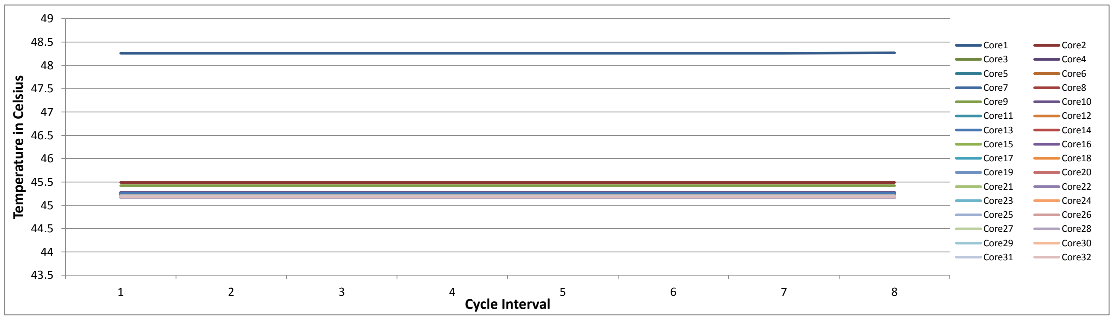


Figure 5.11. Temperature of compute units across cycle intervals (Reduction).

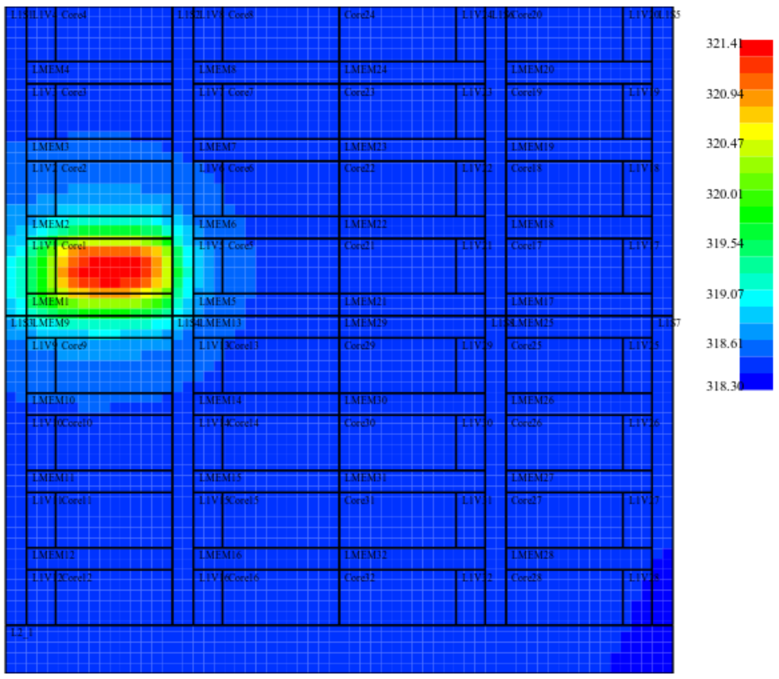


Figure 5.12. Heat map for Reduction process.

# CHAPTER 6

# CONCLUSION

The architecture community needs a GPU power model. Though there have been works on NVIDIA's Fermi GPU architecture, no work has been done on AMD Southern Island GPU architecture. In this thesis, we develop a framework for power model and hot spot detection targeting AMD Southern Island GPU architecture. Our model is configurable and provides power and temperature data at a cycle level. In this thesis, we developed a power model based on McPAT. We also presented methodologies to model different components of GPU hardware using McPAT. Our model estimates the power consumption of different GPU components by using performance statistics from Multi2Sim. Our framework integrates HotSpot and detects hot spots in the GPU hardware floor plan. We also developed a graphical user interface that brings everything under one umbrella. Our framework is robust and can be extended to model future 3-D chip power and heat. Future work of this thesis includes refinements and validation. This is an important task providing that there might be many modeling uncertainties.

GPU hardware architecture is not well documented and very little information is available due to proprietary reasons. We had to rely on the limited information that vendors provide. Because of this, we had to make some assumptions regarding hardware microarchitecture in our model. We need the refinement and validation process to adjust these assumptions. We need to verify the data generated from our model against actual physical power data. It is not possible to get physical power values for each component of a GPU device. We can follow micro-benchmarking process to do this. In this case, each micro-benchmark is designed in such a way to isolate a specific component's power consumption, the error in total power for the micro-benchmarks would help to improve the modeling parameters of the exercised component during refinement process. We could not perform this step due to time constraint and lack of hardware equipment to physically measure the power of a GPU device. This is left as a future work. Notwithstanding its limitations, our model may offer some insight on power and heat dissipation of different GPU hardware components over time during the execution of a program.

