# MULTITHREAD SCHEDULING, SYNCHRONIZATION AND POWER ANALYSIS ON GENERAL-PURPOSE GRAPHICS PROCESSING UNIT

By

JIANMIN CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2012

To my family

ACKNOWLEDGMENTS

First of all, I would like to sincerely thank my PhD advisor Dr. Jih-kwon Peir for his guidance, patience and criticisms, from which I learned not only a lot of researching skills but more importantly, good researching personality. I would also like to thank my other PhD committee members: Dr. Sartaj Sahni, Dr. Jorg Peters, Dr. Ye Xia, Dr. Manuel E. Bermudez and Dr. Lei Zhou for their valuable comments and advices.

I also appreciate the help and friendship from my lab mates and friends: Zhen Yang, Feiqi Su, Chung-Ching Peng, Xudong Shi, David Lin, Zhuo Huang, Gang Liu and Xiao Li, who have made the PhD life much easier and more joyful. I am also thankful to the system staff who maintained the servers where I run most of the simulations on and also the administrative staff including Keri Taylor, Joan Crisman and John Bowers, who are always willing to help.

Above all, none of my achievements would have been possible without the love and support from my family: My parents, grandparents, aunts and uncles. Special thanks go out to my wife Xiaoyuan Li, whose love and company is the most important support to this work. This dissertation is dedicated to them.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS ..... 4

LIST OF TABLES ..... 7

LIST OF FIGURES ..... 8

ABSTRACT ..... 10

CHAPTER

1 BACKGROUND INFORMATION ..... 12

1.1 Nvidia GPU Architecture ..... 15
1.2 Issues and Motivations ..... 18
1.2.1 Host Synchronization and Partial Differential Equation Solver ..... 19
1.2.2 Power Consumption and its Modeling ..... 20
1.2.3 Guided GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency ..... 21
1.3 Performance Evaluation Methodology ..... 23

2 EXPLOITING ASYNCHRONOUS ITERATIVE METHODS ON MANY-CORE GPUS ..... 25

2.1 Introduction ..... 25
2.1.1 Synchronization Overhead ..... 27
2.1.2 Applications Using Iterative Method ..... 30
2.1.2.1 Poisson image editing ..... 31
2.1.2.2 3-D shape from shading ..... 33
2.2 Related Work ..... 34
2.3 Parallelization of Applications ..... 36
2.3.1 Asynchronous Data Communication ..... 37
2.3.2 Choices of Parallelization Parameters ..... 41
2.4 Performance Evaluation ..... 44
2.4.1 Execution Time and Convergence on C1060 ..... 45
2.4.2 Experiment with GTX580 ..... 48
2.4.3 Sensitivity Study ..... 51
2.5 Summary ..... 54

3 STATISTICAL GPU POWER ANALYSIS USING TREE-BASED METHODS ..... 56

3.1 Introduction ..... 56

3.2 Related Work ..... 59
3.3 GPU Power Modeling ..... 61
3.3.1 Characteristics of Kernel Execution and Profiling ..... 61
3.3.2 Sample Workloads and Power Measurement ..... 64
3.3.3 Statistical Modeling ..... 66
3.4 Power Prediction and Understanding ..... 68
3.5 Correlation Analysis ..... 70
3.6 Summary ..... 75

4 GUIDED GPU SCHEDULING: UTILIZING MULTI-THREAD PARALLELISM TO HIDE MEMORY LATENCY ..... 76

4.1. Introduction ..... 76
4.2. Related Work ..... 81
4.3. SM Stall Cycles Using Traditional Round-Robin Scheduling ..... 83
4.4. Program-Directed Warp Scheduler ..... 85
4.5 GPGPU Timing Simulator ..... 91
4.5.1 Ocelot Tracer ..... 94
4.5.2 The Scoreboard and Scheduler ..... 97
4.5.3 The Instruction Simulator ..... 98
4.5.4 The Memory Hierarchy Simulator ..... 98
4.6 Performance Evaluation ..... 100
4.6.1 Execution and Stall Cycles ..... 101
4.6.2 Sensitivity Studies ..... 104
4.7 Summary ..... 107

5 DISSERTATION CONCLUSIONS ..... 108

LIST OF REFERENCES ..... 110

BIOGRAPHICAL SKETCH ..... 116

LIST OF TABLES


| Table | | page |
|---|---|---|
| 1-1 | Resource constraints of Geforce8800, GTX280 and GTX580 | 18 |
| 2-1 | Computation threads VS communication threads | 42 |
| 3-1 | Runtime characteristics from GPGPUSim | 62 |
| 3-2 | List of kernels | 65 |
| 3-3 | Average PE and Mean Squared Error (MSE) of the three methods | 69 |
| 4-1 | PTX instruction category, initiation and execution cycles | 95 |
| 4-2 | Memory hierarchy and configurations | 99 |
| 4-3 | Characteristics of the 12 kernels from CUDA SDK and Rodinia | 100 |

LIST OF FIGURES

Figure page


1-1 Three levels of CUDA program hierarchy ..... 17

1-2 GPU architecture and CTAs assignment ..... 17

2-1 Pseudo-CUDA BFS kernel ..... 29

2-2 BFS execution time with/without host synchronization ..... 30

2-3 Inserting objects with complex outlines into a new background: cloning on the left, seamless cloning with Poisson Image Editing on the right. ..... 31

2-4 An example of shape from shading. An image of a vase is shown on the left and the recovered 3-D surface is shown on the right. The height field of the computed 3-D surface is obtained by integrating the normal vector field estimated from the image ..... 33

2-5 Data exchange between adjacent CTAs ..... 37

2-6 Stripe vs. square CTA scheduling (CTAs with the same filling are executed concurrently) ..... 43

2-7 Execution time and Number of Iterations of Poisson using eight communication/scheduling schemes ..... 46

2-8 Execution time and number of iterations of 3D-Shape using eight communication/scheduling schemes ..... 46

2-9 Comparison of *Base* and the best Execution time and number of iterations on both GPUs ..... 49

2-10 GTX 580 results with and without L1 cache ..... 50

2-11 Execution time, number of iterations and time per iteration of 3D-Shape Intervals ..... 52

2-12 Execution time of 3D-Shape with different number of communication threads ..... 52

2-13 Execution time and number of iterations of 3D-Shape with random scheduling and communication ..... 54

3-1 Regression tree for 52 kernels based on the parameters from *GPGPUSim* ..... 67

3-2 Comparison of predicted and measured power ..... 70

3-3 relative variable importance ..... 71

3-4 Partial dependence plots for five most influential and four architecture related parameters. The rug plots on inside bottom of plots show distribution of kernels across that variable, in deciles ..... 72

3-5 Proximity plot for the random forest model ..... 73

3-6 Normalized variables of selected kernels ..... 74

4-1 The baseline GPGPU architecture ..... 83

4-2 Stall cycle using *fine-RR* and *greedy-RR* ..... 85

4-3 Simplified CUDA and PTX codes of convolutionSep ..... 86

4-4 Comparison of three scheduling algorithms: *oldest-first, simple-p,* and *region-p* ..... 89

4-5 Normalized total cycles of 12 kernels ..... 101

4-6 Normalized stall cycles of 12 kernels ..... 101

4-7 Stall cycles with different number of CTAs per SM ..... 104

4-8 Stall cycles with different global memory latency ..... 105

4-9 Normalized stall cycles with different scheduler overhead ..... 106

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

# MULTITHREAD SCHEDULING, SYNCHRONIZATION AND POWER ANALYSIS ON GENERAL-PURPOSE GRAPHICS PROCESSING UNIT

By

Jianmin Chen

May 2012

Chair: Jih-Kwon Peir
Major: Computer Engineering

Due to excessive power consumptions, limited instruction-level parallelism, escalating processor-memory wall, and available thread-level parallelism in many workloads, computer industry has moved away from building expensive single processor chip with limited performance improvement to multi-core chip for higher chip-level IPCs (Instructions Per Cycle) within an acceptable power budget. Instead of replicating general-purpose Central processing unit (CPU), the recent introduction of Nvidia's and ATI's Graphics Processing Units (GPUs) took a different approach by building many-core GPU as co-processors to connect through a Peripheral Component Interconnect Express (PCI-Express) bus to the host CPU. With hundreds of processing cores and high bandwidth memory systems, the GPU achieves much higher performance than conventional multi-core CPUs.

However, GPU also faces many challenges. First, the inability to perform global synchronization among parallel execution thread blocks on GPU forces the parallel applications to synchronize through the host and incur significant overheads. Second, the fair round-robin scheduling scheme in current GPUs often wastes thread-level

parallelism due to disparate instruction and memory latencies and encounters heavy stall for long-latency global memory accesses. Third, GPUs consume much higher power than CPUs. The current GPU research community needs tools for simulating and predicting GPU power consumption and to understand the source of power consumption in order to help designing and using future low-power GPUs.

Facing these challenges, this dissertation makes the following important contributions. First, it proposes a loosely-synchronized execution model for an important class of applications which use the iterative method to solve Partial Differential Equation (PDE). The proposed model reduces the frequency of host synchronization to alleviate the global synchronization overhead for better performance. Second, it proposes a guided warp scheduling scheme that allows flexible round-robin scheduling distance and uses software guided priority shift among parallel threads to maintain fairness. This priority-based scheduling algorithm enables more overlapping between computation instructions and global memory accesses to reduce scheduling stall cycles. Third, it proposes a sophisticated statistical power model that can not only predict GPU power consumption accurately but also help understanding the power consumption insights including the correlation and partial dependence between various execution parameters and power consumption. Fourth, to facilitate our research, we develop a cycle-based GPU simulator which is flexible and can be used as the framework for other GPU architecture and performance studies.

# CHAPTER 1
# BACKGROUND INFORMATION

Due to excessive power consumptions, limited instruction level parallelism, and escalating processor-memory wall, computer industry has moved away from building expensive single processor chip with limited performance improvement to multi-core chip for higher chip-level IPCs (Instructions Per Cycle) within an acceptable power budget. Instead of replicating general-purpose CPUs in a single chip, the recent introduction of Nvidia's and ATI's GPUs [46], [47], [48], [3] took a different approach by building many-core GPU chip as co-processors to be connected through a PCI-Express bus to the host CPU. The host executes the source main program and initiates computation *kernels*, each with multiple *Cooperative Thread Arrays* (CTAs) (also called *thread blocks*) to be executed in parallel on the attached GPU.

Furthermore, with the emergence of extreme-scale computing, modern graphics processing units (GPUs) have been used to build powerful supercomputers and data centers. For example, in the Top500 list [62] released in June 2011, the world's second fastest supercomputer Tianhe-1A installed in China employs 7168 Nvidia Tesla M2050 general purpose GPUs [48]. LOEWE-CSC, which is located in Germany and ranked at 22$^{nd}$ in the Top500 list, includes 768 ATI Radeon HD 5870 GPUs for parallel computations [3].

In this thesis, we investigate several architecture, performance and power related issues and propose solutions on modern GPUs. First, due to the hardware complexity and uncertainty of mapping software CTAs on hardware, *precise* data

communication and synchronization among threads in different CTAs in a kernel is not permitted on GPU. CTA-level synchronization such as *barrier* must go through the host which is costly and cause performance degradation when fine-granularity of synchronization among parallel CTAs is needed. In order to avoid this overhead, we exploit asynchronous execution among parallel CTAs which is suitable for an important class of applications using iterative method for solving partial differential equations. Our initial study results show that the proposed method can improve 4-5 times of performance compared with the existing host synchronization implementation.

Second, CTA and warp scheduling on many-core GPU is essential to the kernel performance. Existing fairness *round-robin* and *greedy* based scheduling [21] effectively march the warp execution in a synchronous manner and encounter serious stall on the region involving heavy global memory accesses. Such fairness scheduling also causes unbalanced loads at the kernel ending phase. We proposed a scheduler that breaks the harmful fairness and reduces more than 60% of the stall cycles. It can help to balance the CTA assignments in the kernel ending phase. Instead of round-robin, the proposed scheduling method gives priority to the CTA which is closer to the completion to create overlaps between computation and long-latency global memory accesses among parallel CTAs.

Third, although delivering high computation performance, the GPU consumes high power and needs to be equipped with sufficient power supplies and cooling

systems. Unfortunately, existing GPU power measurement tools are very limited. Unlike in studying CPU power, the GPU community lacks needed accurate and flexible tools for power simulation. In this research, we study GPU power models with statistical tools including linear regression trees [8] and random forest methods[7]. Our results show that the random forest method not only delivers accurate power prediction, it also provides insightful analysis between the power consumption and the independent performance variables in comparison with simple regression analysis.

In the remaining of this dissertation, we will give detailed descriptions of the problems, research motivations, proposed solutions, and present some preliminary performance study results for dealing with these essential issues. In an Nvidia GPU chip, multiple streaming processors (*SPs*, or *cores*) are grouped into a few streaming multiprocessors (or SMs) as a hardware scheduling unit. Based on the resource requirement, one or more CTAs can be scheduled on an SM. Each CTA contains one or more 32-thread warps to be executed on multiple SP cores in a Single-Instruction-Multiple-Threads (SIMT) fashion for achieving high floating-point operations per second. Fermi is the latest generation of CUDA-capable GPU architecture introduced by Nvidia [47]. Derived from prior families such as G80 and GT200, the Fermi architecture is enhanced to satisfy the requirements of large-scale computing problem. The GeForce GTX 580 used in this thesis is a Fermi-generation GPU [46].

### 1.1 Nvidia GPU Architecture

In the GTX580 architecture, there are 16 SMs, each with 32 SPs for a total of 512 CUDA cores. Each SM has local shared memory and L1 caches. GTX580 is also equipped with a shared L2 cache for all SMs, six DRAM channels for accessing global memory, a GPU-host interface for transferring data to/from the host, and a Gigathread scheduler for scheduling CTAs on SMs.

There are 16 CUDA cores in each SM. Within a CUDA core, there are a fully pipelined integer ALU and a fully-pipelined floating point unit (FPU). In addition to these regular processor cores, each SM is also equipped with four special function units (SFU) which are capable of executing transcendental operations such as sine, cosine, and square root. Furthermore, each SM has 16 load/store units, which calculate source and destination addresses for sixteen threads per clock. Each SM also has two warp schedulers and two dispatch units to select one warp per cycle to be executed on the cores, the load/store units, or the SFU.

The GPU has multiple levels of memory hierarchies. The *global* memory (also called *device* memory) located off-chip with long access latency can be accessed by all CTAs in a grid. The on-chip per-SM *shared* memory with fast access latency is accessible from all threads within a CTA. An efficient usage of the fast but size-limited (16/48KB) shared memory is critical in reducing expensive global memory accesses. Data that is local to a thread or shared among threads in the same CTA can be moved into the shared memory to exploit reference locality for low latency

accesses. Each SM has 8/16/32K on-chip registers to be used by all CTAs. The off-chip *local* memory can only be accessed by a single thread. The GPU also has off-chip *constant* and *texture* memories for the read-only data, which will be cached in its corresponding caches for faster accesses. Per SM L1 and a unified L2 caches shared by all 16 SMs are first introduced in Fermi generation. All off-chip memory accesses will go through L2 including memory transfer between GPU and CPU. L1 cache can be configured as 16KB, 48KB or disabled as needed. When a kernel is invoked from the host CPU, the needed data must be copied from the host main memory to the device's global memory before the start of executions.

To program the GPU, Nvidia's CUDA extends ANSI C with a few new program constructs and key words [50]. The programmers can use these CUDA extensions to parallelize the programs into three levels as shown in figure 1-1. In the highest level, a *kernel* can be invoked from the host CPU to create a single *grid* to run on the GPU. To better exploit software/hardware parallelism, parallel kernel invocations are allowed on the same or multiple GPUs [43]. Each grid has multiple *CTAs*, which can be specified as a three-dimensional array. Each CTA consists of multiple *threads* also in a three-dimension format. A CTA is scheduled to run on one SM independently with other CTAs. Within each CTA, parallel threads are further grouped into 32-thread *warps*, which is the smallest unit to be scheduled on the hardware units in an SM.

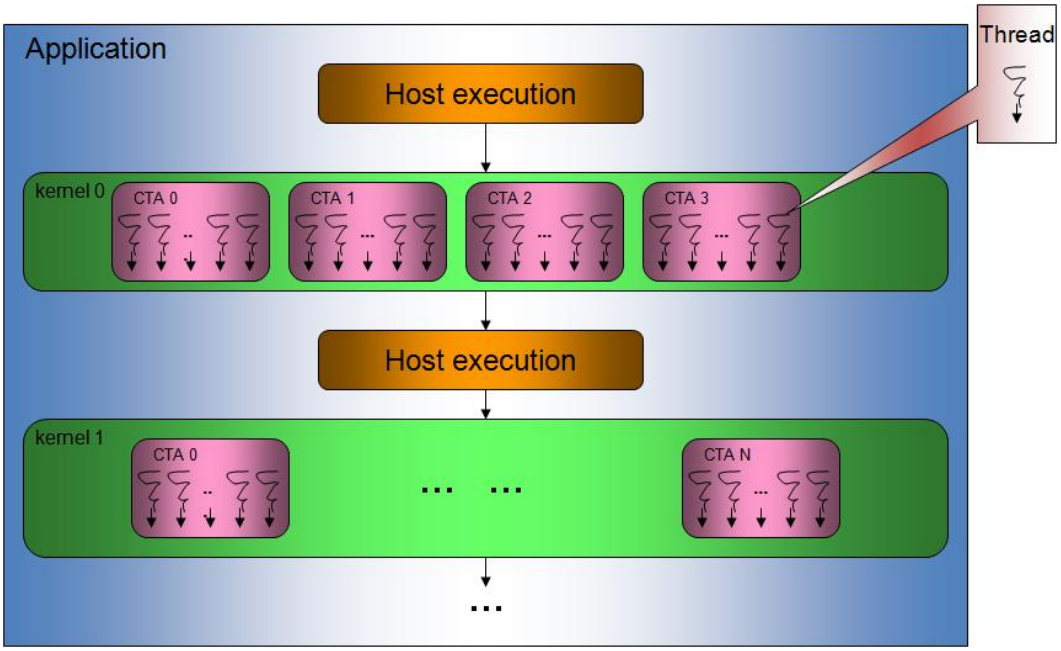


Figure 1-1. Three levels of CUDA program hierarchy

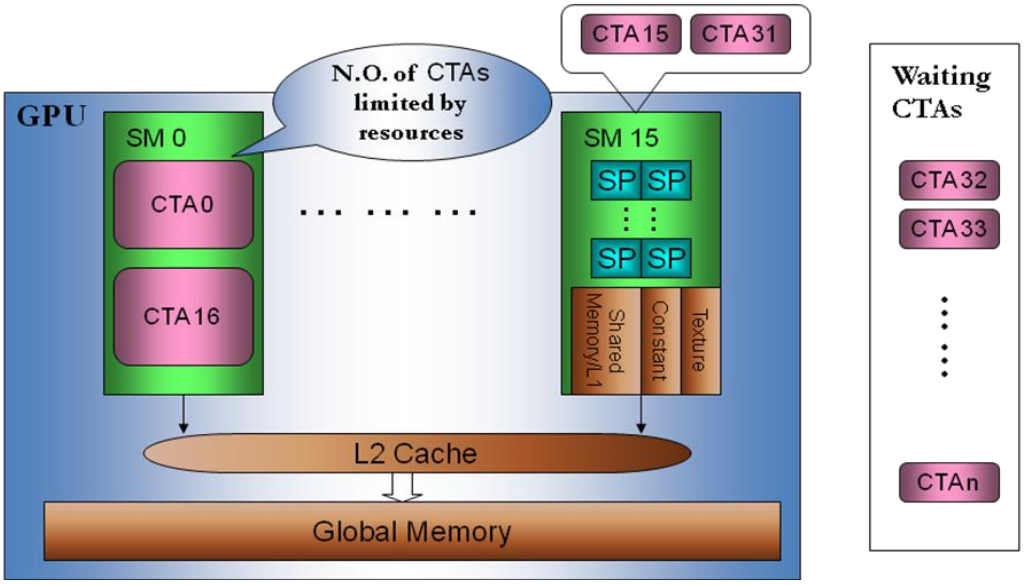


Figure 1-2. GPU architecture and CTAs assignment

Figure 1-2 shows an example of how CTAs are assigned to SMs. For a kernel with n CTAs, the SM can only execute 2 CTAs due to resource constrains on each SM. Then, the first 32 CTAs are assigned to the 16 SMs and the rest of the CTAs will be assigned when any of the executing CTA finishes and release the hardware

resources. Nvidia's GPUs and CUDA present many constraints in creating and scheduling parallel threads on multiple SMs as shown in Table 1-1 with Geforce8800, GTX280 and GTX580 as three examples for the three generations. It relies on the programmers to optimize the CUDA code for efficiently utilizing the underline hardware. The basic strategy is to fully populate all SMs with simultaneous warps from CTAs to achieve the best hardware utilization. In addition, it is important to minimize global memory accesses by exploiting data reference locality using the fast registers, caches, and shared memory.

Table 1-1. Resource constraints of Geforce8800, GTX280 and GTX580

| Resource | Geforce8800 | GTX280 | GTX580 |
|---|---|---|---|
| Threads per SM | 768 | 1024 | 1536 |
| CTAs per SM | 8 | 8 | 8 |
| Threads per Warp | 32 | 32 | 32 |
| Warps per SM | 24 | 32 | 48 |
| Threads per CTA | 512 | 512 | 1024 |
| Registers (32 bits) per SM | 8192 | 16384 | 32768 |
| Shared Memory per SM | 16KB | 16KB | 16KB/48KB |

## 1.2 Issues and Motivations

As we introduced above, although GPUs has abundant computing power and memory bandwidth, it faces several constraints and challenges. First, the host synchronization overhead can severely affect the performance of iterative kernels like PDE solvers. The second challenge is a proper methodology and tool in understanding and measuring GPU's power consumption, which becomes the most important concerns in chip design. The third challenge is from the performance of the current CTA and warp schedulers, which can help with the fairness among all the

warps but cause many avoidable stall cycles in kernel execution. We will introduce them in more details in the following sections.

### 1.2.1 Host Synchronization and Partial Differential Equation Solver

Despite the accessibility of the global memory from all CTAs, there are inherent restrictions for data communication and synchronization among parallel CTAs on GPU. This is due to the fact that a CTA, once scheduled to run on a SM, the resource will not be released until the execution has completed. Therefore, a deadlock is encountered if the current active CTAs depend on the execution of another CTA which had not been scheduled. As a result, *precise* data communication and synchronization among threads in different CTAs is not permitted. To meet the precise synchronization, the kernel must exit back to the host. Afterwards, the host may invoke subsequent kernels to continue the program execution. Hence, frequent host synchronization is necessary for fine-granularity of data communication and synchronization among parallel CTAs in large-scale parallel programs.

The host synchronization incurs two noticeable overheads. First, the lifespan of shared variables in the shared memory is within a single kernel invocation. Therefore, the data in the shared memory must be saved in the global memory for future reuses before the kernel exits. The saved data is likely to be restored back to the shared memory for the next kernel invocation. Second, besides the overhead involved in

kernel initiation, intermediate results may be transferred back to the host for controlling the synchronization function.

In our first research work, we investigate a class of applications which exhibits high parallelism but requires frequent data communication among parallel threads. However, a unique feature is that the precise order of data communication among parallel threads is not required. A typical example involves a partial differential equation (PDE) solver based on chaotic relaxation [11] which uses an iterative algorithm until the solution converges. Although a sequential order of the data communication may speed up the convergence, a *relax execution ordering* enables parallel CTAs to be executed independently without precise data communication and greatly reduces the synchronization overhead. There are concerns about the PDE solver's accuracy that may be critical for some applications. However, in many applications like image smoothing, editing, mesh smoothing, and image imprinting, the actual proofs of convergence are often not important but empirical observances of convergence are.

### 1.2.2 Power Consumption and its Modeling

Although delivering high computation performance, the GPU consumes high power and thus needs to be equipped with sufficient power supplies and cooling systems. For example, the GTX580 based on the new Fermi architecture [47] consists of 512 SPs grouped into 32 SMs. The Fermi GPU can deliver 500+ gigaflops of IEEE standard double-precision floating point operations and over one

teraflop of single-precision peak performance. However, GPUs also consume a lot of power. For example, the high-end Intel i7 CPU chip consumes about 55W, while a high-end GPU GTX590 board consumes up to 360W power. The Appro1246G4 server which features four Tesla 20-series GPUs, two Quad/Six-Core Intel Xeon Processors can consume up to 1400W [2]. Therefore, it is essential to institute an efficient mechanism for evaluating and understanding the power consumption and dissipation requirements when running real applications in these high-end GPUs.

In this research work, we investigate high-level GPU power consumption models using a linear regression tree [8] and sophisticated random forest methods [7]. The random forest method provides the correlation analysis between the power consumption and the independent performance variables so we can better understand GPU's power characteristics. These models are established based on the execution frequencies of instructions and memory accesses in a wide-variety of application samples. To complement the execution frequency profile, we include several performance-sensitive architectural metrics into the analysis for accurate power modeling.

### 1.2.3 Guided GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency

Each SM in a GPU executes multiple 32-thread warps from all active CTAs. In each cycle, the warp scheduler needs to schedule one ready-to-execute instruction from a warp among all active warps if available. There are two common scheduling algorithms: Round Robin (RR) and Greedy. For a RR scheduler, the ready-to-

execute instruction is selected from a warp in each cycle in a round-robin fashion. The greedy scheduler, on the other hand, continues schedules one ready-to-execute instruction per cycle from the same selected warp until there is no more ready instruction. In this case, the scheduler moves to the next warp also in a round-robin fashion.

These two schedulers, especially the RR scheduler, will result in a situation where all warps proceed in a similar speed. Although this can help to keep as many concurrently active warps as possible and maintain their execution fairness, it may result in more stall cycles where no warp can proceed due to various reasons like global memory accesses or barriers.

In this research work, we investigate the guided GPU scheduling algorithms to greatly reduce the stall cycles and to help balance the CTA assignment. The basic idea is straight-forward, instead of selecting the read-to-execute instruction from warps in a fine round-robin fashion, the proposed scheduler assigns priority to different warps and always try to schedule the ready instruction from the warp with the highest priority which is the warp with the smallest warp ID. It also move the priority among CTAs through the SM in turns. In other words, the prioritized scheduler try to stagger the execution of different warps/CTAs for better performance while keep the fairness by switching the priority.

### 1.3 Performance Evaluation Methodology

In order to evaluate the performance, several tools and methodologies are used in our research.

**Profiler**: CUDA computing profiler [44] can collect timing information about kernel execution and different memory transfer operations in one SM. The profiler helps programmers to identify performance bottlenecks in kernels.

**GPGPUsim**: GPGPUsim [1] is a cycle-level simulator that simulates the PTX instruction set running on Nvidia GPUs. The PTX resembles the real ISA of Nvidia GPU. But the existing tool cannot support PTX 2.0 and Fermi generation architecture.

**Power meter**: The energy consumptions of kernels are measured using a YOKOGAWA WT210 Digital Power Meter [67] as statistical power samples for power consumption prediction and analysis.

**Trace-generating Ocelot**: Ocelot [19] is a framework for heterogeneous system, providing various backend targets for CUDA kernels and analysis modules for the PTX instruction set. It has a functional simulator that is compatible with the newest PTX version so it can be used to generate instruction and memory traces for further architecture research.

**Cycle-accurate GPU Simulator**: We implemented the trace-driven based cycle level simulator for simulating the latest Fermi architecture and PTX2.0. It is driven by the trace from Ocelot and it includes major components including

Warp/CTA scheduler, Instruction execution module and Memory hierarchy module, etc.

**Benchmark**: To cover a wide variety of GPU application kernels, we selected kernels from *CUDA SDK* [49] , *Rodinia* Benchmark [12] and *Parboil* Benchmark [56]. We also collect kernels from a computer graphics application *Poisson Image Editing* [14] which seamlessly integrates new image contents into a given background image, and a computer vision application *3D Shape from Shading* [14] which deals with recovering the 3-D structure of an object from its image.

# CHARPTER 2
# EXPLOITING ASYNCHRONOUS ITERATIVE METHODS ON MANY-CORE GPUS

As described in Chapter 1.1, we investigated an important class of applications using chaotic partial differential equation (PDE) solver on GPU and proposes a scheme to improve their performance even with the constraint about the threads communication and synchronization. Chapter 2 is organized as follows. We will first introduce the detailed communication and synchronization overhead and also the applications we will use in 2.1. Related work will be given in 2.2 followed by the details in parallelizing the applications using CUDA in 2.3. The preliminary performance results are described in 2.4 followed by ongoing work in 2.5. 2.6 summarizes Chapter 2.

## 2.1 Introduction

*Precise* data communication and synchronization among threads in different CTAs within a grid is not permitted. Instead, a barrier synchronization is only at a lower level among the threads in the same CTA using *syncthreads()*. To meet the synchronization requirement among multiple CTAs, the only way is to exit back to host and invoke a kernel afterwards. Therefore, frequent host synchronization is necessary for large-scale parallel programs executed by multiple CTAs when fine-granularity of data communication and synchronization is required among the parallel CTAs. The Faster Global Barrier in [64] sets *flags* in the global memory, which are accessible from all CTAs. By setting/testing the flags from multiple CTAs,

a global barrier can be accomplished without going back to host. However, given the fact that a CTA must complete the execution before releasing the hardware resources, this scheme can only work when all CTAs are scheduled and executed simultaneously. Many CUDA applications with high parallelism, including those used in this work, invoke more CTAs than what the hardware can execute simultaneously.

As described in Chapter 1.1, the host synchronization incurs two noticeable overheads. First, this will result in the waste of the data previously loaded into shared memory by a CTA that can be potentially used by another CTA. Second, We will have the overhead of kernel initiation and possibly the intermediate results transfer between GPU and CPU. However, a unique feature in many iterative applications is that the precise order of data communication among parallel threads is not required.

In order to accomplish tight data communication among the CTAs, we use an *asynchronous* data exchange mechanism through the global memory [63]. Each CTA periodically sends newly computed data to the global memory in exchange for the needed data produced by other CTAs. In this work, we exploit this inter CTA communication mechanism to speed up the iterative algorithm. In handling the data exchanges, two different types of threads, *computation* and *communication* are created in each CTA. The computation threads execute the normal computation functions while the communication threads are dedicated for storing and fetching the data to and from the global memory. It is important to note that this data communication through the global memory can be performed independently among

CTAs. There is no particular order in exchanging the data and hence it is referred to as *asynchronous* data communication. The computation and communication threads can be overlapped to hide the communication overhead. The frequency of data exchanges is based on the ability to overlap the computation and communication threads as well as the balance between the global memory traffic and the convergence speed.

Performance evaluations using a Poisson Image Editing and 3-D Shape from Shading show that the *asynchronous* data communication is effective in reducing host synchronization with minimal impact on the convergence of the solution. What is more, the tuning of this data communication can significantly improve the performance. By overlapping the communication and computation with proper tuning of the CTA scheduling, we observe that a speedup of 4-5 times is achievable for both applications in comparison with the solution with fine granularity of host synchronization on Tesla C1060 and Fermi generation GTX580.

### 2.1.1 Synchronization Overhead

In order to study the performance impact of the data communication and synchronization requirement among parallel CTAs, in parallelizing the applications, we isolate the data communication and synchronization requirement from other domain of program optimizations on GPUs. We first try our “best effort” to optimize the register and shared memory usage per CTA to be fitted into the available resources in each SM. We also consider the efficiency of accessing the global

memory by arranging stride-1 accesses among the threads in a warp. Meanwhile, we consider creating enough schedulable warps from CTAs to populate the SMs for keeping the SP pipeline busy. Based on the hand-optimized code, we can then evaluate different data communication and synchronization mechanisms for the selected applications. The detailed descriptions of the two selected applications and their parallelization will be given in the following sections.

Next, in order to demonstrate the performance impact of the host synchronization, we experiment with a parallel Breadth First Search (BFS) algorithm [24]. The BFS problem is to find the minimum number of edges needed to reach every vertex in a graph *G* from a source vertex *s*. The parallel BFS starts from *s* by assigning an edge count of 1 to all its neighboring vertexes. Afterwards, the previously found neighbors become the source vertexes for searching their immediate neighbors whose edge counts are equal to 2. This process continues until all vertexes in *G* are either visited or unreachable from *s*. Each of these traversal steps is referred as a *level* and is assigned a value that is equal to the number of edges measured from the original source *s*. During the traversal, a vertex, once visited, is dropped from further considerations.

In the experiment, we limit the parallelization to a single CTA with 512 threads so that the host synchronization can be replaced by *syncthreads()* within a CTA as a barrier for separating the levels of searches. Furthermore, we experiment with a relatively small graph with 3K vertexes in order to fit a small read-only vertex data

array in the shared memory to include its reloading overhead in each kernel invocation. During the traversal, two barrier synchronizations are needed at each traversal level. The total amount of synchronizations depends on the connectivity of the graph.

```
tid = getthreadID; Finish = false;
while (NOT Finish) do
    Finish = true;
    for all nodes vid calculated by thread tid do
        if (vid is marked as ”current level”) then
            mark all unvisited neighbors of vid as ”next level” and set their edge counts;
        end if
    end for
__syncthread();
for all nodes vid calculated by thread tid do
    if (vid is marked as ”next level”) then
        change its mark to ”current level”; mark itself visited; Finish = false;
    end if
end for
__syncthread();
end while
```

Figure 2-1. Pseudo-CUDA BFS kernel

In this experiment, multiple graphs are generated using the graph generation tool from [4] with different edge/vertex ratios. Since the number of vertexes is fixed, the traversal levels decrease with the number of edges. However, graphs with more edges increase the time used in searching for the neighboring vertexes; hence they require longer execution time. The pseudo-CUDA kernel program without host synchronization is given in Figure 2-1. In Figure 2-2, we compare the execution time with and without the host synchronization. The horizontal axis represents the total

number of traversal levels, hence the synchronization frequency for the respective graphs. As expected, we observe that the synchronization through the host, even with a small number of barrier synchronizations, degrades the performance significantly. The execution time gap between host and no-host synchronization increases with the frequency of synchronizations. Host synchronization is needed with more than one CTA.

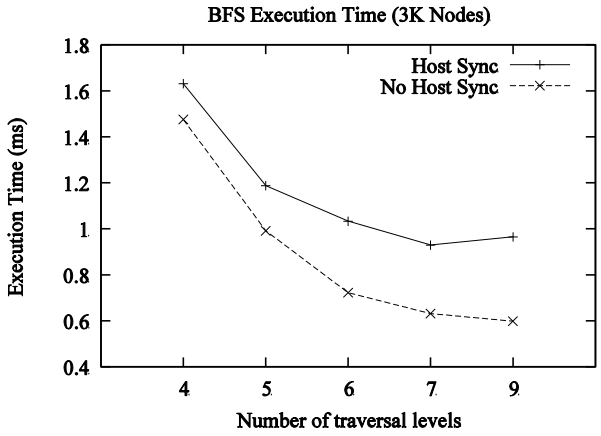


Figure 2-2. BFS execution time with/without host synchronization

### 2.1.2 Applications Using Iterative Method

Iterative methods are frequently used to obtain numerical solutions for partial differential equations (PDE) arising from a variety of engineering and scientific problems. A fast and accurate PDE solver can be valuable for designing computational solutions to a wide range of applications and problems. In this work, we investigate two applications that require a high-performance PDE solver on grids: *Poisson image editing* from computer graphics and *shape from shading* from computer vision. In both applications, a crucial component is a module that solves a specific PDE on a grid, and the performances of the two algorithms depend critically

on the ability of the solver to efficiently and accurately compute the numerical solutions.

#### 2.1.2.1 Poisson image editing

Image editing is a computer graphics application that has become increasingly common in our everyday life, especially with the advent of inexpensive digital cameras. Adobe's Photoshop is perhaps the most popular image editing software that has been used widely for editing, modifying and polishing images. A useful image editing function is the ability to seamlessly integrate new image contents into a given background image. Figure 2-3 shows an interesting example in which new image patches are introduced into the background image of a swimming pool [53].


Figure 2-3. Inserting objects with complex outlines into a new background: cloning on the left, seamless cloning with Poisson Image Editing on the right. [Reprinted with permission from Patrick Perez, 2003. Poisson image editing. Association of Computing Machinery (ACM) Transactions on Graphics (Page 316, Figure 3). ACM, New York, New York]

As the direct cloning of the intensity, values produces unacceptable results with visible seams at the patch boundaries that significantly degrades the realism of the image, the challenge here is to develop an efficient and accurate algorithm that can automatically integrate different image contents to produce a realistic-looking

composite image. The method proposed in [53] has become popular due to its conceptual simplicity and ease of implementation. By cloning the image gradients directly, the algorithm achieves seamless integration of different image patches by solving a Poisson equation on a grid with Dirichlet boundary condition. If ***I*** denotes the intensity values of a given patch in the final composite image, [53] proposes to estimate ***I*** by solving the following variational problem:

$$\min_{\boldsymbol{I}} \iint_{\Omega} \| \nabla \boldsymbol{I} - v \|^2 \quad with\, \boldsymbol{I}\,|_{\partial\Omega} = \boldsymbol{I}^*\,|_{\partial\Omega} \qquad (1)$$

where *v* denotes the image gradients of the new patch and ***I****** is the background image. The corresponding Euler-Lagrange equation is the Poisson equation with Dirichlet boundary condition:

$$\triangle \boldsymbol{I} = \mathbf{div}\ v, \quad \boldsymbol{I}\,|_{\Omega} = g \qquad (2)$$

This equation can be solved on the image grid by first converting it into a linear system of equations using a finite difference scheme that numerically approximates the partial derivatives. The linear system can then be solved using iterative method such as Gauss-Seidel that iteratively updates the value at each grid point using the values at its neighbors. While Gauss-Seidel is a well-established technique for solving PDEs numerically, its convergence rate can be slow and may require many iterations before stopping, particularly for images with large number of pixels. From a practical viewpoint, a fast numerical Poisson equation solver is imperative for developing an interactive image editing software.

#### 2.1.2.2 3-D shape from shading

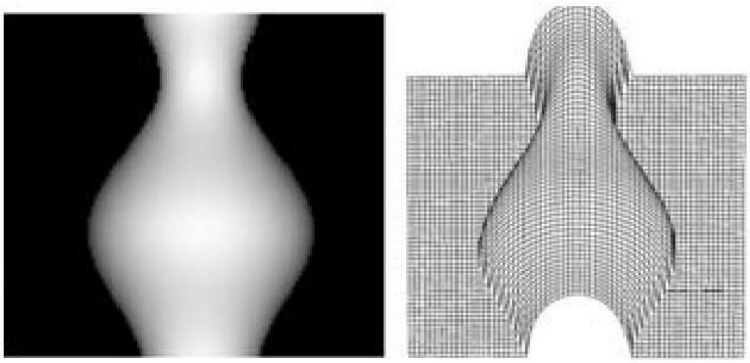

Figure 2-4. An example of shape from shading. An image of a vase is shown on the left and the recovered 3-D surface is shown on the right. The height field of the computed 3-D surface is obtained by integrating the normal vector field estimated from the image

The second application is a classical problem from computer vision that deals with recovering the 3-D structure of an object from its image. The ability to accurately recover the shape is crucial for many computer vision applications such as object recognition, object detection and other AI applications such as human face recognition, human-computer interaction (HCI) and virtual reality. Many techniques for recovering an object 3-D shape have been developed and a good portion of these algorithms require the solution of partial differential equations. The basic idea underlying shape from shading [27] is to recover the normal vector field of the observed surface from image intensity values directly using a known shading model, which provides a formula that relates the normal vectors and external illuminations to the observed intensities. Modeling the 3-D surface as the graph of a height function ***H(x, y)*** defined over a grid, the normal vector at a point is a function of the partial

derivatives of ***H***, *f, g*. The normal vector field is recovered by minimizing the energy functional where *Rs(f, g)* denotes the predicted intensity value.

$$\mathcal{E}(f,g) = \iint (\boldsymbol{I}(x,y) - R_s(f,g))^2\,dx\,dy + \lambda \iint (f_x^2 + f_y^2 + g_x^2 + g_y^2)\,dx\,dy \quad (3)$$

$$\nabla^2 f = -\lambda(E(x,y) - R_s(f,g))\frac{\partial R_s}{\partial f}$$
$$\nabla^2 g = -\lambda(E(x,y) - R_s(f,g))\frac{\partial R_s}{\partial g} \quad (4)$$

The Euler-Lagrange equation shown in Equation 2-4 now gives a system of partial differential equations in *f* and *g* that can be converted into a linear system of equations using a finite difference scheme, and both *f* and *g* can then be solved simultaneously using an iterative method similar to Gauss-Seidel.

## 2.2 Related Work

There has been a large collection of literatures addressing the architecture, programming, parallelization, and performance issues with CUDA on modern many-core GPUs [36], [43], [46], [51], [58], [59]. Many performance results of various applications and published papers are displayed in the CUDA Zone [45]. Due to the tight resource constraint on GPUs, program optimization is difficult and nonlinear. In [59], the authors performed an exhaustive search on the optimization space for applications running on NVIDIA GeForce 8800 GPUs. They demonstrated that some small changes in resource allocations could result in significant varying effects for the system performance due to the lack of runtime control for thread scheduling and resource allocation. For high-performance, global memory bandwidth must not be the bottleneck during the execution. Dealing with the memory bandwidth issue using

software-managed local memories has been discussed in [58]. A memory model to improve the performance of nested loops on GPUs is reported in [23]. An analytical model for estimating the number of memory requests to show the critical role of global memory access latency in the overall execution time is described in [25]. In [64], a Faster Global Barrier was proposed to avoid going back to host for synchronization. Recently, a GPU simulator is implemented for analyzing performance of CUDA applications [5]. Fung et. Al [20] proposed a dynamic warp regrouping technique to minimize the performance degradation stemming from divergent branches. Based on their evaluation, 4.7% hardware overhead can achieve 20.7% performance benefits. Meng et. al. [40] proposed a dynamic warp subdivision scheme which divides a warp into warp-splits upon branch divergence and memory latency divergence to further improve the GPU performance. Lee et. al. [11] proposed a GPGPU-specific software and hardware prefetching mechanisms to improve the performance of GPGPUs.

There are many works done in solving the problem using the iterative algorithm and its performance on many-core GPUs [6], [11], [39], [41], [63]. A mathematical survey summarizes known convergence conditions for the iterative algorithm [10]. Among these works, [63] implemented the Jacobi's iterative method for the 2-D Poisson equation on a structured grid for a heterogeneous multi-CPU and multi-GPU architecture. They reduced the synchronization bottleneck between iterations by basic fast-and-loose asynchronous algorithms based on chaotic relaxation. However,

they did not have separate communication threads and did not consider that CTAs would be executed in groups which can affect the communication efficiency. In addition, we move the boundary values with long-stride accesses into a small stride-1 data array to be efficiently written and read to and from the global memory. Furthermore, we thoroughly studied the impact of different frequency of local synchronization and communication.

### 2.3 Parallelization of Applications

Our focus is on improving the performance of the core PDE solvers. The numerical solutions for the PDEs are obtained by iteratively updating the values at each grid point with convergence criterion supplied by the users. Typically, the value at a grid point is updated using the values at its neighbors (including itself) from the previous iteration. On a uniprocessor, the updates can be performed only sequentially and the overall running time depends on the size of the grid as well as the specified convergence criterion. However, the Gauss-Seidel-type PDE solver here does not require exact sequential updates in the sense that it is not necessary to use the values of its neighbors at previous iterative step. Numerically, the exact synchronization is not required since the convergence property of the algorithm would not be fundamentally altered by asynchronous updates that form the core of our parallelization scheme.

The basic approach is to parallelize the updates in each iterative step using multiple CTAs. Each CTA produces its own local updates and the updated boundary

values are propagated through the global memory to its neighboring CTAs. As illustrated in Figure 2-5, the entire data grid in the global memory is divided into a set of smaller rectangular blocks (data blocks), and each data block is loaded into one shared memory and updated by one CTA. Every data block contains a number of *core* nodes plus a few *boundary* nodes, whose values will be used by the corresponding neighboring blocks. In the illustrated example, each boundary node between blocks 5 and 6 is marked by a little dot. The shaded nodes represent the boundary data from the neighboring blocks as indicated by the dashed arrows.