# BIBLIOGRAPHY

# BIBLIOGRAPHY

[1] Opencv (open computer vision). `http://opencv.org/`,.

[2] AMD. Amd graphics cores next (gcn) architecture. `http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`, 2012. Accessed: 2014-04.

[3] AMD. Southern islands series instruction set architecture. `http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf`, 2012.

[4] AMD. Amd sdk sample browser. `http://developer.amd.com/app-sdk/codelisting.php?q=Accelerated%20Parallel%20Processing`, 2014. Accessed: 2014-04.

[5] Semiconductor Industries Association. Model for assessment of cmos technologies and roadmaps (mastar). `http://www.itrs.net/models.html`, 2007.

[6] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[7] David Brooks, Vivek Tiwari, and Margaret Martonosi. *Wattch: a framework for architectural-level power analysis and optimizations*, volume 28. ACM, 2000.

[8] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[9] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. Tree structured analysis on gpu power study. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 57–64. IEEE, 2011.

[10] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[11] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[12] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 135–140. ACM, 2001.

[13] Al Krum. *Thermal Management in The CRC Handbook of Thermal Engineering.* 2000.

[14] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 487–498. ACM, 2013.

[15] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[16] Zhi Li. *Power Modeling and Optimization for GPGPUs.* PhD thesis, University of Kansas, 2013.

[17] Jieun Lim, N Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. Power modeling for gpu architecture using mcpat. *Georgia Institute of Technology, Tech. Rep*, 2013.

[18] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[19] Aaftab Munshi. The opencl specification, 2012.

[20] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122. IEEE, 2010.

[21] Karthik Ramani, Ali Ibrahim, and Dan Shimizu. Powerred: A flexible modeling framework for power efficiency exploration in gpus. *Worskshop on GPGPU*, 2007.

[22] Kevin Skadron, Tarek Abdelzaher, and Mircea R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 17–, Washington, DC, USA, 2002. IEEE Computer Society.

[23] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, March 2004.

[24] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 673–686. IEEE, 2013.

[25] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 62–68, 2007.

[26] Tomohisa Wada, Suresh Rajan, and Steven A Przybylski. An analytical access time model for on-chip cache memories. *Solid-State Circuits, IEEE Journal of*, 27(8):1147–1156, 1992.

[27] Wikipedia. Graphics processing unit. `http://en.wikipedia.org/wiki/Graphics_processing_unit`, 2012. Accessed: 2014-04.

[28] Steven JE Wilton and Norman P Jouppi. Cacti: An enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, 1996.

[29] Wei Wu, Lingling Jin, Jun Yang, Pu Liu, and Sheldon X-D Tan. A systematic method for functional unit power estimation in microprocessors. In *Proceedings of the 43rd annual Design Automation Conference*, pages 554–557. ACM, 2006.

# VITA

## MD MAINUL HASSAN

### EDUCATION

- M.Sc., Computer and Information Science,

  University of Mississippi, to be awarded May 2015.

  Thesis: Power and HotSpot Modeling for Modern GPUs.

- B.Sc., Computer Science and Engineering,

  Bangladesh University of Engineering and Technology, October 2009.

### RESEARCH EXPERIENCE

- Research Assistant, Aug 2013 - May 2015

  Heterogeneous Systems Research (HEROES) Lab,

  University of Mississippi.

### HONORS and FELLOWSHIPS

- Upsilon Pi Epsilon Honor Society, 2015

  University of Mississippi Gamma Chapter.

### PUBLICATIONS and PRESENTATIONS

- A Low Power and High Performance Face Detection on Mobile GPU. International Conference on Energy Aware Computing Systems & Applications, 2015.

### JOB EXPERIENCE

- Software Engineer (Level 6), starting from Jun 2015

  IMS Health (USA) Ltd., Seattle, WA.

- Senior Software Engineer, Mar 2010 - Aug 2013

  IMS Health (BD) Ltd., Dhaka, Bangladesh.

- Member, R&D, Jan 2010 - Mar 2010

  Commlink Infotech Ltd., Dhaka, Bangladesh.

- Junior Assistant Vice President, IT, Oct 2009 - Jan 2010

  Delta Life Insurance Company Ltd., Dhaka, Bangladesh.