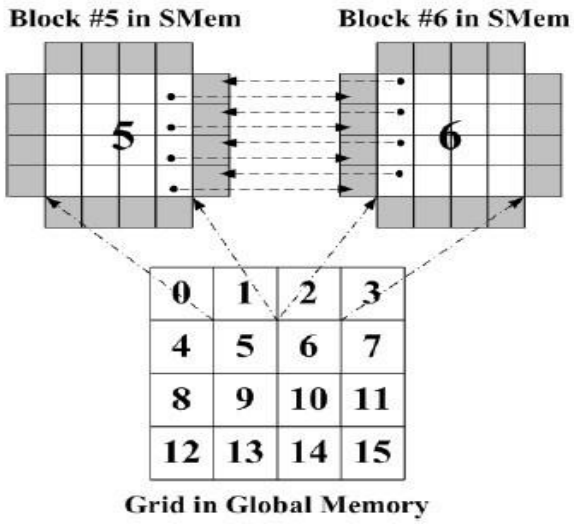


Figure 2-5. Data exchange between adjacent CTAs

### 2.3.1 Asynchronous Data Communication

A straight-forward approach is to synchronize all CTAs back to the host after each iterative step on the GPU. The host checks the convergence and decides whether to invoke another kernel call to continue the iterative step. Before exiting, all CTAs must save the intermediate results from the shared memory to the global memory. In the next iterative step, these results along with boundary nodes

computed by other adjacent CTAs in the previous iterative step are reloaded from the global memory.

Since precise data communication is not required for the iterative PDE solver, one intuitive approach to alleviate the synchronization and communication overhead is to reduce the frequency of host synchronizations. Instead of each iterative step, a *coarse* synchronization approach synchronizes the CTAs after *n* iterative steps. During the *n* iterative steps, each CTA is executed independently without exchanging data with its neighboring CTAs. Although simple without data exchange overhead, the coarse synchronization likely slows down the convergence since the boundary values in each CTA cannot be updated using new data from its neighboring CTAs during the entire *n* iterative steps.

To speed up the convergence in the coarse synchronization approach, we use *asynchronous* data communication method among CTAs during *n* iterative steps between adjacent host synchronizations. Each CTA periodically sends the newly computed boundary values to the global memory for its neighboring CTAs. Meanwhile, the CTA fetches the new boundary values computed by the neighboring CTAs from the global memory to continue the following iterative steps. We call this data exchange activities as an *inter-CTA* communication. As there is no specific order for exchanging the data between neighboring CTAs, each CTA may not always fetch the newest computed boundary values. Nevertheless, as the iterative step

moves forward, each data exchange phase will likely get newer data from its neighboring CTAs.

There are five important performance considerations in this *asynchronous* inter CTA data communication method: The first consideration is to create separate *communication threads* in each CTA for moving the boundary values from/to the global memory. Separating communication threads from the computation threads overlaps the overhead of data communication with the real computation. Care must be taken, however, to make sure that the increase of the communication threads will not prevent the CTAs from scheduling due to the resource constraint in each SM.

The second consideration is the need for the *local synchronization* between adjacent phases of data exchanges. Given separate computation and communication threads, the communication threads should wait for all computations to reach to a proper synchronization point before the data exchange phase can begin. Such local synchronizations synchronize the execution pace between the computation and communication threads and guarantee to store the newest boundary values to the global memory. It pays a small overhead to synchronize all threads in a CTA in each data exchange phase.

The third consideration is the *frequency* of data exchanges within the *n* iterative steps. Clearly, the most important factor is the ability to overlap the communication threads with the computation threads. Too frequent of data exchanges could make the communication threads a bottleneck and slows down the computation threads.

Also, although more frequent data changes help the convergence, it increases the global memory traffic and can therefore slow down the iterative step.

The fourth consideration is the resource constraints in scheduling the grid on the GPU and its impact on inter CTA data communication. Given a large problem size, it is conceivable that all the CTAs in a grid cannot be scheduled at once to the available SMs. Instead, a grid is executed by groups of CTAs sequentially. Each group consists of a portion of the CTAs that can be scheduled together. A CTA, once scheduled to run on an SM, will not release its allocated resources until it finishes the entire execution. This constraint further disrupts the asynchronous data communication. Even with periodic data exchanges, new boundary data would not be available if the data is produced by a CTA that has not been scheduled. Therefore, an important strategy in parallelizing the applications is to minimize the boundary communication among multiple scheduling groups of CTAs so that less iterations will be needed to converge.

The fifth consideration is the efficiency of data exchange through the global memory. To improve the efficiency, NVIDIA’s GPU coalesced global memory accesses from 16 threads in a half warp with regular access pattern (e.g. stride-1 accesses). To avoid inefficient data exchange, we move the boundary values with long stride accesses into a stride-1 data array to be efficiently written and read to and from the global memory.

### 2.3.2 Choices of Parallelization Parameters

**1) Choices of Thread/Data Block Sizes:** One obvious way to minimize the data communication is to lower the number of boundary nodes. Simple geometric observations reveal that larger and square blocks are preferred over smaller and rectangular blocks given the same number of nodes. However, the choice of the data block size (number of nodes) for each CTA is also influenced by two other factors. First, the size of the CTA is limited by the size of the shared memory. Second, the global memory accesses from all threads of a half warp can be coalesced into a single memory transaction as long as the access pattern among the threads forms stride-1 accesses. Therefore, for efficient *coalesced* memory accesses, the width of the boundary nodes in the column dimension is set to a multiple of 16. Furthermore, the number of threads in each CTA must be confined within the allowable 512 threads for C1060 and 1024 threads for GTX580. Therefore, with 5 data blocks needed in the algorithm to be stored in shared memory for computation, we select the data block size of 32×20 for C1060 and 32×60 for GTX580 to provide the most efficient data communication, meanwhile, each thread computes and updates two data nodes in each iterative step to reduce the number of threads per CTA to 320 for C1060 and 960 for GTX580 that leaves rooms for the communication threads.

**2) Computation vs. Communication Threads:** Two types of threads: *computation* and *communication* are used in overlapping the data exchange of the boundary nodes between adjacent CTAs with the real computation functions. For a

problem size of 480×480 in our experiment, there are 15×24 CTAs for C1060 and 15×8 CTAs for GTX580; With the limit of 512 threads per CTA for C1060 and 1024 threads per CTA for GTX580, this parallelization leaves a room of 192threads and 64 threads respectively for data communication. As illustrated in Table 2-1, after loading the needed data block into the shared memory, the computation and communication threads execute independently. *__syncthreads()* is used to synchronize all threads in each data exchange phase. Depending on the frequency of data exchange, the computation threads can go through multiple (*I*) iterative steps before synchronizing with the communication threads. This synchronization guarantees to store the new updated boundary data to the global memory in each data exchange phase. Such synchronization is necessary, especially when the delays between computation and communication can be very different.

Table 2-1. Computation threads VS communication threads

| Computation Threads | Communication Threads |
|---|---|
| *Initialization Phase:*<br>Load 30×20 data nodes into shared memory<br>*__syncthreads()* | *Initialization Phase:*<br>Load the boundary values into shared memory<br>*__syncthreads()* |
| *Main Phase:*<br>While not done {<br>Compute *I* iterations<br>*__syncthreads()*<br>} | *Main Phase:*<br>While not done {<br>Store and Load boundary values to/from Global memory<br>*__syncthreads()*<br>} |
| *Ending Phase:*<br>Store the new 30×20 data nodes to global memory | *Ending Phase:* |

**3) CTA Scheduling:** Due to insufficient hardware resources, scheduling of CTAs to maximize the benefit of inter CTA communication is essential. For Tesla C1060, there are 30 SMs per GPU, hence, up to 30 active CTAs can be active at

any given time and for GTX580 with 16 SMs per GPU, up to 16 CTAs can be active concurrently. However, given a 480×480 grid in our experiment, we have much more CTAs. All the CTAs in a grid are executed by groups sequentially. The sequential group execution impacts the inter CTA communication since new boundary data cannot be available if the data is produced by a CTA that has not been scheduled. Therefore, it is important that the scheduled CTAs need to be spatially contiguous on the grid so that inter CTA communications are effective among active neighboring CTAs. The thread execution manager schedules the CTAs according to their CTA IDs. We experiment with two group scheduling techniques: *Stripe* (linear) and *Square* (tiled). In the stripe configuration as shown in Figure 2-6, the CTA IDs are assigned so that the set of active CTAs will form a stripe to maximize the inter CTA communication on the side with efficient coalesced global moves. Similarly, in the square configuration, the set of active CTAs will form a square to maximize the boundary nodes among the active CTAs.

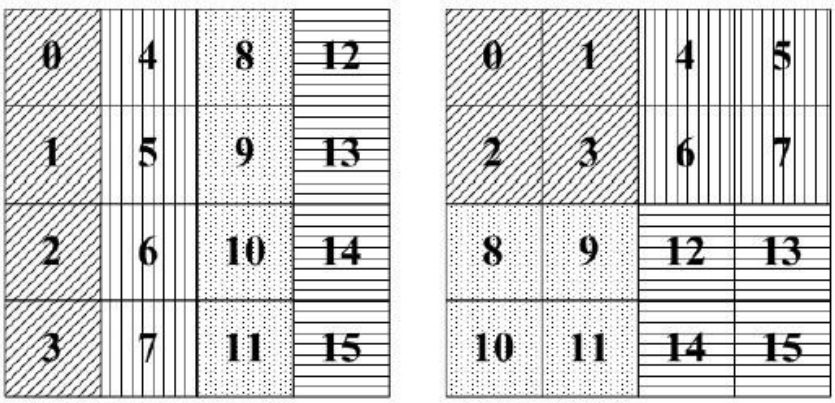


Figure 2-6. Stripe vs. square CTA scheduling (CTAs with the same filling are executed concurrently)

## 2.4 Performance Evaluation

To carry out the experiment, both applications, poisson image editing (*Poisson*) and 3D shape from shading (*3DShape*) were parallelized with CUDA 2.3 on NVIDIA Tesla C1060 and CUDA 3.0 on GTX580. In our experiment, we used a problem size of 480×480 for both applications using the established convergence thresholds reported in the literature [27], [53]. We experiment with *eight* combinations of different synchronization and communication mechanisms, data combination techniques, and CTA scheduling strategies. As the basis of comparison, the first approach is to synchronize each iterative step through the host (*Base*). The second approach is to use coarse synchronization (*Coarse*) through the host after certain number of iterative steps without any data communication in between. The remaining approaches use coarse host synchronizations as the basis but allow data communications among CTAs between two consecutive host synchronizations. There are three data communication options. First, each CTA only communicates with its up and down neighbors. Given the fact that the nodes in the upper and lower boundaries of a CTA are stored continuously in memory, hence can be coalesced, this option provides efficient data communications. Second, each CTA communicates with all four neighbors to get the needed new data for the boundary nodes. Third, since the left and the right boundaries of each CTA are not consecutive in memory, this option combines the non-consecutive boundary nodes into a small array to shorten the communication cost. Furthermore, we also compare two CTA

scheduling options, *stripe* and *square* for alleviating the impact on scheduling and executing the grid by groups of CTAs. Scheduling CTAs in squares reduces the number of adjacent nodes that have not been scheduled. In combining these communication and scheduling options, the third, fourth, and fifth, approaches use stripe scheduling with up/down communication (*UD-commstripe*), 4-way communication (*4W-comm-stripe*), and 4-way with data combining (*4W-comm-combine-stripe*) respectively. Similarly, the sixth, seventh, and eighth approaches using square scheduling with up/down communication (*UD-commsquare*), 4-way communication (*4W-comm-square*), and 4-way with data combining (*4W-comm-combine-square*).

### 2.4.1 Execution Time and Convergence on C1060

We first examine the total execution time and convergence speed of various communication and scheduling mechanisms on C1060. In this study, the applications are parallelized with 320 computation threads and 32 communication threads in each CTA. The interval of data communication is set to be every 4 iterative steps. More sensitivity studies will be given in Section V-B. Figure 2-7 shows the total execution time and the number of iterations to reach the convergence of *Poisson* for the eight communication and scheduling combinations. The results are measured over different intervals of host synchronization ranging from every 10 to every 100 iterative steps. Note that due to a much longer execution time (53.6 sec), the *Base* approach of using fine-granularity synchronization through

host is not shown in Figure 2-7. Similarly, Figure 2-8 plot the results of *3D-Shape*, where *Base* took 95.4 sec to execute and is also omitted in Figure 2-8.

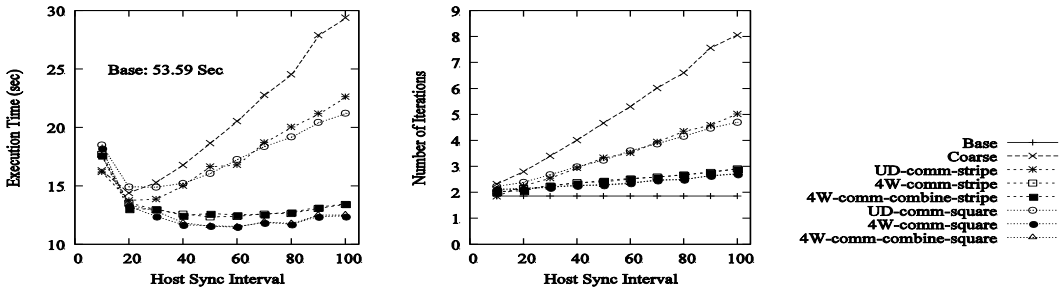


Figure 2-7. Execution time and Number of Iterations of Poisson using eight communication/scheduling schemes

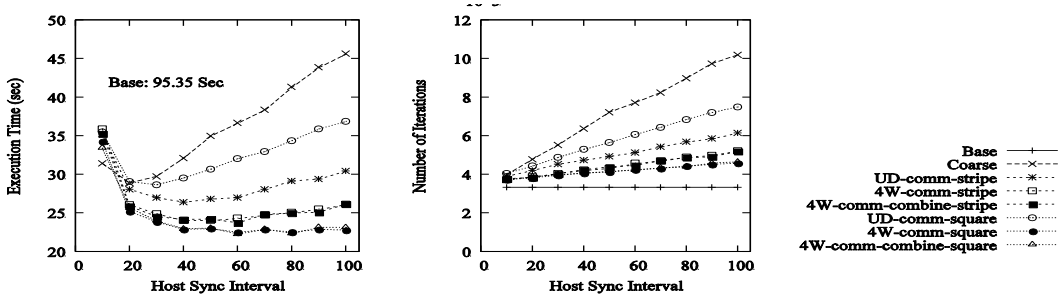


Figure 2-8. Execution time and number of iterations of 3D-Shape using eight communication/scheduling schemes

We can make several important observations. First, we observe a significant execution time improvement using the asynchronous inter CTA data communication mechanisms over the fine-granularity host synchronization for both applications. The improvement is also substantial in comparison with *Coarse*, where no data communication occurs between the host synchronizations. Although *Base* has the fastest convergence with respect to the number of iterative steps, the overhead of going back and forth between the host and the GPU increases the execution time per iterative step by about 5 times. With a host synchronization every 60 iterative steps, the overall execution times for *Poisson* are reduced from 53.6 seconds for

*Base* to 11.4 seconds for *4W-comm-combine-square*. Similarly for *3D-Shape*, the execution times are reduced from 95.4 seconds to 22.2 seconds. A speedup of 4-5 times can be achieved for both applications.

Second, among various data communication and scheduling approaches, the 4-neighbor communication with the square scheduling shows the fastest execution time, followed by the same 4-neighbor communication with stripe scheduling. Both approaches are relatively insensitive to the interval of host synchronization as long as the interval is 20 iterative steps or greater revealing reasonably accurate data communication between two host synchronizations. For the CTA scheduling, *stripe* suffers from more adjacent unscheduled CTAs, hence shows slower convergence than that with *square* scheduling.

Third, communicating with only the upper and lower neighbors has worse performance than that of the 4-neighbor communications. Although communications only with upper and lower neighbors can coalesce all global memory accesses and reduce the effective data moves, the lack of a portion of the new boundary values greatly slows down the convergence.

Fourth, the data combining technique for non-stride-1 nodes in the left and right boundaries of each CTA during data communication does not show any noticeable difference. Detailed analysis discloses that data combining indeed reduce global memory moves. However, lengthening the communication may not affect the

execution time as long as the bottleneck is in the computation threads. More evidences will be given in the sensitivity study.

Fifth, the general behavior between the two applications with respect to the data communication and CTA scheduling is very similar. Numerically, however, it takes almost twice as long for *3D-Shade* to converge as that for *Poisson*.

### 2.4.2 Experiment with GTX580

There are many performance enhancements in Fermi from early GPU generations. In this work, we focus on the features that are related to exploiting the proposed weak execution ordering approach in running iterative numerical solving methods. Our experiment is carried out on Nvidia's GTX 580.

Like early generations, Fermi supports multiple-level of parallel programming models. At the highest level, Fermi supports simultaneous execution of multiple kernels from the same applications. Each kernel consists of many *threads* that execute the same program in the SIMT fashion. Threads are grouped into *thread blocks* which can be distributed to one or more SMs to be executed in any order. Although concurrent thread blocks can communicate through the global memory, like in early Nvidia's GPU, there is no mechanism in the GPU device to enforce any synchronization order among the concurrent thread blocks.

One of the most noticeable enhancements in Fermi is its memory hierarchy system. Each SM now has 64 KB fast on-die RAM which can be configured to either 16KB L1 cache with 48KB shared memory or 48KB L1 cache with 16 KB shared

memory. In addition, Fermi GPU has a 768KB on-die L2 cache shared by all SMs. The option of the L1 cache provides flexibility to cache data with dynamic access patterns. Fermi provides cacheable and non-cacheable options to the L1 cache. However, there is no hardware cache coherence mechanism among multiple L1 caches. Whenever a local L1 block is modified, the block is invalidated and the modified block is written back to the shared L2 to keep the latest copy. In this case, however, any existing stale copies will remain in other L1 caches.

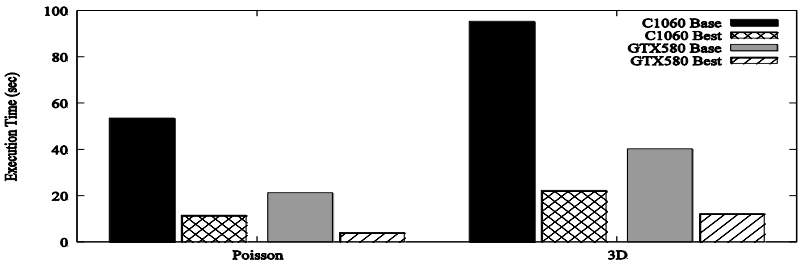


Figure 2-9. Comparison of *Base* and the best Execution time and number of iterations on both GPUs

In the experiment on GTX 580, we first disable the L1 cache. Figure 2-9 shows the execution time of the *Base* with host synchronization and the best coarse synchronization using the *4W-comm-comb-square* scheme. The best synchronization interval of 60 on C1060 and 40 on GTX 580 are used. We observe that the GTX 580 can run the *Base* with 2.52 and 2.37 times faster than the C1060 for Poisson and 3D respectively, mainly due to larger number of cores and higher per-core performance. With coarse synchronization, we observe similar improvement on GTX 580 as that on C1060. For Poisson, the execution time reduces from 21.2 to

4.1 seconds using *4W-comm-comb-square* while the execution time reduces from 40.1 to 12.2 seconds for 3D.

Next, we experiment on GTX 580 with L1 cache enabled using coarse synchronization. In Figure 2-10, the results of the *4W-comm-comb-square* scheme for both with and without L1 cache are plotted. We observe that enabling L1 cache actually results in worse performance in both execution time and number of iterations. Although L1 cache reduces global memory access latency, it provides little help to accelerate the kernel execution since communication overhead is already overlapped with computation. This is evident from the time per iteration in Figure 2-10 where both configurations have the same performance. The main cause of performance degradation with L1 caches is due to lengthening the iterations for convergence (Figure 2-10). Since there is no hardware coherence mechanism among L1 caches, the updates of the boundary nodes to the global memory originated from one SM will not cause invalidation to the stale copies in the L1 cache of other SMs. As a result, the neighbor blocks may not see the new data, hence delay the convergence.

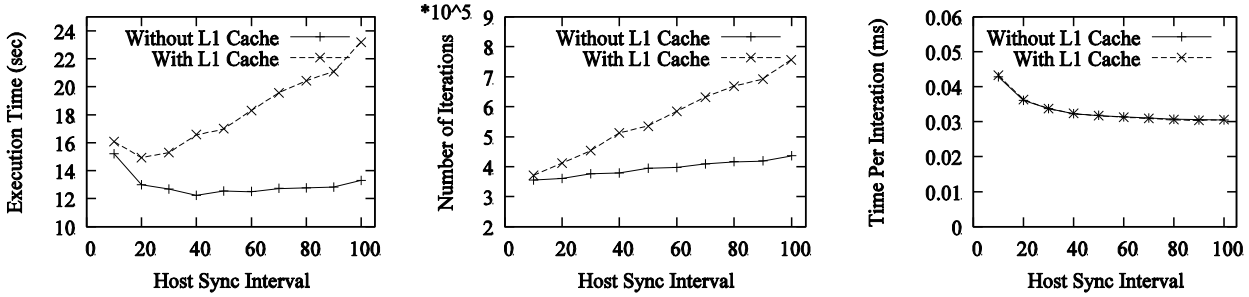

Figure 2-10. GTX 580 results with and without L1 cache

### 2.4.3 Sensitivity Study

The number of communication threads and the interval of inter CTA communication impact the performance of the asynchronous data communication among multiple CTAs. The CTA scheduling also plays an important role. In this section, we show the results of sensitivity studies based on these three parameters.

We first simulate different inter CTA communication intervals varying from 1 to 10 iterative steps. In this study, we use 32 communication threads with 320 computation threads as described before. The host synchronization interval is fixed at every 60 iterative steps. Figure 2-11 shows the respective execution time, the total number of iterative steps, and the time per iterative step with different inter CTA communication intervals. We can observe that inter CTA communication on every iterative step increases the execution time significantly. This is due to the longer execution time per iterative step as shown in Figure 2-11. We collect detailed timing in terms of number of cycles for the computation and the communication threads. The computation threads takes roughly 5500 cycles per iterative step while the communication thread of *UD-comm* takes 6000-7000 cycles and *4W-comm* takes 10000-12000 cycles depending on the CTA scheduling and the boundary data combining schemes. As a result, the communication threads dominate the execution time when the inter CTA communication is on each iterative step. When the interval is lengthened to two iterative steps, the execution times of the two thread types are very similar. The computation threads dominate the execution time when the interval

becomes 4 iterative steps or longer. The interval of four iterative steps has the fastest execution time because of its ability to totally overlap the communication and computation threads with faster convergence in comparison with longer intervals. Longer intervals reduce the frequency of getting the updated boundary values. The effect of data combining becomes evident when the execution time is dominated by the communication threads. We can observe that the execution times are much faster with data combining for *4W-comm* with communication interval of 1.

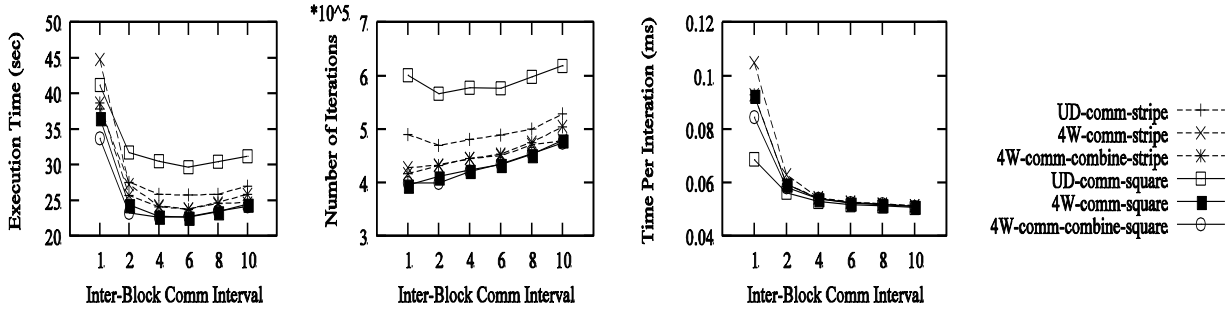


Figure 2-11. Execution time, number of iterations and time per iteration of 3D-Shape Intervals

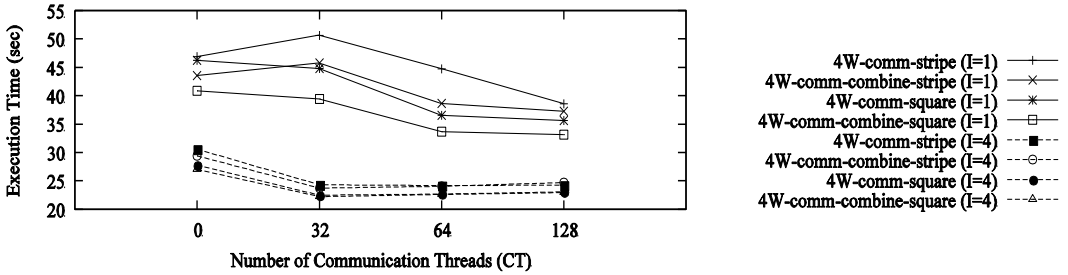


Figure 2-12. Execution time of 3D-Shape with different number of communication threads

Next, we evaluate the impact on the number of communication threads. We try the inter CTA communication interval of 1 and 4 iterative steps. Recall that one-step interval creates the bottleneck in the communication threads. We vary the communication threads from 0 to 128 threads. Since there are 320 computation

threads, a single CTA can accommodate up to 192 separate communication threads. Without communication threads, we use the computation threads to accomplish the data movement. The execution time with different number of communication threads are plotted in Figure 2-12. Note that we omit *UD-comm* since it does not communicate all needed boundary values and performs worse than *4W-comm*.

Three observations can be made from the results. First, as expected, one-step interval takes much longer execution time than that for the four-step interval. This is again due to the bottleneck in the communication threads with one-step interval. Second, when the computation thread is the bottleneck, integrating the communication function into the computation thread increase the execution time as shown by the results of four-step interval. Third, when the communication thread is the bottleneck in one-step interval, it is advantage to use 128 communication threads to move all 104 boundary values in parallel. In this case, the 0 communication threads can slightly out-perform the 32 communication threads because without communication threads, each computation thread only needs to move up to one boundary node while with only 32 communication threads, each communication thread needs to move multiple boundary values.

We also evaluate and compare the performance of random scheduling where each CTA will compute a random data block. For the random scheduling, we also experiment with and without the 4W-comm-combine communication which are denoted as *Coarse-random* and *4W-comm-combine-random*. Figure 2-13 shows

their execution time and number of iterations compared with that of the *Base, Coarse-stripe, 2W-comm-stripe* and *4W-comm-combine-square* on GTX580 with different host sync intervals. We can make the following observations from this figure. First, the random scheduling is better than the regular coarse scheduling scheme, which is due to the reduced number of iterations as shown in the figure. However, the *4W-comm-combine-square* still has the best performance. Second, the communication does not help with the random scheduling. Since the data block assignment is random, the neighboring blocks of the active ones are likely to be idle, so the communication cannot bring in new data nodes and has no effect at all.

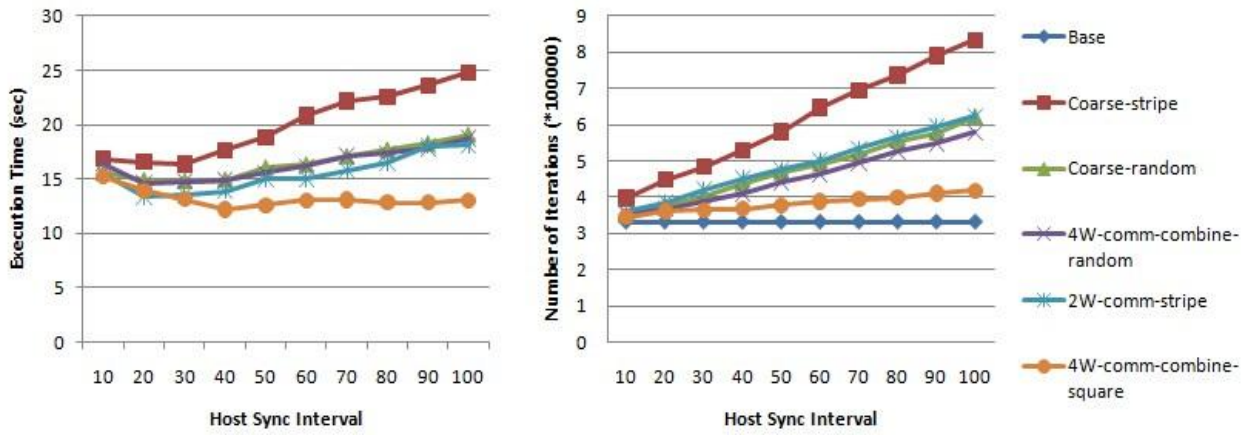


Figure 2-13. Execution time and number of iterations of 3D-Shape with random scheduling and communication

## 2.5 Summary

Inexpensive GPU hardware along with the CUDA programming model brings desk-top supercomputing to all users. However, due to the lack of precise data communication and synchronization mechanisms among the parallel CTAs on a GPU, significant overhead occurs to accomplish such a function through the host

CPU. In this work, we thoroughly study and analyze the inter CTA communication and its impact on an important class of applications based on chaotic relaxation of PDE solvers. During parallel computation on the GPU, separate and independent communication threads hide the communication overhead for exchange the needed boundary data. Performance measurement on Tesla C1060 showed that by reducing the host synchronization, a speedup of 4-5 times can be achieved for the two studied applications compared with the host synchronization implementation.

# CHAPTER 3
# STATISTICAL GPU POWER ANALYSIS USING TREE-BASED METHODS

As described in Chapter 1.2, we will compose the GPU power model to predict and understand power consumptions in the second work. Chapter 3 is organized as follows. We will first introduce the motivation and an overview of model composition in 3.1. Related work will be given in 3.2 followed by the details in power modeling in 3.3. The power prediction verification and understanding is described in 3.4. 3.5 summarizes Chapter 3.

## 3.1 Introduction

GPU can achieve impressive GFLOPS in many kernels, but it also consumes high power. As introduced in Chapter 1.2, the Tesla 20-series GPU can deliver up to 500+ gigaflops of IEEE standard double-precision floating point operations and over a teraflop of single-precision peak performance with a power consumption of 200-300W. In the Appro 1426G4 server [2] with four Tesla 20-series GPUs and two Quad/Six-Core Intel Xeon Processors with 96GB DDR memory, the power supply requirement is 1400W. Therefore, it is important to understand and be able to predict GPU power consumption so that we can better design and use this architecture.

Existing GPU power measurement tools are very limited. Since the current GPUs are not equipped with power sensors to directly assess the power dissipation at runtime, the GPU power is either observed from a separate power meter or derived from the probed voltage and current on the GPU. Such approaches require additional hardware devices and are not a practical solution. On the other hand, software approaches to estimate the power consumption are still in the early stages.

Unlike CPU power, which can be modeled at different levels by simulation tools such as Wattch [9] and Simple power [66], the GPU community lacks an accurate and flexible tool for power simulation. Although some preliminary attempts have been made, the existing tools are limited to a specific architecture or component and more importantly, they cannot provide much insight about GPU power consumption due to limited analyzing ability and metrics they can collect. For example, Qsilver [60], an OpenGL based performance simulator, was extended to include a power model for the OpenGL rendering pipeline. PowerRed [54] includes both analytical and empirical power model along with an area-aware interconnect model to estimate power at the architectural level.

In this work, we will compose GPU power models with several statistical tools including linear regression trees [8] and random forest methods [7]. Other than a high accuracy of prediction, the random forest method provides insightful analysis between the power consumption and the independent performance variables. The sample data used to feed this model is mainly the execution frequencies of instructions and memory accesses in a wide-variety of application samples. We also include several performance-sensitive architectural metrics into the analysis. Our methodology consists of the following steps.

**Workload profiling and power measurement**: We collect 52 application kernels from popular GPU benchmark suites [12], [49], [56] and from a few real applications [14]. These applications are parallelized using CUDA and run on an experimental system with GTX 280. The energy consumptions of these kernels are

measured using a YOKOGAWA WT210 Digital Power Meter [67] as statistical power samples for power consumption prediction and analysis.

**Measuring independent variables from sample application kernels**: A GPU simulator *GPGPUSim* [5] for simulating application execution is used to measure the executed instructions and memory accesses as well as other important architectural parameters. Unlike the GPU profiler [49], *GPGPUSim* is flexible and can be modified to collect execution statistics. In this study, we modify *GPGPUSim* to collect 18 execution related and 4 architecture related metrics and use them as the independent variables for power analysis.

**Statistical analysis from the samples**: Based on 52 sample application kernels, we first use multiple linear regression analysis to build the power equation with a set of independent variables. We then apply the sophisticated random-forest method [7] to model the power with high accuracy. The random forest approach can also shed insights on the correlation between power and individual variables.

**Power prediction, correlation and verification**: The random forest model identifies most influential variables in power prediction including register access (Register), single-precision floating-point instruction (S.FP) and global memory access (GMInst). It also shows that several performance-sensitive architecture metrics indeed play a role in modeling the power consumption. In addition, a random forest model can output a proximity plot, which provides an indication of which observations (kernels) are effectively close together in the eyes of the random forest model Furthermore, we verify the accuracy of the model using leave-one-out cross

validation (LOOCV). The average percentage error (PE) is 7.77% in comparison with the measured power consumption for the random forest method. The average PEs increase to 11.68% and 11.70% respectively for using the regression tree and the multiple linear regression approaches.

### 3.2 Related Work

Running large-scale parallel applications on modern GPUs has been widely adopted recently. Although there have been many reports for the performance analysis on GPUs [37], [59], [64], [68], the study of the important GPU power consumption issues is very limited. Hong and Kim [26] presented an empirical model which accurately predicts GPU run time power from activities of individual components including floating point unit, register file, ALU, etc. They rely on an early performance model [25] to estimate the execution behavior, instead of predicting the power using statistical samples. Due to the limitation of their model, applications with intensive control flows, asymmetric execution among CTAs, and intensive use of texture and constant caches cannot be modeled accurately. Ma et al. [38] presented an SVR regression model to predict the dynamic power consumption of a GPU with the counters from *perfkit*. Since *perfkit* is developed to debug and profile OpenGL and Direct3D applications, their model is more suitable for graphics applications. Nagasaka et. al [42] proposed a linear regression based statistical model to predict GPU power using performance counters collected from the runtime profiler [49]. With GTX 285 GPU and CUDA version 2.3, their model is able to predict the dynamic power reasonably accurate. A recent work-in-progress [13] is the first using the

random forest method to model GPU power consumption with limited correlation analysis.

Our approach extends the model using the random forest method and is superior in two aspects. First, we show that the random forest model for predicting GPU power consumption is more accurate than the approaches using linear regression and regression tree. Second, with more performance variables collected from *GPGPUSim*, we use the random forest model to rank the importance of individual variables and their partial dependences to the power consumption. This correlation analysis helps computer architects for better GPU designs. It also helps compiler developers for better use of energy/power efficient GPUs.

Efforts have been made to design frameworks for the GPU power simulation. For example, Ramani et al [54] introduce the PowerRed tool, which is an attempt to model the GPU power at the architectural level. There are also few works concentrating on the tradeoff between performance and energy consumption of GPUs. In [57], Rofouei et al. use a novel platform to collect runtime energy dissipation of a computing system when running different applications. They demonstrate that a GPU is more energy efficient compared to a CPU when the performance reaches above certain bound. Huang et al. [28] conduct a similar study to investigate the energy efficiency on a hybrid CPU+GPU environment. Ren et al. [55] consider different implementations of matrix multiplication kernels and run them on different devices (i.e., CPU, CPU+GPU, CPU+GPUs) to compare the respective

performance and energy consumptions. They show that when the CPU is given an appropriate share of workload, the best energy efficiency can be delivered.

## 3.3 GPU Power Modeling

### 3.3.1 Characteristics of Kernel Execution and Profiling

The GPU power consumption is closely related to the runtime characteristics during the kernel execution. In this section, we present all the runtime characteristics that we analyzed and describe the methods for profiling them. In general, the total execution frequency of different types of instructions and memory accesses largely determines the GPU power consumption. However, the execution efficiency also influences the power assumption. In this study, we include four performance-sensitive architectural parameters.

In order to collect the kernel execution characteristics, we use a GPU simulator *GPGPUSim*, which is a cycle-level simulator that simulates the PTX instruction set running on Nvidia GPUs. The PTX resembles the real ISA of Nvidia GPU and has demonstrated reasonably close IPC results [1]. *GPGPUSim* consists of a function simulator (*cuda-sim*) that executes PTX kernels, a performance simulator (*gpgpu-sim*) that simulates the timing behavior of a GPU, and an interconnection network simulator (*intersim*) that models the interconnect delays. After the *cudafe* separates GPU and CPU code, the *nvcc* compiles and produces the PTX assembly code for GPU. Then, the *ptxas* assembler generates thread-level register, CTA-level shared memory, and CTA-level constant memory usage information to determine how many CTA can be assigned to each SM. *GPGPUSim* also implements a custom CUDA

runtime library to divert CUDA API calls to *GPGPUSim*. During the simulation, the simulated kernel C code runs natively on the CPU of the simulated machine.

Table 3-1. Runtime characteristics from GPGPUSim

| Notation | Description |
|---|---|
| ShMInst | Total shared memory instructions |
| PMInst | Total parameter memory instructions |
| LMInst | Total local memory instructions |
| GMInst | Total global memory instructions |
| TMInst.hit | Total texture cache hit |
| TMInst.miss | Total texture cache miss |
| CMInst.hit | Total constant cache hit |
| CMInst.miss | Total constant cache miss |
| Register | Total register accesses |
| S.FP | Total single-precision FP instructions |
| D.FP | Total double-precision FP instructions |
| INT | Total integer instructions |
| ALU | Total arithmetic logic unit instructions |
| SFU | Total special function unit instructions |
| Atomic | Total atomic instructions |
| Barrier | Total barrier instructions |
| M.Barrier | Total memory barrier instructions |
| Branch | Total branch instructions |
| UncoalesMem | Total global memory accesses that cannot be uncoalesced |
| D.Branch | Total divergent branches |
| BankConf | Total bank conflicts in accessing caches and shared memory |
| Occupancy | Ratio of active warps to the maximum number of warps can be supported in a SM of GPU |

Table 3-1 summarizes the 22 runtime characteristics including 18 execution frequencies and 4 architecture parameters we collected from *GPGPUSim* for power modeling and analysis. We separate accesses to register, shared memory, local memory, global memory, texture memory and constant memory for more accurate modeling. Since the texture and constant memory accesses are cacheable, we further separate them into hits and misses to reflect their power consumption.

There are four performance-sensitive architectural features to be considered in our power analysis. First, individual global memory accesses with certain regularity from all threads in a half-warp (assigned to the same functional unit of the SPs) can

be coalesced into one or fewer memory moves for reducing the bandwidth requirement. In establishing the power model, the *un-coalesced* global memory accesses can be used complement the total global memory accesses. Second, in the SIMT mode, a single instruction must be executed by all simultaneous threads in a half-warp. Divergent branches which produce different outcomes among simultaneous execution threads must execute the true path and the false path separately, hence can be used as a complement to the total number of branches in the power model. Third, on-chip storage such as shared memory, caches, and registers are constructed with 16 banks to improve the bandwidth. Bank access conflicts cause serialization of thread execution in a half-warp. These bank conflicts impact the power consumption for instructions accessing the shared memory, caches, and registers.

Fourth, Nvidia's GPUs and CUDA present many constraints in creating and scheduling parallel threads on multiple SMs as shown in Table 1-1. It relies on the programmers to optimize the CUDA code for efficient utilizing the underline hardware. The basic optimization strategy is to fully populate all SPs in a SM with simultaneous CTAs and threads to achieve the best hardware utilization. Unfortunately, the optimization space of CUDA/GPU is multi-dimensional and non-linear [59] such that optimizing one constraint often impact another constraint. The SM *occupancy* is an important metrics reflecting the scheduling and execution efficiency of CTAs in SMs and will also be used in the power analysis.

It is important to note that due to the limited sample size we collect, it may encounter the over-fitting problem in constructing the power model with large number of independent variables. As a result, the prediction accuracy may be sacrificed. Nevertheless, the goal of this work is not only to build an accurate model for power prediction, but more importantly, to analyze and understand the correlation of power consumption to different runtime characteristics.

### 3.3.2 Sample Workloads and Power Measurement

To cover a wide variety of GPU application kernels, we selected kernels from *CUDA SDK* [49] with many popular applications. We also use kernels from *Rodinia* Benchmark [12] which covers a wide range of applications in parallel communication patterns, synchronization techniques, and power consumption studies, and from *Parboil* Benchmark [56] which is comprised of various applications developed to be better suited for measuring GPU performance. We also collect kernels from a computer graphics application *Poisson Image Editing* [14] which seamlessly integrates new image contents into a given background image, and a computer vision application *3D Shape from Shading* [14] which deals with recovering the 3-D structure of an object from its image. Due to the constraints in the simulator with unsupported instructions, we cannot run a few kernels from these benchmark suits. In addition, we adjust a few small kernels (<0.5ms) to a bigger problem size. In case that the cost of kernel invocation and host return dominate the execution time, these small kernels are not included. Lastly, we exclude kernel *besect_large* in *eigenvalues* which has only 1 CTA. Table 3-2 lists all the 52 kernels we used.

Table 3-2. List of kernels

| Benchmark | Kernels |
|---|---|
| 3DFD | 1.stencil_3D_16x16_order8 |
| binomialOptions | 2.binomialOptionsKernel |
| BlackScholes | 3.BlackScholesGPU |
| convolutionSeparable | 4.convolutionColumnsKernel |
| | 5.convolutionRowsKernel |
| convolutionTexture | 6.convolutionColumnsKernel |
| | 7.convolutionRowsKernel |
| dct8x8 | 8.CUDAkernel1DCT |
| | 9.CUDAkernel1IDCT |
| dxtc | 10.compress |
| fastWalshTransform | 11.fwtBatch1Kernel |
| | 12.fwtBatch2Kernel |
| Histogram | 13.histogram64Kernel |
| matrixMul | 14.matrixMul(single) |
| | 15.matrixMul(Double) |
| MersenneTwister | 16.RandomGPU |
| | 17.BoxMullerGPU |
| quasirandomGenerator | 18.quasirandomGeneratorKernel |
| scalarProd | 19.scalarProdGPU |
| scan | 20.scan_best |
| | 21.scan_naive |
| | 22.scan_workefficient |
| simpleAtomicIntrinsics | 23.testKernel |
| sobolQRNG | 24.sobolGPU_kernel |
| Transpose | 25.transpose |
| | 26.transpose_naïve |
| sad | 27.mb_sad_calc |
| mri-q | 28.ComputeQ_GPU |
| lbm | 29.performStreamCollide_kernel |
| backprop | 30.bpnn_layerforward CUDA |
| | 31.bpnn_adjust_weights_cuda |
| Heartwall | 32.kernel |
| hotspot | 33.calculate_temp |
| Particlefilter | 34.kernel |
| lud | 35.lud_internal |
| nw | 36.needle_cuda_shared_1 |
| | 37.needle_cuda_shared_2 |
| histogram | 38.histogram256Kernel |
| MonteCarlo | 39.MonteCarloOneBlockPerOption |
| 3D-shape | 40.SI-Base |
| | 41.SI-Coarse |
| | 42.SI-UD-comm-square |
| | 43.SI-4w-comm-square |
| | 44.SI-4w-comm-combine-square |
| Poisson-image | 45.GS-Base |
| | 46.GS-Coarse |
| | 47.GS-UD-comm-square |
| | 48.GS-4w-comm-square |
| | 49.GS-4w-comm-combine-square |
| Kmeans | 50.kmeansPoint |
| | 51.invert_mapping |
| particlefiter | 52.likelihood_kernel |

We use YOKOGAWA WT210 Digital Power Meter to measure the overall system energy consumption. Each kernel execution is repeated multiple times for more accurate measurement. The power consumption *P* can be calculated from the

measured energy *E* such that *P* = *E/T,* where *T* is the running time. However, the measured power includes both the GPU power and the system power of CPU and other components on the mother board. To isolate the system power, we remove the GTX280 card and measure the idle system energy repeatedly. The average measured system power without GPU is about 113W which is about 35-40% of the overall measured power. This constant system power can then be subtracted from the measured power of each benchmark to get the correct GPU power for the kernel samples.

### 3.3.3 Statistical Modeling

Tree-based models provide an alternative to the classic linear regression models [8]. The tree models are fitted through a recursive partitioning algorithm whereby a dataset is successively split into increasingly homogenous subsets until the information gained by additional splits is not out-weighed by the additional complexity due to the tree's growth. Tree structured models are adept at capturing non-linear and non-additive behavior, e.g. interactions among independent variables are routinely and automatically handled. Random forest, proposed by Leo Breiman [7], is an ensemble learning algorithm that combines many individual trees in the following way: For each tree, (1) a bootstrap sample is drawn from the original sample; (2) a tree is grown using the bootstrap sample of the data generated in the step (1), and at each split the candidate set of variables is a random subset of all the variables. The response variable is predicted using the average of the predictions of

all the trees in the forest. Studies have shown that the prediction accuracy of the random forest ensemble is usually better than the one from an individual tree.

Figure 3-1 shows a single regression tree with four leaves (terminal nodes) from analyzing the 52 benchmark kernels. For example, the leftmost leaf, which includes 8 kernels, indicates the average GPU power consumption for the kernels with “Register<6.863e+11” (register access less than 6.863e+11), “GMInst < 3.833e+9” (global memory instructions less than 3.833e+9) and “S.FP < 3.532e+9” (single precision floating point instructions less than 3.532e+9) is 135.7 watts .

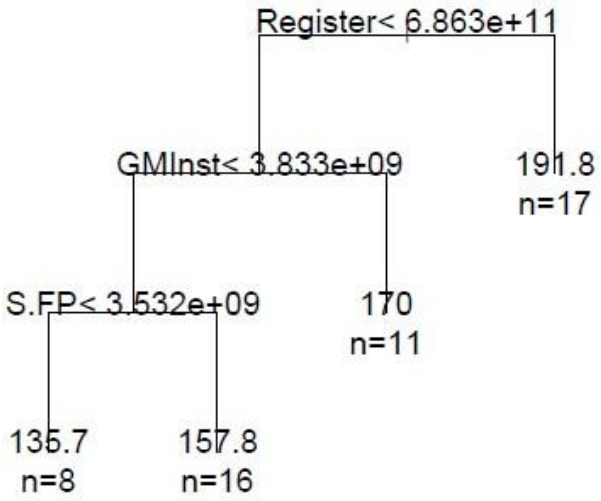


Figure 3-1. Regression tree for 52 kernels based on the parameters from *GPGPUSim*

Although a random forest model usually consists of hundreds of trees, it does not have to be treated like a black box. A random forest model can be summarized, interpreted and visualized similarly to conventional regression models. This includes identifying parameters that are most influential in contributing to the response’s variation, and visualizing the nature of dependence of the fitted model on these important parameters. The random forest model provides a relative variable

importance measure which is based on the number of times a variable is selected for splitting, weighted by the squared improvement to the model as a result of each split, and then average over all trees. The relative influence is scaled so that the sum adds to 100, with a higher number indicating a stronger influence. The ranking of relative influence of power consumption allows us to gain insight on which runtime characteristics has the most impact to the power consumption.

Another useful interpretation tool is *partial dependence* plot, which shows the effect of a subset of variables on the response after accounting for the average effects of all other variables in the model. Hence, it helps us to visualize the trend of impact of a subset of variable on the response in a low dimensional plot. In addition, a random forest model can also output a proximity plot, which provides an indication of which observations (kernels) are effectively close together in the eyes of the random forest model through a two-dimensional plot. Such information helps us understanding the similarity and difference with respect to power consumption among the sample application kernels. In this study, we fixed the number of trees in random forest to be 500. The experiments on the random forest and regression tree are done by using the *randomForest* and *rpart* packages in R, respectively [22], a free software environment for statistical computing.

### 3.4 Power Prediction and Understanding

The prediction performance is evaluated through leave-one-out cross-validation (LOOCV). LOOCV is a repeated procedure which uses a single observation from the original sample as the validation data, and the remaining observations as the training

data. Table 3-3 summarizes the prediction performance of three competing methods (random forest, regression tree and linear regression) using the LOOCV. We see that random forest is superior with an average percentage error (PE) 7.77%. In comparison, the average PEs for regression tree and multiple linear regression approaches are 11.68% and 11.70% respectively.

Table 3-3. Average PE and Mean Squared Error (MSE) of the three methods

| | Random Forest | Regression Tree | Linear Regression |
|---|---|---|---|
| PE | 7.77% | 11.68% | 11.70% |
| MSE | 302.6 | 637.8 | 2548.0 |

Comparisons between the estimated and the measured power of individual application kernels using Random Forest method are given in figure 3-2. A majority of the kernels show high accuracy in their power prediction. However, there are also a few kernels with low accuracy. For example, kernel 3.BlackScholes's predicted power is much lower than the measured power. This kernel not only has high frequencies on regular computing instructions like S.FP and ALU, it also has high frequency of SFU instructions. The SFU instruction is executed on a separate special function units other than using the SPs. As a result, both hardware units can be active simultaneously for a long period of time and consume higher power. However, since such high frequency of SFU usage together with high frequency of regular computing instructions is rarely presented in other kernels, the power model underestimates the power in this special case. Another example is kernel 17.BoxMullerGPU in MersenneTwister. This kernel has 32 CTAs running on 30 SMs. So 2 SMs execute 2 CTAs while the rest only execute 1 CTA. In this case, there are just 2 SMs working for about half of the time, resulting in low power consumption. No

other kernel has this scenario. Therefore, the model predicted much higher power consumption. In both cases, increasing the sample size should ease the inaccuracy.

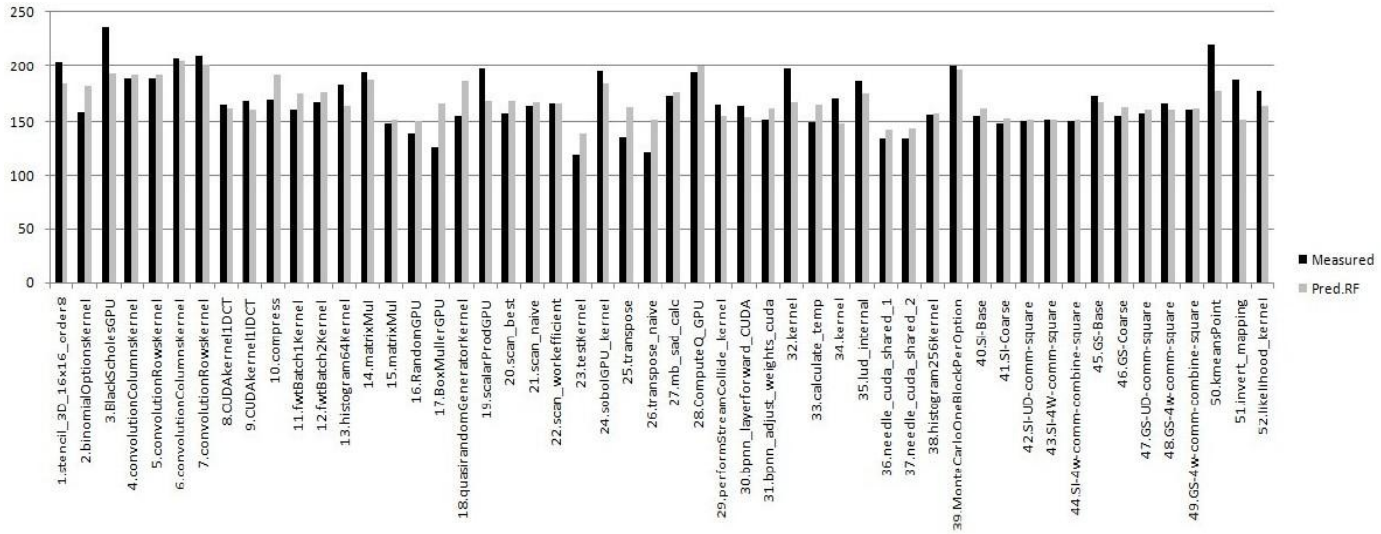


Figure 3-2. Comparison of predicted and measured power

### 3.5 Correlation Analysis

In this section, we show the correlation analysis between the dependent power consumption and the independent variables. We also show the proximity among the sample kernels. Figure 3-3 shows the relative importance for both runtime performance variables and architecture related parameters. Among all the input variables, we see that Register, S.FP, GMInst, ALU and INT are the five most influential variables. Obviously, register usage is influential since it is likely to be used in most instructions. Single-precision floating-point instructions produce high power since they represent a significant portion of instructions in floating-point kernels. Global memory moves consume high powers and are often necessary due to limited on-chip storage. ALU and Integer instructions are also ranked high since they are used for manipulating indices, decision-making instructions, and temporary variables.

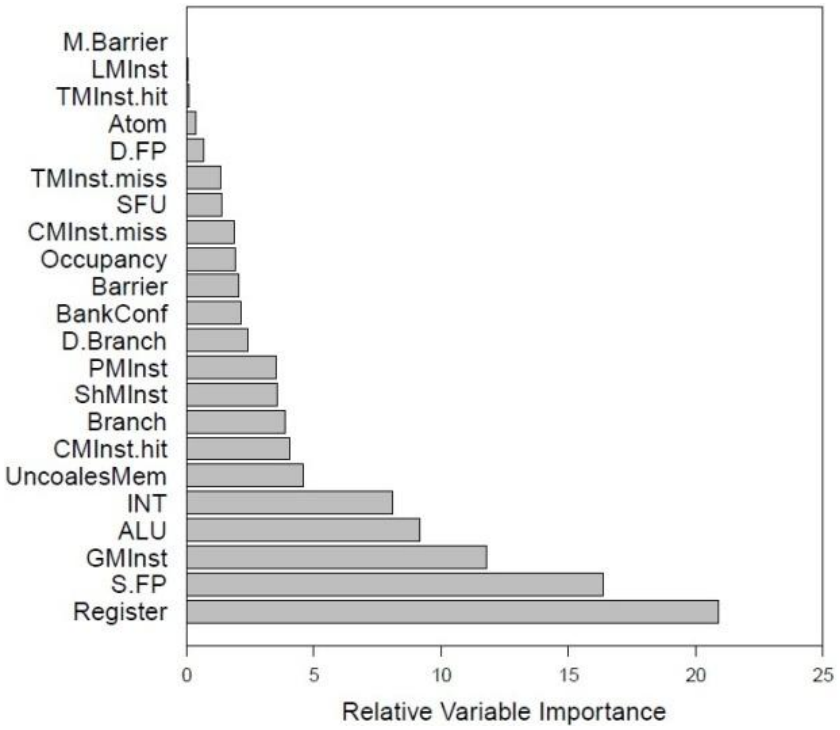


Figure 3-3. relative variable importance

It is interesting to see that *uncoalesced* global memory (UncoalesMem) is ranked sixth in relative importance. Given the fact that *GPGPUSim* does not handle multiple granularities of physical global memory moves, UncoalesMem represents the amount of GMInst that is uncoalesced, hence causes more physical global memory moves. The other three architecture metrics, D.Branch, BankConf, and Occupancy also play a role, but are not as significant as UncoalesMem.

Figure 3-4 shows the partial dependence plots for the four most influential operations (Register, S.FP, GMInst, and ALU) and four architecture related parameters. In each plot, all other metrics remain the average value. Note that in these plots, the Y-axis is scaled from 160 to 180 watts for the four most influential variables while Y-axis is from 165 to 172 watts for the architecture parameters to accommodate different range of influence. From the partial dependence plots for the

four most influential variables, we can see that the increase of power generally goes along with the variable ranking. Due to the small sample size the increment slope is not very smooth. Among the variables, we observe very little changes when the total number of ALU instructions reaches 1.6E+11. This is due to the lack of samples beyond this amount of ALU instructions.

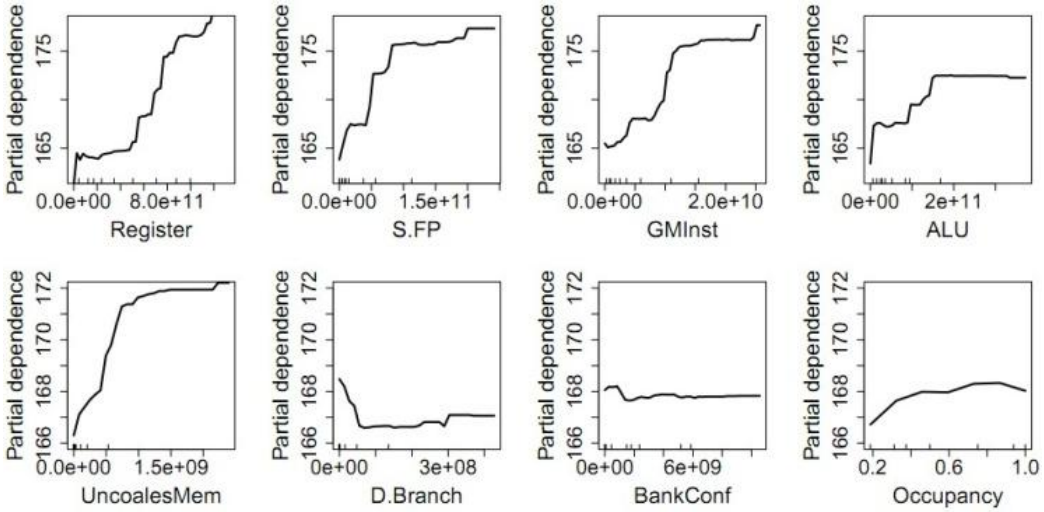


Figure 3-4. Partial dependence plots for five most influential and four architecture related parameters. The rug plots on inside bottom of plots show distribution of kernels across that variable, in deciles

For the architecture related parameters, we see that UncoalesMem shows a clear correlation that more power is consumed with more uncoalesced global memory accesses, which require more physical moves. D.Branch and BankConf are seemly on the opposite direction such that with higher volume, less power consumption is produced. This is generally true since divergent branch and bank conflict slow down the execution, cause inefficient use of the resources and result in lower power consumption. However, the overall power impact is more complicated. Due to small sample size, the trend is not very stable. Occupancy, on the other hand, shows more consistent results such that given higher occupancy, the more power

consumption is observed. Given the fact that the execution rate is generally faster by providing more overlapping CTAs to hide memory or computation latency, high occupancy consumes more power. But, the power drops a little with highest occupancy because there might be more than enough active warps in each SM to fill the pipeline, which may result in contention for on chip resources and slow down the execution [5].

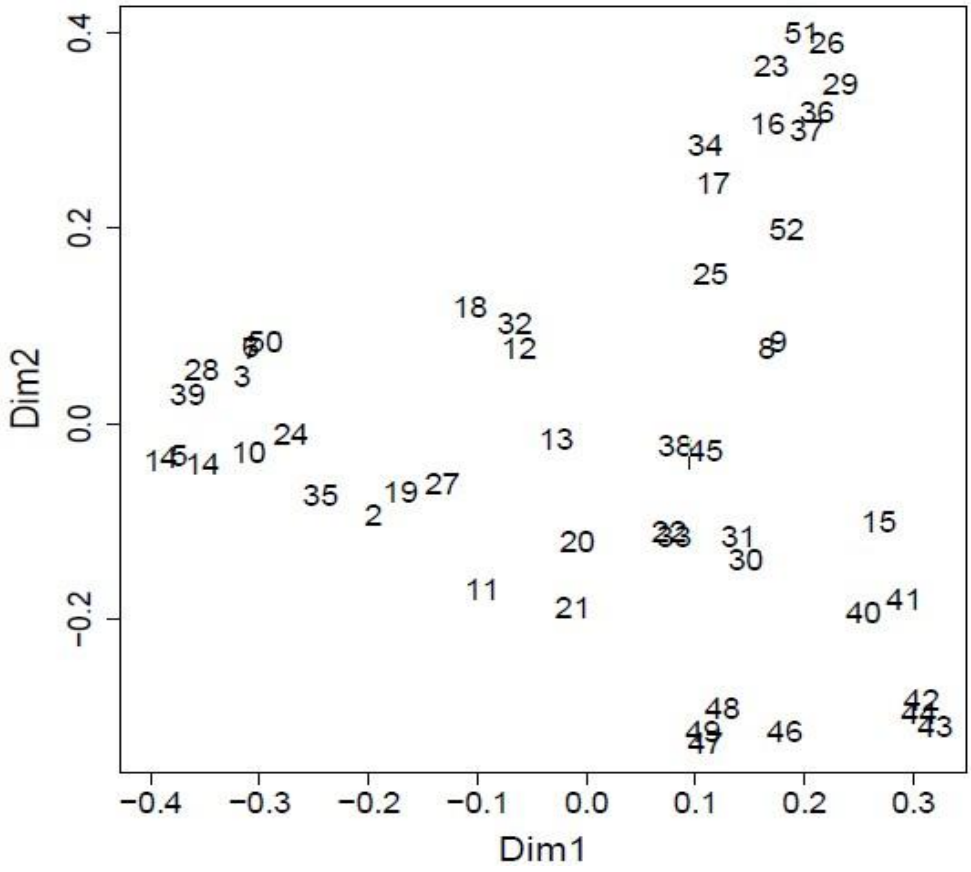


Figure 3-5. Proximity plot for the random forest model

Figure 3-5 shows the proximity plot which can help us filtering out kernels with similar power behavior to select a set of benchmarking kernels for future power studies. This plot gives an indication of which observations are effectively close together in the eyes of the random forest. Since an individual tree is unpruned, the terminal nodes (also called leaf) will contain only a small number of instances (i.e.

kernels). If kernel i and kernel j both land in the same terminal node, increase the proximity between i and j by one. At the end of the run, the proximities are divided by the number of trees and proximity between a case and itself set equal to one. Then the distance is calculated by one minus the proximity value. After that, the multidimensional scaling [18] is applied to the distance matrix. The key idea of the proximity plot is to approximate the original set of distances with distances corresponding to a configuration of points in a low-dimensional Euclidean space, so that the distances between the items in the low-dimensional space (e.g. 2D space in Figure 3-5) will be as close to the original distances as possible.

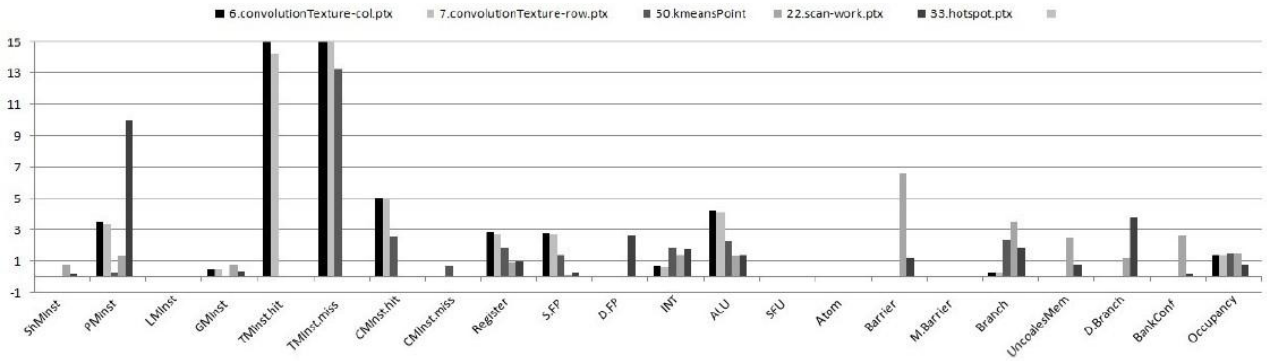


Figure 3-6. Normalized variables of selected kernels

In Figure 3-5, we can see that some kernels are very close to each other. For example, there are two sets of kernels; each consists of kernels with high similarity. The first set has kernels 6.convolutionColumnsKernel, 7.convolutionRowsKernel and 50.kmeansPoint while the second set has kernels 22.scanworkefficient and 33.hotspot. Figure 3-6 shows the value for the 5 kernel's metrics normalized to the average value of all the kernels. The first set of kernels have above average number of register, S.FP, GMInst and ALU which are the most important 4 variables in our model and much higher than average number of texture cache misses and constant

cache hits. The second set of kernels has close to average amount of register and ALU. They also have similar amount of GMinst and almost no S.FP instructions. Their power consumptions also follow this similarity. The first set has high measured power consumption of 207.3, 210.3 and 220.3 and the second set has 165.6 and 149.1 respectively.

### 3.6 Summary

Power consumptions in high-end GPUs require expensive power supplies and cooling systems. We present in this work a powerful GPU power modeling method to understand and predict the power consumption while running real application kernels. A sophisticated ensemble learning algorithm based on random forest analysis that combines many individual regression trees is used to establish the GPU power consumption model. Comparing with the linear regression or the regression tree approaches, our results show that the random forest model is not only more accurate in power prediction using a set of runtime performance metrics, it also provides sufficient insights for its users to understand the correlations between the GPU runtime power consumption and the individual performance metrics.

# CHAPTER 4
# GUIDED GPU SCHEDULING: UTILIZING MULTI-THREAD PARALLELISM TO HIDE MEMORY LATENCY

## 4.1. Introduction

Multithreading is a well-known technique to hide the instruction and memory latency. General-Purpose Graphics Processing Unit (GPGPU) explores application parallelism by building massive multithreading capacity in hardware for hiding long instruction and memory latencies. The traditional fine-granularity multithreading schedules a ready instruction from active threads in every cycle in a round-robin fashion. Once scheduled, the next instruction in the same thread will not be scheduled until all current ready instructions in other threads have been scheduled. With sufficient amount of threads (i.e. parallelism), the instruction and memory latencies can be hidden. The coarse-granularity multithreading (also known as *greedy* scheduling), on the other hand, continuously schedule ready instructions from the same thread until a stall (mainly due to data dependence) is encountered. It then switches to schedule ready instructions from the next thread also in a round-robin fashion to hide even longer latency.

In this work, we investigate thread scheduling algorithms in today's high-performance, highly-threaded GPGPUs [3], [46], which works as a co-processor to a host CPU through a PCI-Express bus. The host CPU executes application programs and initiates kernel calls to ship a program segment and the associated data to the GPU to exploit the parallelism in the program for running on massively parallel GPU hardware. Note that in this work we use the Nvidia Fermi-generation GTX-580 [46]

with compute capabilities 2.0 as a reference system. The methodology and studies can be applied to other GPGPUs.

The GPU scheduler has two levels, a global CTA scheduler (denoted as CTA-sch) and per-SM warp scheduler (denoted as Warp-sch). The CTA-sch controls the kernel execution by scheduling concurrent CTAs on each SM based on the resource requirement. Once scheduled, a CTA will run independently to the end before releasing the SM resources. The kernel completes when all CTAs finish execution. Each CTA has one or more parallel warps. All warps usually execute the same sequence of SIMT instructions to exploit data parallelism. In each SM, all warps in active CTAs compete for hardware resources. The Warp-sch schedules the next ready instruction selected from all active warps. If no ready instruction is available, a stall cycle is encountered and the scheduler will try in the next cycle.

The instructions in a warp have different latencies. A typical ALU instruction has latency in the low 10's of cycles while the latency for memory instruction has a wide range from 10's to several hundred cycles depending on where the data is located in the memory hierarchy. Since all active warps in an SM usually execute the same sequence of instructions, the traditional fine-granularity round-robin (*fine-RR*) and greedy round-robin (*greedy-RR*) scheduling algorithms are inefficient for instructions with wide ranges of latencies. Given sufficient number of active CTAs/warps, *fine-RR* and *greedy-RR* are overkill for short-latency instructions and waste valuable parallelism. On the other hand, when all warps wait for a global memory load with 700+ cycle latencies, neither of the round-robin scheduling

schemes can hide such a long-latency load with limited active warps and cause stalls in the SM.

In this work, we focus on the warp scheduler and its scheduling algorithms.[1] We evaluate two new warp scheduling techniques to effectively utilize the available warp parallelism in hiding the long-latency global memory loads. The basic idea is to save available parallelism for hiding short-latency instructions and use it to overlap with long-latency memory loads from other warps. The first technique allows *flexible* round-robin distances among active warps. The simple *oldest-first* scheduling scheme accomplishes the goal by always schedule a ready instruction starting from the oldest warp. During the execution in a *compute region* which consists of a sequence of short-latency instructions, the *oldest-first* schedules instructions from a subset of the active warps and saves the rest warps for hiding future long-latency instructions in this subset. The *oldest-first* scheme consumes minimum warps with short-latency instructions to reach to the long-latency global loads so that these loads can be overlapped with other warps remained in the compute regions. However, the *oldest-first* creates an *unfair* scheduling with warps progressing in different pace. It may prolong the completion of a CTA due to the *tailing* effect when a few warps take longer to finish. It may also affect the kernel ending especially for kernels with small number of CTAs.

[1] Note, we cannot obtain information about how warp scheduling works in Nvidia GPUs. We consider the traditional fine-granularity round-robin scheduling and the greedy scheduling mentioned in a recent paper [21]

The second technique permits flexible *priority-shift* among active warps to maintain fairness. In contrast to the *fine-RR* and *greedy-RR*, the flexible *priority-shift* technique shifts the priority for maximizing the overlap between compute instructions and global memory loads. One straight-forward approach is to shift the priority when the current warp encountered a stall due to long-latency load. Delaying priority shift till global loads can forward a few warps to the global loads faster to increase the chance of overlapping with other warps still executing compute instructions. This prioritized scheduling is similar to oldest-first but always starts from the warp with the priority.

However, our study based on several kernel programs reveals that each warp often executes a small number of global loads separated by few simple compute instructions and/or shared memory stores. When these global loads are closely together, shifting priority on every long-latency load leaves small *compute regions* which are insufficient to overlap with the subsequent global loads. Therefore, we consider a *memory-region* based priority-shift approach which shifts the scheduling priority after each memory region in which one or more global loads are separated only by a small number of compute instructions. Combining multiple global loads in a memory region allows these loads to be scheduled closely together to increase per-warp memory-level parallelism as well as to improve the chance for overlapping with other warps in the compute region. To capture the memory region dynamically at runtime, however, requires proper runtime history and complicates the hardware. In

this work, we evaluate a *software annotated* priority-shift hint to control the scheduling priority without increasing the hardware complexity.

For performance evaluations of the warp scheduling algorithms, we develop a cycle-based PTX instruction simulator to simulate kernel execution on a generic Fermi-like GPGPU architectures. We select twelve applications from CUDA SDK [49] and Rodinia benchmarks [12]. These kernels encounter high frequency of stall cycles in scheduling ready instructions mainly due to global memory accesses. The simulation results demonstrate significant performance improvement of the proposed scheduling algorithms. In comparison with the fine-granularity round-robin scheme, the pure oldest-first, the simple prioritized, and the region-based prioritized scheduling algorithms improve the kernel execution time by an average of 7.8%, 9.9% and 11.6% while the stall scheduling cycles decreased by about 34.5%, 46.0% and 56.1%, respectively. We also conduct sensitivity studies on the performance impact with respect to the amount of concurrent warps, the scheduling overhead, and the global memory latency. As expected, the results show that the stall cycle goes down with the increase of the concurrent warps and goes up with the global memory latency and the scheduling overhead. Among the scheduling algorithms, the region-based prioritized scheduling has better performance consistently.

This work makes two key contributions. First, we propose and evaluate a new GPU warp scheduling algorithm which permits *flexible* round-robin scheduling distance for efficiently utilizing multithread parallelism and use *program-guided* priority shift among active warps (threads) to maintain fairness. It provides better

overlaps between long-latency global loads and short-latency compute instructions and shows performance advantages over the traditional round-robin scheduling scheme. Second, we accomplish thorough evaluations of various warp scheduling algorithms. We also appraise the performance impact on various warp scheduling algorithms based on the amount of concurrent warps, the warp scheduling overhead, and the global memory latency. Chapter 4 is organized as follows. Section 4.2 summarizes key features of CUDA programming model and GPGPU architecture. To motivate this work, it also shows the performance degradation of using the traditional round-robin schedulers. Section 4.3 introduces the proposed program-guided warp scheduler. Section 4.4 describes the baseline GPU simulator. Performance evaluations are given in Section 4.5. Related work is in Section 4.6, followed by a conclusion in Section 4.7.

## 4.2. Related Work

There have been many researches on GPGPUs recently [14], [24], [51], [58], [64] but most of them are trying to optimize a certain application according to GPGPU's special architecture. There are also works to modify the architecture for better performance but few works are about improving warp scheduler.

Lee [33] proposed a Prefetching Mechanisms for the presence of many threads in-flight. They also propose the adaptive throttling to solve the problem that even with 100% accuracy, prefetching can still degrade performance in GPGPUs. The new prefetch instruction [43] introduced since the Fermi generation can overlap the compute region and the stall cycles in the same warp. However, it cannot overlap the

regions among different warps, which is very important since there are usually many concurrent warps on each SM. Tarjan [61] proposed a sharing tracker cache design that allows inter-SM cache sharing to capture inter-SM temporal locality and provides substantial reductions in off-chip bandwidth requirements. Fung [20] focused on the impact of branch divergence on GPGPU performance for general purpose applications. They found that significant performance improvements can be achieved with their proposed dynamic warp formation and scheduling mechanism. Meng [40] proposed dynamic warp subdivision to create warp-splits when the SIMD processor does not have sufficient warps to hide latency and leverage MLP. It allows threads in the same warp to interleave their execution in an asynchronous manner.

There have been many researches about task scheduling algorithms for various systems [31], [34], [35]. Li [35] focused on the efficiency of different tasks sharing the same GPGPU. They proposed a static scheduler to allow multiple kernels to run concurrently and showed performance improvements compared with sequential execution. However, they did not consider improving the performance of a single kernel with new scheduling algorithms. There are also works about thread scheduling among CPUs or hybrid systems. Lakshminarayana [31] proposed a new thread scheduling policy that is aware of the new asymmetry in AMPs and showed performance improvements for several benchmarks and workloads. Li [34] proposed a new thread scheduling algorithm that can achieve accurate fairness and high performance for a diverse set of workloads.

There have been several efforts to develop a GPGPU simulator. Barra system [16] is the only functional simulator to execute the actual binary code, however, the ISA changes with different generations of GPGPUs and it only supports the first generation, so it is not suitable to research Tesla, Fermi and later GPGPUs. GPGPUSim [5] is the only public cycle level simulator with PTX code as the input. This simulator is closely related to the actual GPGPU architecture and it now supports PTX2.0. GPUOcelot [19] by Gregory has a functional simulator that is compatible with the newest PTX version so it can be used to generate instruction and memory traces for further architecture research. MacSim [30] from GIT is a cycle level simulator driven by the trace from GPUOcelot but it is based on the older generations GPGPUs and it is not public.

### 4.3. SM Stall Cycles Using Traditional Round-Robin Scheduling

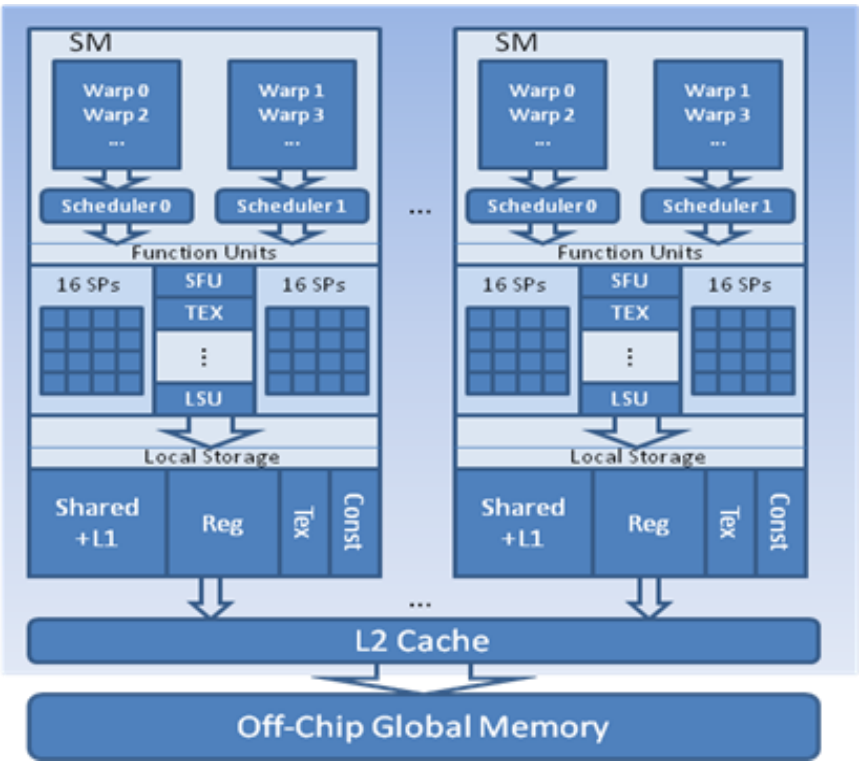


Figure 4-1. The baseline GPGPU architecture

For studying the impact of the warp scheduling algorithm, we consider a baseline GPGPU architecture depicted in Figure 4-1 which closely resembles the Nvidia GTX 580 architecture with compute capability 2.0. We use a set of common CUDA application kernels to drive the simulation. Due to the lack of the ISA of the Nvidia GPUs, we develop a cycle-based GPGPU simulation model based on the intermediate PTX instructions, which are commonly used in other GPGPU architecture studies [20][33].

In CUDA programs, all warps usually execute the same sequence of SIMT instructions to exploit data parallelism. In each SM, all warps in active CTAs compete for a scheduling slot and hardware resources. The Warp-sch schedules the next ready instruction selected from all active warps. In the baseline model, there are two parallel warp schedulers and instruction dispatch units, one for odd warps and the other for even warps in each SM. It takes 2 cycles to initiate pipeline execution of a ready instruction selected from the active warps. In case that there is no ready instruction available, a stall cycle is encountered and the scheduler will try again in the next cycle.

In this section, we demonstrate the challenges of using a fine-RR or a greedy-RR warp scheduling schemes to hide a wide range of instruction and memory latencies. In Figure 4-2, the percentages of stall cycles with respect to the total execution cycles of twelve application kernels selected from CDUA SDK and Rodinia benchmark suites are plotted. These results are obtained from running twelve application kernels on the baseline GPGPU simulator. Note that the stall cycles are

collected when a Warp-sch cannot find a ready instruction to schedule. We report the average stall cycles over the entire 16 SMs. The details of the simulator and the characteristics of the benchmarks will be given in Section 4-5 and 4-6. The results of twelve application kernels demonstrate a wide range of stall cycles from 5% to almost 80% of the total execution cycles with an average of 33% and 31% respectively for the fine-RR and the greedy-RR schemes. Nine kernels have 20% or more stall cycles. Such high stall percentage in warp scheduling underutilizes the SM hardware and degrades kernel performance. These high percentages of stall cycles motivate us to investigate better warp scheduling algorithms.

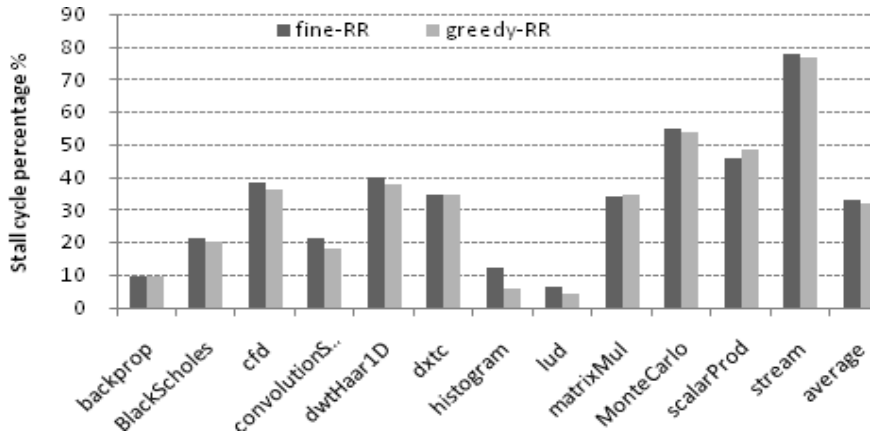


Figure 4-2. Stall cycle using *fine-RR* and *greedy-RR*

### 4.4. Program-Directed Warp Scheduler

The new warp scheduler consists of two key components, an *oldest-first* scheduling algorithm and a software-annotated region-based *priority-shift* mechanism. In this section, we use a convolutionSep kernel [49] to illustrate memory and compute regions in real applications. An abstract program model which consists of different memory and compute regions is used to illustrate warp execution under

different scheduling algorithms. Note that due to the lack of public ISA (Instruction Set Architecture) of Nvidia GPUs, our study is based on an intermediate PTX code which closely resembles the target ISA.

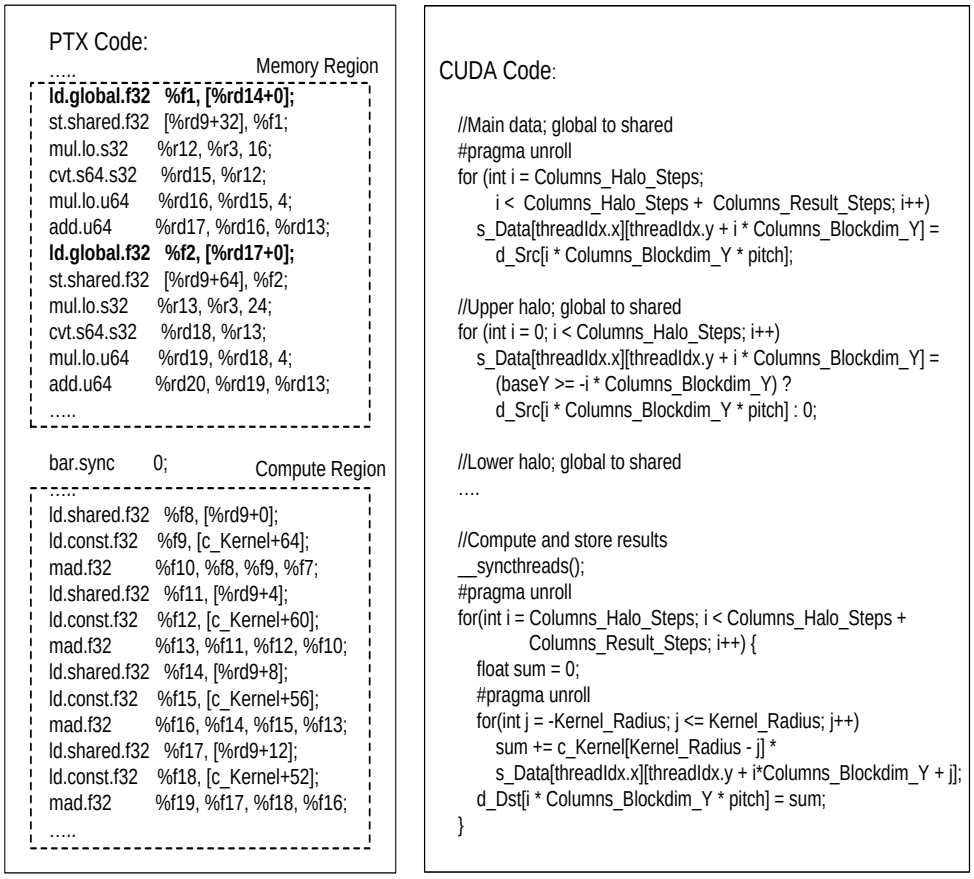


Figure 4-3. Simplified CUDA and PTX codes of convolutionSep

Figure 4-3 shows the source CUDA program and the PTX code of the convolutionSep kernel which has 530 PTX instructions. In this example, multiple threads (warps) in a CTA cooperatively move a block of data with its halos from global memory into the on-chip shared memory. Before each load, there are a few instructions to compute the load address. After 10 global loads, computations start using the data from the shared memory. There are over 400 short-latency PTX instructions in the compute region (partially shown in the figure). Every thread

executes the same instruction stream. This kernel has 9216 CTAs with 4 warps per CTA, so each of 16 SMs will execute a pair of memory region and compute region repeatedly for a total of 2304 warps. Note that such repeated executions of a memory region followed by a compute region indeed exist in most kernels.

Both *fine-RR* and *greedy-RR* perform poorly for this example. They exhaust all the parallelism in the compute region with hundreds of short-latency instructions and encounter long stalls in the memory region with a few global loads. The pure *oldest-first* scheduling scheme gives the priority to the oldest warp and the scheduler always picks the ready instruction starting from that warp. The main advantage of the oldest-first is its ability to break the fix round-robin distance of the entire active warps. It can dynamically adjust the needed number of warps to hide different instruction latencies. A noticeable disadvantage, however, is to treat the active warps unfairly and may cause starvation to the low priority warps.

To remedy the unfairness of the oldest-first scheduling, we include a mechanism to rotate the priority among active warps. The first approach is to follow the oldest-first to schedule a subset of the active warps through their compute region without stalls for fast reaching to the following memory region. While waiting for the return of the data from global memory, the priority is shifted to the next warp. The oldest-first is still applied from the warp which owns the priority. This simple priority-shift mechanism (denoted as *simple-p*) is similar to the *greedy-RR* scheduling except that the priority is shifted only when the warp is waiting for the long-latency global load. Although resolving the unfairness, this approach may waste valuable

parallelism when the length of compute regions exceeds the needed parallelism to cover the subsequent global load. Given two consecutive compute regions, one with excessive length and the other with inadequate length, the excessive region cannot be saved to cover the following insufficient compute region for hiding the latency of the subsequent global memory load.

The second approach relies on compiler or programmer to insert software-annotated priority-shift hints. When a warp executes a priority-shift hint, the priority is shifted to the next warp. In contrast to the *simple-p* where the priority is shifted to the next warp when the warp is waiting for a global load, the software-annotated priority-shift approach shifts the priority after a memory region. Recall that each memory region consists of one or more global loads separated by a small number of short-latency compute and/or shared-memory instructions. Such a small compute region is insufficient to overlap with the following global load, hence can be combined with adjacent global load to delay the priority shift. Based on the knowledge of the positions of the global loads, the programmer/compiler can decide a memory region and insert the priority-shift. We refer this approach as a program-directed, region-based priority-shift scheduling scheme, or simply *region-p.*

In Figure 4-4, we illustrate the execution of 4 groups of warps under three scheduling algorithms, *oldest-first*, *simple-p*, and *region-p*. In this figure the rectangular represents a compute region, the double-edge arrow indicates a global memory load, and "S→" denotes a priority-shift. The length of the rectangular and the arrow defines the execution time.

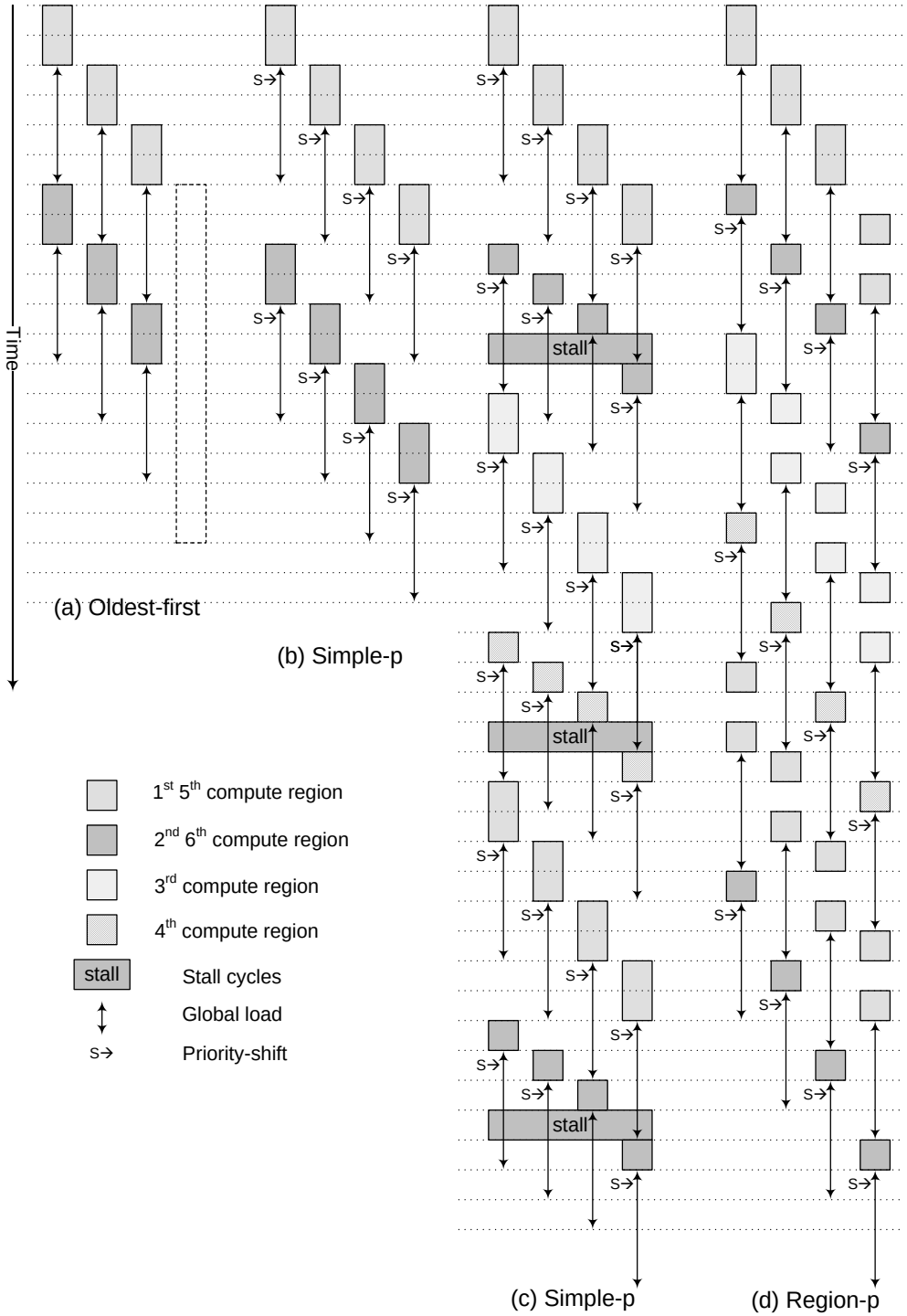


Figure 4-4. Comparison of three scheduling algorithms: *oldest-first, simple-p,* and *region-p*

In this abstract example, each warp group has sufficient parallelism to cover short-latency instructions. For simplicity, we omit the details inside each group. Part (a) shows that *oldest-first* indeed breaks the long round-robin distance and overlaps

the global loads with the compute regions for two compute/memory regions. However, since the 4$^{th}$ group is not required in hiding the latency of the global load, *oldest-first* starves the 4$^{th}$ group. Part (b) demonstrates the fairness among the four groups with *simple-p* for the two compute/memory regions. The global load latencies are also hidden completely. With excessive short-latency instructions in the compute region, delay is encountered between the two compute/memory regions without harming the overall performance.

Next, in order to demonstrate the power of *region-p*, we simulate 6 consecutive compute/memory regions which consist of a region with excessive compute instructions followed by a region with insufficient instructions. Such behavior repeats for 6 regions. *Simple-p* wastes the parallelism in region 1, 3 and 5 and encounters stall cycles in region 2, 4 and 6 as illustrated in Part (c). In *region-p*, the programmer/compiler combines region 1 and 2, 3 and 4 as well as 5 and 6 for inserting the priority-shift due to the small compute region 2, 4 and 6. As a result, the excessive parallelism in region 1, 3 and 5 can be shared with region 2, 4 and 6 respectively to completely overlap with the global loads as demonstrated in Part (d).

In our study, we insert the priority-shift hints into the kernels for simulating *region-p* as following. To identify a memory region, we first estimate the threshold $l / w$ of hiding the memory latency, where $l$ is the off chip global memory access latency and $w$ is the number of active warps. We define a sequence of memory loads are in the same memory region if they satisfy the following two conditions. First, the accumulated latency from a memory load to its dependent instruction is shorter than

*l* / *w*. Second, the distance between adjacent load-use pair is also shorter than *l* / *w*. The above two conditions identify the loads closely together and lack of sufficient compute instructions to hide the latency. The priority-shift hint is then inserted after the last load in a memory region.

### 4.5 GPGPU Timing Simulator

In order to evaluate the performance, we develop a trace driven cycle level timing simulator for Nvidia's GPU (t*GPU-sim*). *tGPU-sim* simulates the execution cycles of CUDA kernels on GPUs. To reduce the development efforts, *tGPU-sim* incorporates a *tracer* which leverages the existing GPU function-level simulator *Ocelot* to generate pre-decoded PTX execution traces. *tGPU-sim* consists of three major components, a CTA and warp scheduling unit (*scheduler*), a cycle-based instruction execution model (*inst-sim*) and a cycle-accurate memory hierarchy model (*mem-sim*). The basic functions of these components are described in the following. Detailed descriptions are given in subsequent sections.

**Ocelot-Tracer:** *Ocelot* interprets the PTX code and emulates kernel execution at the functional level. The tracer dumps the PTX execution traces from Ocelot for cycle-based timing simulation of the kernels. To help the efficiency of simulation, pre-decoded PTX instructions including the instruction opcode classified into categories, the source and destination operand register IDs, and memory addresses are collected from Ocelot to avoid duplicating the PTX decoder in the timing simulator.

**Scheduler**: The scheduler follows the Fermi GPU with compute capability 2.0 architecture. It will have two both levels, a CTA scheduler (C*TA-sch*) and a Warp

scheduler (*Warp-sch*). At the higher level, the C*TA-sch* schedules a CTA on a SM as long as the SM has enough resources, usually released by another CTA upon completion. All active warps in a SM compete with each other for hardware resources. The lower level *Warp-sch* selects the next ready-to-execute instruction from all active warps. We simulate two parallel schedulers and instruction dispatchers, one for odd warps and the other for even warps. It takes 2 cycles to schedule a read-to-execute instruction selected from all active warps, hence one instruction can be scheduled per cycle if there are sufficient warps and hardware resources.

*tGPU-sim* simulates PTX instruction execution on the processing units in each SM based on the SIMT execution model. Each SM has a few processing units for executing a PTX instruction from a warp. Each arithmetic instruction or memory access takes multiple cycles to execute. To keep track of the data dependences, a *scoreboard* for each active warp is implemented to record the status of the instruction execution and the availability of the hardware resources. In each simulation cycle, the scheduler updates and maintains the scoreboards and selects the next ready-to-execute instruction from all active warps.

**Inst-sim**: The *inst-sim* simulates instruction executions. Based on the Fermi architecture with compute capability 2.0, each SM has two sets of general-purpose both ALU (for integer and logical operations) and FPU (for floating point operations) cores, one set of special function units (SFU), and one set of load-store units (LSU). Each ALU, FPU and LSU unit consists of 16 processing cores while the SFU has 4

processing cores to execute one instruction from 32 threads in a SIMT fashion. All units are pipelined and the throughputs are 2 and 8 respectively for the units with 16 and 4 processing cores. The LSU executes loads and stores and initiate memory accesses. Each memory request goes through different memory hierarchies based on the cycle-accurate *mem-sim* model.

**Mem-sim**: The memory hierarchy system consists of a variety of memories including registers, shared memory, various L1 caches, L2 caches, global memory, local memory, constant memory, and texture memory which resemble the memory organization of Nvidia's Fermi architecture. Programmers partition the on-chip per-SM storage into programmer-controlled shared memory and hardware-managed data L1 caches. L2 cache is shared by all SMs. All accesses to the global, local, constant, and texture memories go through the L2 cache. Constant and texture memories have their own per-SM L1 cache. Note that since data L1 is private in each SM, it encounters cache coherence problem. Nvidia keeps the coherence protocol simple. Whenever a write to the data L1 cache, the target line is invalidated and the write is write-through to L2, except for updates to the local memory which uses write-back scheme. The shared-memory coherency is handled by the programmer. Writes to constant and texture memory are not allowed. Memory access through different levels of memories takes different delays. The *Mem-sim* simulates cycle-accurate delays for accessing the respective memory.

### 4.5.1 Ocelot Tracer

The tracer generates two trace files: a uniform kernel PTX execution trace (*kernal-trace*) and an individual warp execution trace (*warp-trace*). The global kernel-trace consists of a complete sequence of PTX instructions for the entire kernel while the *warp-trace* personalizes individual warp execution traces. Since the warps likely execute a similar set of PTX instructions, collecting a global uniform kernel trace greatly reduces the trace size. The individual *warp-trace*, on the other hand, records the execution sequence of each individual warp. To simplify the trace, the PC is used for recording the PTX instruction in the *warp-trace*. The pre-decoded information can be retrieved from the global *kernel-trace*. For memory instructions, on the other hand, individual memory reference of the same instruction for each thread in a warp is unique and these addresses may vary among different execution instances of the same instruction. Therefore, they must be recorded in the *warp-trace*.

The PTX instruction trace in the *kernal-trace* includes the original source PTX code and needed pre-decoded information. The pre-decoded information is used by the t*GPU-sim* to avoid duplicating the decoder. Since the goal of this work is to study memory performance, we collect the needed pre-decoded information especially for memory references. The pre-decoded PTX information includes the instruction types and attributes as well as the source / destination operand information. To simplify the instruction simulation, we separate the PTX instructions into 23 categories as shown in Table 4-1. We simplify the PTX instruction execution by simulating instructions in the same category with the same latency and initiation delay on the

same hardware unit. For our simulation, we take the initiation delay and execution latency from [49] as shown in Table 4-1.

The operands can be either register or memory. For register operand, a pre-decoded ID is recorded. For memory operands, the operand size attributes .vec and .type is also recorded in the kernel-trace. The individual memory addresses of all 32 threads are calculated from Ocelot and recorded in the warp-trace to allow different addresses among different execution instance of the same instruction.

Table 4-1. PTX instruction category, initiation and execution cycles

| Category | Hardware Unit | Through-put | Execution Latency | PTX instructions |
|---|---|---|---|---|
| Integer | ALU | 2 | 24 | add, sub, add.cc, addc, sub.cc, subc, mul, mad, mul24, mad24, sad, div, rem, abs, neg, min, max, popc, clz, bfind, brev, bfe, bfi, prmt, mov<br>Note, these integer inst. with *type*={ .u16, .u32, .u64, .s16, .s32, .s64 }; |
| Float_single | ALU | 2 | 24 | testp, copysign, add, sub, mul, fma, mad, div, abs, neg, min, max<br>Note, these Float-single inst. with *type* = { .f32 }; |
| Float_double | ALU | 1 | 48 | testp, copysign, add, sub, mul, fma, mad, div, abs, neg, min, max<br>Note, these Float-double inst. with *type* = { .f64 }; |
| Special_single | SFU | 8 | 48 | rcp, sqrt, rsqrt, sin, cos, lg2, ex2<br>Note, these special-single with *type* = { .f32 }; |
| Special_double | SFU | 8 | 72 | rcp, sqrt, rsqrt, sin, cos, lg2, ex2<br>Note, these special-double with *type* = { .f64 }; |
| Logical | ALU | 2 | 24 | and, or, xor, not, cnot, shl, shr |
| Control | ALU | 2 | 24 | bra, call, ret, exit |
| Synchronization | ALU | 2 | 24 | bar, member, vote |
| Compare & Select | ALU | 2 | 24 | set, setp, selp, slct |
| Conversion | ALU | 2 | 24 | Isspacep, cvta, cvt |
| Miscellanies | ALU | 2 | 24 | brkpt, pmevent, trap |
| Video | ALU | 2 | 24 | vadd, vsub, vabsdiff, vmin, vmax, vshl, vshr, vmad, vset |
| Load_shared | LSU | 2 | 30 | ld, ldu<br>Note, .ss = .shared ; .vec and .type determine the size of load. Note also that we omit .cop since no cacheable in Ocelot |

Table 4-1. Continued

| Category | Hardware Unit | Through-put | Execution Latency | PTX instructions |
|---|---|---|---|---|
| Load_global | LSU | 2 | 600 | ld, ldu, prefetch, prefetchu<br>Note, .ss = .global; .vec and .type determine the size of load. Note, Ocelot may not generate prefetch since no caches |
| Load_local | LSU | 2 | 600 | ld, ldu, prefetch, prefetchu<br>Note, .ss = .local; .vec and .type determine the size of load. Note, Ocelot may not generate prefetch since no caches |
| Load_const | LSU | 2 | 600 | ld, ldu<br>Note, .ss = .const; .vec and .type determine the size of load |
| Load_param | LSU | 2 | 30 | ld, ldu<br>Note, .ss = .param; .vec and .type determine the size of load |
| Store_shared | LSU | 2 | 30 | st<br>Note, .ss = .shared; .vec and .type determine the size of store |
| Store_global | LSU | 2 | 600 | st<br>Note, .ss = .global; .vec and .type determine the size of store |
| Store_local | LSU | 2 | 600 | st<br>Note, .ss = .local; .vec and .type determine the size of store |
| Read_modify_write_shared | LSU | 2 | 600 | atom, red<br>Note, .space = shared; .type determine the size |
| Read_modify_write_global | LSU | 2 | 600 | atom, red<br>Note, .space = global; .type determine the size |
| Texture | LSU | 2 | 600 | tex, txq, suld, sust, sured, suq |

The *warp-trace* records execution traces of individual warps. In CUDA computation model, each kernel has a number of CTAs and each CTA has a number of warps. The *warp-trace* omits the CTA level and records all individual warp traces with a unique warp ID. In the *warp-trace* file, each warp trace consists of a sequence of PCs of the PTX instructions. The pre-decoded information can then be obtained from the *kernel-trace* using the PC. In case of a memory instruction, the *mem-sim* can retrieve the 32 memory addresses stored sequentially in the *warp-trace* followed the PC of the instruction.

### 4.5.2 The Scoreboard and Scheduler

*tGPU-sim* has two schedulers, *CTA-sch* and *Warp-sch*. *CTA-sch* controls kernel execution. It assigns a CTA to run on a SM based on the CTA resource requirement and the available hardware resources in each SM. At the beginning of timing simulation, *CTA-sch* reads the simulated GPU architecture configuration and the executed kernel information from config. The *CTA-sch* uses a simple round-robin scheduling scheme to assign a CTA on a SM in a round-robin fashion until all SMs are full or all the CTAs are exhausted. When a CTA is completed, the *CTA-sch* is called to assign the next CTA to the SM if sufficient hardware resources are released. The kernel completes when all CTAs finish execution.

Once a CTA is scheduled on a SM, all warps in the CTA are active. A scoreboard is initialized for each active warp to keep track of the instruction status and to enforce correct data dependences. In each cycle, the *Warp-sch* selects the next ready instruction from all active warps to be executed on the available hardware units. An instruction is ready to be scheduled when all its operands are ready. An instruction, once scheduled, will take multiple cycles to finish. The hardware unit and the execution cycles for different instruction categories are given in Table 4-1. When an instruction finishes, the scoreboard is updated and the hardware unit is released. GPU follows in-order execution model. The instruction is always fetched and scheduled sequentially in each warp. When the current instruction of a warp is not ready, the *Warp-sch* moves to schedule instructions in other warps.

Before the start of simulation, *tGPU-sim* reads the simulated GPU architecture configuration and the executed kernel information from a configuration file (*tGPU.config*). *tGPU.config* consists of the simulated GPU architecture configuration including the memory hierarchy organization, latency, bandwidth, and the simulated instruction type, execution latency, as well as the characteristics of the running kernel including the number of CTAs, warps, and the resource requirement, etc. This information is necessary for the simulator to simulate the timing of kernel execution.

### 4.5.3 The Instruction Simulator

The *inst-sim* simulates PTX instruction executions according to the PTX2.1 ISA [49]. Since the goal of this work is to study cache performance under the impact of various scheduling algorithms, we simplify PTX instruction execution. We group PTX instructions into 23 categories as shown in Table 4-1. Each category of instructions has its latency and throughput on the same hardware unit. Note that since the instruction sequence is from the execution order in *Ocelot*, the timing simulator only simulates the timing information without executing the instruction. In our simulation, we take the throughput and execution latency from CUDA programming guide 3.2 and [65]. We also assume instruction can be fetched from an ideal instruction cache with zero cycle latency and infinite bandwidth.

### 4.5.4 The Memory Hierarchy Simulator

Based on Fermi architecture, we simulate complex memory hierarchy using cycle-accurate model. In our simulator, we assume that tens of cycles latency of instructions can cover the register access latency so we don't simulate register

accesses. We simulated the on chip memory access to shared memory and L1, constant and texture caches. The shared memory has 32 banks and each bank can allow 1 access per cycle. So the access latency of a shared memory instruction is the shared memory latency (30) plus bank conflicts cycles. The L1, constant and texture caches will share a single MSHR with 64 entries.

Table 4-2. Memory hierarchy and configurations

| | Size | Bandwidth | Latency | Throughput | MSHR | Comment |
|---|---|---|---|---|---|---|
| Register | 32K | Infinite | 0 | -- | -- | |
| Shared memory | 16 or 48KB | 32 banks, 8-byte interlev. | 30 | 32 | -- | |
| Data L1 | 16 or 48KB | 1 128b block / cycle | 30 | 1 | 64 shared | 4-way LRU |
| L2 | 768KB | 8 banks /gpu<br>128b block interlev. | 100 | 8 | 128 | Shared, w-back<br>64-way LRU<br>For all mem |
| Global memory | 6GB | 8 channels/gpu | 600 | 8 | -- | Through L1/L2 |
| Local memory | 6GB | 8 channels/gpu | 600 | 0 | -- | Through L1/L2 |
| Constant/ Texture memory | 64K/6GB | 8 channels/gpu | 600 | 0 | -- | Separate C-L1 and T-L1 cache<br>Through L2 |
| Constant, Texture L1 | 12K | 1 128b block / cycle | 30 | 1 | 64 shared | 4-way LRU |

All on chip cache access will take 30 cycles and each cache can take 1 cache block access per cycle. We also simulate L2 in 8 banks. Each bank can allow 1 cache block access per cycle and can issue 1 cache block access to off chip memory per cycle. L2 also has a MSHR with 128 entries. We limit the off chip

memory bandwidth to 8 cache blocks per cycle and fix the latency to be 600 cycles. The detailed simulated memory hierarchy configuration and parameters is given in Table 4-2.

## 4.6 Performance Evaluation

Table 4-3. Characteristics of the 12 kernels from CUDA SDK and Rodinia

| Kernel | Total CTAs | Warps/ CTA | CTAs/ SM | Warps/ SM | Schedule group/SM | # PTX inst. | # global loads | Memory region |
|---|---|---|---|---|---|---|---|---|
| backprop | 8192 | 8 | 6 | 48 | 86 | 103 | 2 | 2 |
| BlackScholes | 480 | 4 | 5 | 20 | 6 | 129 | 3 | 1 |
| cfd | 1212 | 6 | 2 | 12 | 38 | 1041 | 36 | 5 |
| convolutionSep | 9216 | 4 | 8 | 32 | 72 | 530 | 10 | 1 |
| dwtHaar1D | 256 | 16 | 3 | 48 | 6 | 115 | 2 | 1 |
| dxtc | 768 | 2 | 8 | 16 | 6 | 952 | 2 | 2 |
| histogram | 240 | 6 | 8 | 48 | 2 | 148 | 1 | 1 |
| lud | 15876 | 8 | 6 | 48 | 166 | 102 | 3 | 3 |
| matrixMul | 1500 | 8 | 4 | 32 | 24 | 128 | 2 | 1 |
| MonteCarlo | 256 | 8 | 5 | 40 | 4 | 191 | 1 | 1 |
| scalarProd | 96 | 8 | 6 | 48 | 1 | 104 | 1 | 1 |
| stream | 96 | 16 | 3 | 48 | 2 | 113 | 8 | 2 |

In this section, we present the simulation results including the execution cycles and the stall scheduling cycles of the five scheduling algorithms, *fine-RR, greedy-RR, oldest-first, simple-p,* and *region-p*. We also show performance sensitivity studies of the scheduling algorithms with respect to the hardware multithreading capacity, the scheduling overheads, and the global memory access latency. Table 4-3 summarizes the characteristics of the twelve application kernels selected from CUDA SDK and Rodinia benchmarks. The description of these kernels can be found in their websites. We select the kernels with sufficient number of CTAs to fully utilize all 16 SMs. In order to compare different scheduling algorithms, the selected kernels also

demonstrate a significant amount of stall cycles for schedulers to schedule ready instructions. In the table, “CTAs/SM” represents the number of CTAs that can be scheduled on each SM concurrently. With 16 SMs, “Schedule group/SM” indicates the number of schedule groups of “CTAs/SM” that each SM needs to execute. Finally, “Memory region” shows the number of memory regions in each kernel in which one or more global loads are grouped into for inserting the priority-shift hint.

### 4.6.1 Execution and Stall Cycles

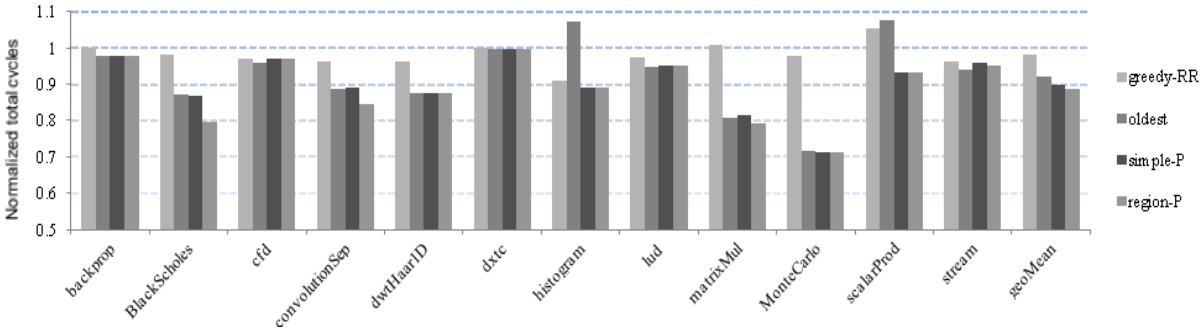


Figure 4-5. Normalized total cycles of 12 kernels

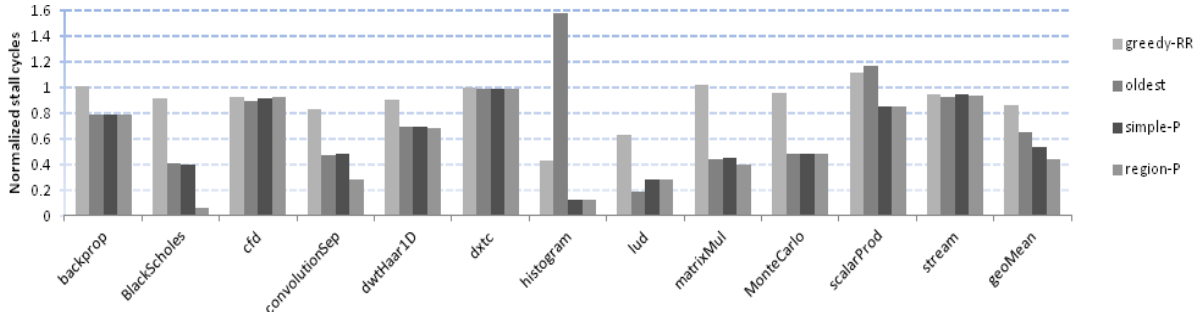


Figure 4-6. Normalized stall cycles of 12 kernels

The normalized execution cycles of *greedy-RR, oldest-first, simple-p,* and *region-p* with respect to *fine-RR* are plotted in Figure 4-5. We can make several important observations. First of all, we observe that in comparison with *fine-RR*, the average (geometric mean) execution times of the twelve kernels are improved by

2.0%, 7.8%, 9.9%, and 11.6% respectively for the four scheduling algorithms. Since each kernel has fixed number of instructions to execute, the improvements come mainly from the decrease of the scheduling stall cycles. Figure 4-6 shows that compared with *fine-RR,* the stall cycles are reduced by 13.3%, 34.5%, 46.0% and 56.1% respectively for the four scheduling algorithms. These results demonstrate the effectiveness of reducing the stall cycles from the flexible round-robin scheduling distance and the priority-shift mechanisms. Since the performance improvement varies heavily from individual workloads, we will discuss the insight of performance improvement based on individual workloads.

Among the scheduling algorithms, *Region-p* has superior performance over other scheduling methods for BlackScholes, convolutionSep, and matrixMul. In order to understand the benefits of *region-p*, we take a close look at the convolutionSep kernel (Figure 4-3). There are 9 global loads separated by 6 instructions and one separated by 15 instructions. Such small number of instructions is not enough to hide each global load. *Region-p* combines these 10 global loads into one memory region and switch priority after the last global load. As a result, the subsequent compute region with 400+ short-latency instructions can be used to overlap with the next memory region. Such a behavior is also found in BlackSholes, matrixMul, and stream with 3, 2, and 4 global loads in each memory region.

Next, *Simple-p* shows comparable performance with *region-p* for other application kernels. Obviously, for workloads with the same global loads and memory regions (Table 4-3), there is no difference between *simple-p* and *region-p*.

However, with 36 global loads in 5 global regions in cfd, *region-p* does not show any benefit. This is due the fact that cfd only has 12 warps on each SM. With 2 schedulers in each SM for odd and even warps, only 6 warps can be used by each scheduler to hide the latency. In this case, the scheduler needs to execute all warps to cover even the short-latency instructions, hence no benefit for *region-p*. Stream is different. Although it combines 8 global loads into 2 memory regions, the compute region is very small. Therefore, all scheduling schemes show similar performance for this memory-bound kernel.

All three scheduling algorithms: *Oldest-first*, *simple-p*, and *region-p* have the benefit of flexible round-robin distance to fit the instruction latency. However, o*ldest-first* performs poorly for some workloads due to its unfair scheduling among active warps and the penalty in kernel ending phase. This is evident for workloads with small number of scheduling group such as histogram and scalarProd. Table 4-3 shows that these two kernels have relatively small number of CTAs with only 1 or 2 scheduling groups. The results show that compared with the baseline *fine-RR*, *oldest-first* generates 58% and 17% more stall cycles respectively for histogram and scalarProd. On the contrary, *simple-P* and *region-P* can reduce the stall cycles by 88% and 15% because they not only overlap the compute regions and memory regions of different warps to reduce stall cycle, they also shift priority among all warps to keep fairness so all warps can finish at roughly the same time.

### 4.6.2 Sensitivity Studies

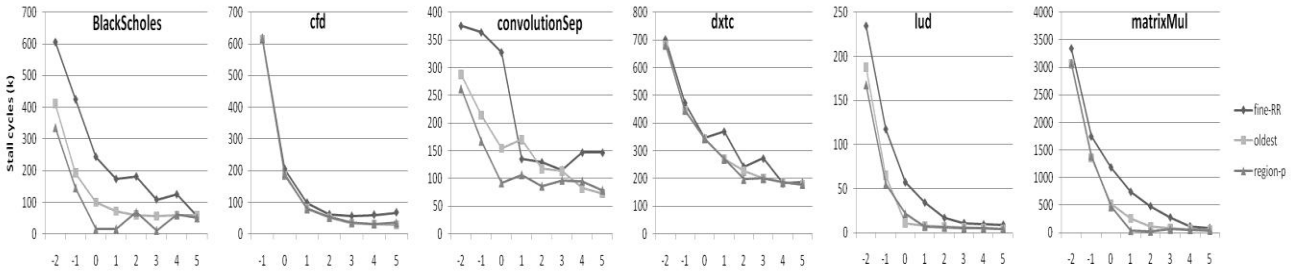


Figure 4-7. Stall cycles with different number of CTAs per SM

The ability to hide the long-latency global loads depends on the available software parallelism as well as the capacity of the active hardware warps. In Figure 4-7, we show the impact on stall cycles when varying the number of active CTAs in each SM. In this simulation, we assume the active CTAs/warps in each SM can be increased without resource constraints. To simulate a wide range of active CTAs, we select six kernels which have large number of CTAs. In this figure, the *x*-axis represent the number of active CTAs ranging from *c*-1 to *c*+5, where *c* is the number of CTAs under the resource constraints as shown in Table 1-1, and the *y*-axis is the stall cycles throughout the kernel execution. Note that we omit *greedy-RR* and *simple-p* in the sensitivity studies since their results are similar to *fine-RR* and *region-p* respectively.

The results indicate that increasing concurrent warps can significantly reduce the stall cycles for the three scheduling algorithms in all six kernels. However, when there are only few stall cycles left, adding more parallelism will no longer help. Among the three scheduling algorithms, *region-p* has the least amount of stall cycles followed by *oldest-first* and *fine-RR* has the most. Another observation is that *region-*

*p* can achieve the minimal stall cycles with fewest CTAs. With the resource constraint, *region-p* can reach the minimum stall cycles for BlackScholes, convolutionSep and lud. On the other hand, increasing the active CTAs beyond the resource constraint is beneficial for cfd, dxtc, and matrixMul. Note that there is a slight inconsistency in stall cycles when increasing the number of active CTAs. For example, BlackScholes has 480 CTAs and each SM executes 30 CTAs. Under the hardware constraints, each SM can have 5 concurrent CTAs so that 30 CTAs can be divided into 6 scheduling groups. There are 5 groups if the active CTAs increase to 6. However, when the active CTAs become 7, there are still 5 groups with 2 CTAs in the last group. Therefore, 7 concurrent CTAs do not help much. With small CTAs in the last group, more stall cycles may occur due to less parallelism to explore. Cache performance presents another undetermined factor. As the number of CTAs per SM changes, L1 and L2 cache performance is also affected.

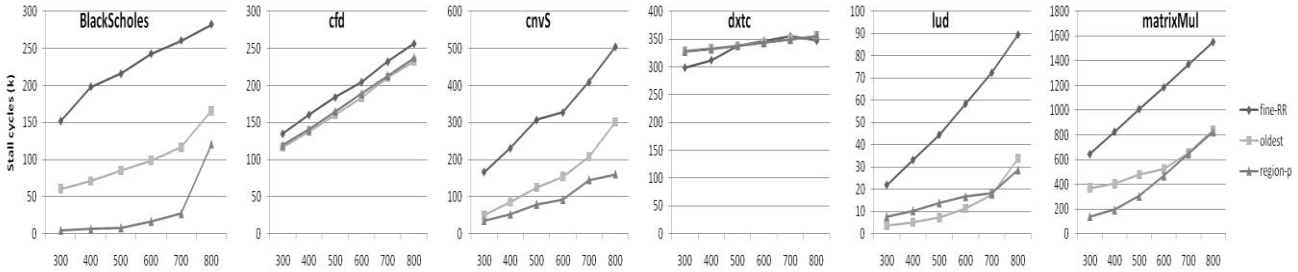


Figure 4-8. Stall cycles with different global memory latency

Global memory latency plays an important role in the kernel execution time and the scheduling stall cycles. In general, the longer the latency is, the lengthier the execution time will be due to more stall cycles in waiting for the data from global memory. Figure 4-8 plots the results of the stall cycles for *fine-RR, oldest-first*, and

*region-p* with respect to off-chip DRAM latencies ranging from 300 to 800 cycles. The stall cycles increase with the latency for all three scheduling algorithms. However, the stall cycles of *region-p* and *oldest-first* increase slower than that of *fine-RR* because of their ability to overlap compute and memory regions to reduce memory latency stalls. The above observation is true for all kernels except for dxtc in which the stall cycles are rather insensitive with the latency. Detailed analysis reveals that dxtc has small active warps (16) and the stall cycles are mainly due to tight compute dependences. This is also evident from the little difference in stall cycles among the three scheduling algorithms. Kernel cfd also has small number of active warps (12) which is insufficient to hide the compute instruction latency. However, the stalls due to memory latency play a heavier role than that in dxtc.

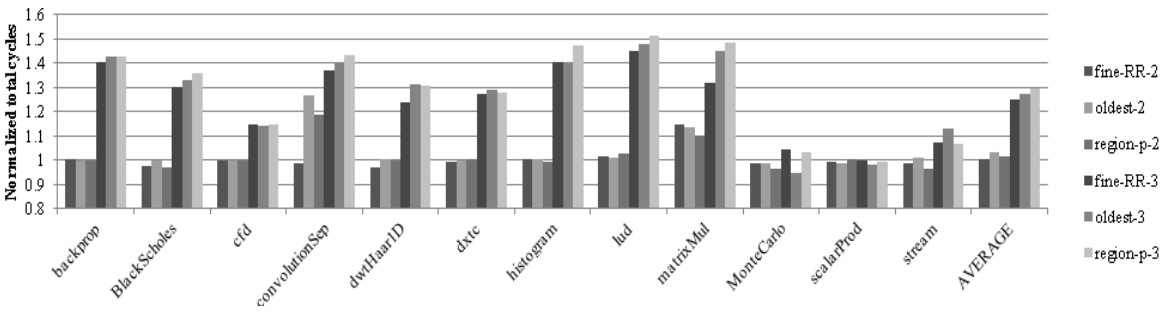


Figure 4-9. Normalized stall cycles with different scheduler overhead

The scheduler overhead also impacts the overall performance. As described in section 4, it takes 2 cycles to initiate a ready instruction to the SP and Load/Store function units, and 8 cycles to the SFU due to the amount of hardware units. However, each *Warp-sch* can try to schedule a ready instruction in the following cycle in case no ready instruction is found in the previous cycle. It is expensive to search and schedule a ready instruction in one cycle. In order to see the impact of

the scheduling overhead, we experiment with scheduling overhead of 2 and 3 cycles. Figure 4-9 shows the total cycles of the three schedulers normalized to their baseline configuration with 1 cycle scheduling overhead. We can see that changing the overhead to 2 cycles has little effect on the total cycles because the hardware units can only accept an instruction in every 2 cycles. However, if we increase the overhead to 3 cycles, it significantly increases the total cycles by about 25-30% on the average. It is essential to keep the scheduling overhead low to match the pipeline initiation delay. Among the three scheduling algorithms, *region-p* is less sensitive with 2-cycle scheduling, but slightly more sensitive with 3-cycle scheduling.

## 4.7 Summary

GPGPU becomes increasingly popular for general-purpose parallel computation. It explores parallelism in applications and relies on massive multithreading capacity to efficiently utilize the many-core hardware for achieving high performance. In this work, we study the critical thread (warp) scheduling algorithms in highly-threaded GPGPUs. We propose and evaluate a new warp scheduling algorithm which permits flexible round-robin scheduling with program-guided priority shift. The performance evaluation results demonstrate significant performance improvement over the traditional round-robin scheduling. We also evaluate the performance impact on the hardware multithreading capacity, the global memory latency, and the scheduler overhead. The results provide a thorough understanding of the performance behavior for various warp scheduling algorithms which help improving future GPGPU designs.

# CHAPTER 5
# DISSERTATION CONCLUSIONS

GPGPU has been proven to be very powerful and effective in many areas, especially scientific computing area. Programmers have reported speedup up to hundreds of times for different programs [45]. However, although GPUs has an excellent GFlops and memory bandwidth, it has several important constraints and concerns like the inability of in-kernel global synchronization, high power consumption and inefficient warp scheduling in each SM.

The first work focuses on the significant overhead to accomplish global synchronization and thoroughly study and analyze the inter CTA communication and its impact on an important class of applications based on chaotic relaxation of PDE solvers. Performance evaluation shows a speedup of 4-5 times of the two applications compared with the host synchronization implementation.

The second work is to compose a GPU power model to understand and predict the power consumption of applications. In this work, we use GPGPUSim to collect detailed information of kernels from real applications and use a sophisticated ensemble learning algorithm: random forest to compose the model and do the analysis. The evaluation shows that this model has high accuracy and is able to provide many insights to understand GPU's power consumption.

The third work proposes innovative warp scheduling schemes in the SM. Although GPU has massively parallel hardware, traditional warp scheduler will result in poor performance because of a lot of stall cycles due to the waste of parallelism. Our scheduler can stager the compute and memory access regions so that they can be overlapped and hence the stall cycles will be greatly reduced, which results in a much

better performance. Our scheduler can also keep the fairness among different CTAs and warps so that there won't be any tailing effect even if there are limited number of CTAs in the kernel.

LIST OF REFERENCES


[1] T.M. Aamodt, A. Bakhoda, W. Fung, "Tutorial on gpgpu-sim: A performance simulator for massively multithreaded processor research," *Tutorial 42nd International Symposium on Microarchitecture,* 2009.

[2] Appro, "Appro 1u tetra gpu server - 1426g4," http://www.appro.com/product/1426G4server_overview.asp , 2010.

[3] AMD, "ATI Radeon HD 5870 Graphics," http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx , 2011.

[4] D. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," *Technical report*, National Science Foundation, 2006.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," *Proceedings of 2009 International Symposium on Performance Analysis of Systems and Software,* PP. 163-174, 2009.

[6] A. Balevic, L. Rockstroh, A. Tausendfreund, S. Patzelt, G. Goch, and S. Simon, "Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose gpus," *Proceedings of 11th International Conference Computational Science and Engineering,* pp. 327-334, 2008.

[7] L. Breiman, "Random forests," *Machine Learning, 45*, pages 5–32, 2001.

[8] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, "Classi_cation and Regression Trees," *Chapman and Hall/CRC, 1 edition,* 1984.

[9] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," *Proceedings of 27th International Symposium on Computer Architecture,* PP. 83-94, 2000.

[10] A. Frommer, and D.B. Szyld, "On Asynchronous Iterations," *Journal of Computational and Applied Mathematics*, PP 201-216, 2000.

[11] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and Its Applications*, PP 199–222, 1969.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *Proceedings of 2009 International Symposium on Worload Characterization,* PP. 44-54, 2009.

[13] J. Chen, B. Li, Y. Zhang, L. Peng and J.K. Peir, "Statistical GPU Power Analysis Using Tree-based Methods," *Proceedings of 2011 International Green Computing Conference and Workshops,* PP. 1-6, 2011.

[14] J. Chen, Z. Huang, F. Su, J.K. Peir, J. Ho, and L. Peng, "Weak execution ordering - exploiting iterative methods on many-core GPUs," *Proceedings of 2009*

*International Symposium on Performance Analysis of Systems and Software,* pp.154-163, 2010.

[15] J. Chen, B. Li, Y. Zhang, L. Peng, and J. Peir, “Tree Structured Analysis on GPU Power Study”, *Proceedings of 2011 International Conference Computer Design,* PP. 110-118, 2011

[16] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," *Proceedings of 18th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems,* PP. 351-360, 2010

[17] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," *Proceedings of 9th International Conference Computational Science,* PP. 914-923, 2009.

[18] T.F. Cox and M.A.A. Cox, "Multidimensional Scaling," *Chapman and Hall*. 2000

[19] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems," *Proceedings of 9th International Conference Parallel Architectures and Compilation Techniques,* PP. 353-364, 2010

[20] W.W.L. Fung, I. Sham, G. Yuan and T.M. Aamodt, “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” *Proceedings of 40th IEEE/ACM International Symposium on Microarchitecture,* PP. 407-418, 2007.

[21] M. Gebhart, D. R. J., D. Tarjan, S.W. Keckler, W.J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," *Proceedings of 38th International Symposium on Computer Architecture*, PP. 235-246, 2011.

[22] R. Gentleman and R. Ihaka, "The comprehensive r archive network," http://cran.r-project.org/, 2011.

[23] N.K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," *Proceedings of ACM/IEEE Conference Supercomputing*. Article 89, 2006.

[24] V.V.P. Harish and P.J. Narayanan, "Large graph algorithms for massively multithreaded architectures," *Technical report*, TR-74 Indian Institute of Information Technology, 2009.

[25] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *Proceedings of 36th International Symposium on Computer Architecture*, PP. 152-163, 2009.

[26] S. Hong and H. Kim, "An integrated gpu power and performance model," *Proceedings of 37th International Symposium on Computer Architecture*, PP. 280-289, 2010.

[27] B. Horn, "Robot vision," *Technical report*, The MIT Press, 1986.

[28] S. Huang, S. Xiao, and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," *Proceedings of 2009 International Symposium on Parallel & Distributed Processing,* PP. 1-8, 2009.

[29] Khronos, "OpenCL - The open standard for parallel programming of heterogeneous systems" http://www.khronos.org/opencl/

[30] H. Kim, "Performance Analysis Models and Tools," https://www.teragrid.org/c/document_library/get_file?uuid=17b3b22e-82d1-40d5-ae62-77939e7a0273&groupId=24026

[31] N. B. Lakshminarayana, J. Lee, and H. Kim, Age based scheduling for asymmetric multiprocessors. *Proceedings of 23rd International Conference Supercomputing, 2009.*

[32] B.C. Lee and D.M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," *Proceedings of 12th International Conference Architectural support for programming languages and operating systems,* PP. 185-194, 2006.

[33] J. Lee, N.B. Lakshminarayana, H. Kim and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," *Proceedings of 43th International Symposium on Microarchitecture* , pp.213-224, 2010.

[34] T. Li, D. Baumberger, and S. Hahn, Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *Proceedings of 14th ACM Symposium on Principles and Practice of Parallel Programming,* 2009.

[35] T. Li, V. Narayana, and T. El-Ghazawi, A Static Task Scheduling Framework for Independent Tasks Accelerated Using a Shared Graphics Processing Unit, In *Proceedings of 17th International Conference on Parallel and Distributed Systems,* 2011.

[36] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Microarchitecture*, PP. 39–55, 2008.

[37] W. Liu and M.W. Schmidt, ”Performance predictions for general-purpose computation on GPUs," *Proceedings of 2007 International Conference Parallel Processing*, pp.50-50, 2010.

[38] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for gpu-based computing," *Proceedings of Workshop Power-Aware Computing and Systems*, 2009.

[39] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," *Proceedings of 23rd International Conference Supercomputing*, PP. 256–265, 2009.

[40] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," *Proceedings of 37th International Symposium on Computer Architecture* PP. 235-246, 2010.

[41] P. Micikevicius, "3d finite difference computation on gpus using cuda," *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, PP. 79–84, 2009.

[42] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of gpu kernels using performance counters," *Proceedings of 37th International Conference Green Computing* PP. 115-122, 2010.

[43] Nvidia, "Cuda programming guide," http://developer.nvidia.com/cuda-downloads, 2011.

[44] Nvidia, "Nvidia cuda visual profiler," http://developer.nvidia.com/nvidia-visual-profiler, 2011.

[45] Nvidia, "Nvidia cuda zone," http://developer.nvidia.com/category/zone/cuda-zone 2011.

[46] Nvidia, "Nvidia geforce series gtx580, gtx280, 8800gtx, 8800gt," http://www.nvidia.com/object/geforce_family.html, 2011.

[47] Nvidia, "Whitepaper: Nvidia's next generation cudatm compute architecture: Fermi," http://www.nvidia.com/object/fermi_architecture.html, 2010.

[48] Nvidia, "Tesla m2050 and tesla c2070 computing processor board," http://www.nvidia.com/object/tesla_computing_solutions.html, 2011.

[49] Nvidia, "Cuda SDK," http://developer.nvidia.com/gpu-computing-sdk, 2011.

[50] Nvidia, "Cuda: What is cuda?" http://developer.nvidia.com/what-cuda, 2011.

[51] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," *Proceedings of the IEEE*, PP. 879–899, 2008.

[52] D.Patterson, "The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges," *http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf*. 2010.

[53] P. P´erez, M. Gangnet, and A. Blake, "Poisson image editing," *ACM Transaction on Graph*, PP. 313–318, 2003.

[54] K. Ramani, A. Ibrahim and D. Shimizu, "Powerred: A flexible power modeling framework for power efficiency exploration in gpus," *Proceedings of Workshop General Purpose Processing on Graphics Processing Units*, 2007.

[55] D.Q. Ren and R. Suda, "Investigation on the power efficiency of multi-core and gpu processing element in large scale simd computation with cuda," *Proceedings of 2nd International Conference Green Computing*, PP. 309-316, 2010.

[56] I. Research group, "Parboil benchmark suite," http://impact.crhc.illinois.edu/parboil.php, 2011.

[57] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, "Energy-aware high performance computing with graphic processing units," *Proceedings of Conference on Power aware computing and systems,* 2008.

[58] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," *Proceedings of 13th ACM Symposium on Principles and practice of parallel programming*, PP. 73-82, 2008.

[59] S. Ryoo, C.I. Rodrigues, S.S. Stone, S.S. Baghsorkhi, S.Z. Ueng, J.A. Stratton, and W.W. Hwu, "Program optimization space pruning for a multithreaded gpu," *Proceedings of 6th International Symposium on Code generation and optimization*, PP. 195–204, 2008.

[60] J. Sheaffer, K. Skadron, and D. Luebke, "Studying thermal management for graphics-processor architectures," *Proceedings of 2005 International Symposium on Performance Analysis of Systems and Software,* PP. 54-65, 2005.

[61] D. Tarjan and K. Skadron, "The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches," *Proceedings of 2010 International Conference High Performance Computing, Networking, Storage and Analysis,* PP. 1-10, 2010.

[62] Top500, "Top 500 supercomputer list," http://www.top500.org/list/2011/06/100, 2011.

[63] S. Venkatasubramanian and R.W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," *Proceedings of 23rd International Conference Supercomputing*, PP. 244–255, 2009.

[64] V. Volkov and J.W. Demmel, "Benchmarking gpus to tune dense linear algebra," *Proceedings of 22nd International Conference on Supercomputing*, pages 1–11. 2008.

[65] H. Wong, M.M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," *Proceedings of 2005 International Symposium on Performance Analysis of Systems and Software,* pp.235-246, 2010.

[66] W. Ye, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," *Proceedings of 37th Design Automation Conference,* pp.340-345, 2000.

[67] YOKOGAVA, "WT210," http://tmi.yokogawa.com/us/

[68] Y. Zhuo, X.-L. Wu, J. Haldar, W. mei Hwu, Z. pei Liang, and B. Sutton, "Accelerating iterative field-compensated mr image reconstruction on gpus," *Proceedings of 2010 International Symposium on Biomedical Imaging: From Nano to Macro,* PP. 820-823, 2010.

# BIOGRAPHICAL SKETCH

Jianmin Chen received his bachelor's degree in Computer Science from the School of Computer Science and Technology in Beijing University of Posts and Telecommunications, Beijing, China. He received his PhD in the Department of Computer Information Science Engineering in University of Florida in the spring of 2012.