6-18-2014

Performance and Power Optimization of GPU Architectures for General-purpose Computing

Yue Wang
University of South Florida, 305248163@qq.com

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Engineering Commons

Scholar Commons Citation

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Performance and Power Optimization of GPU Architectures for General-purpose Computing

by

Yue Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Nagarajan Ranganathan, Ph.D.
Srinivas Katkoori, Ph.D.
Adriana Iamnitchi, Ph.D.
Bo Zeng, Ph.D.
Soumyaroop Roy, Ph.D.

Date of Approval:
June 18, 2014

Keywords: Power Gating, Cache, Workload Balancing, Dynamic Frequency Scaling, PID Controller, Linear Programming

Copyright © 2014, Yue Wang
DEDICATION

I dedicate this dissertation work to my loving parents for providing me their unconditional love and support in my journey of study. I also dedicate this dissertation to my dearest friend Yu Meng, who has helped me very much throughout the process.
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my doctoral advisors Dr. Nagarajan Ranganathan for his valuable guidance and support that is given to me and my work. His knowledge, work ethics, and the professional standards he adheres to inspire me to put my heart and mind into solving the problems that are presented in this dissertation. I would also like to thank Dr. Srinivas Katkoori, Dr. Adriana Iamnitch, Dr. Bo Zeng, and Dr. Soumyaroop Roy for taking their time to participate in my doctoral committee and providing valuable suggestions to improve this manuscript. To my loving parents Tianjun Wang and Yan Liu. I am forever grateful to you for raising me to be the man that I am today. To Dr. Labrador and Dr. Perez, thank you for introducing me to USF and all their help. To my research colleagues and friends Himanshu, Ransford, Michael, Ziad, Matt, Saurabh, Santosh, Anthony, Ashish, Satrajit, Ingo, Diego Weijian, Haiqiong, Jing, Yingying, Yun, Xiang, Caitrin, Shilpa, Ravi, Bingxiong, and others. Finally, I would like to extend my gratitude to the CSE department for the support throughout all the years.
TABLE OF CONTENTS

LIST OF TABLES ....................................................................................................................... iv

LIST OF FIGURES ..................................................................................................................... v

ABSTRACT .............................................................................................................................. vii

CHAPTER 1: INTRODUCTION ................................................................................................. 1
  1.1 Power and Energy Concern in GPU Computing Systems ............................................. 2
  1.2 Performance Concern in GPU Computing Systems ...................................................... 2
  1.3 Contributions of This Dissertation .............................................................................. 3
  1.4 Outline of Dissertation .................................................................................................. 5

CHAPTER 2: BACKGROUND AND RELATED WORK .............................................................. 7
  2.1 GPU Architecture and Terminology ............................................................................... 7
      2.1.1 Single Instruction Multiple Thread (SIMT) Architecture ....................................... 8
      2.1.2 Warp Scheduling .................................................................................................... 9
      2.1.3 Memory Hierarchy ................................................................................................ 10
  2.2 Power Consumption in CMOS Circuit ........................................................................... 11
      2.2.1 Power Gating Technique ....................................................................................... 12
      2.2.2 Dynamic Frequency Scaling ............................................................................... 13
  2.3 Proportional-Integral-Derivative Controlling Algorithm ................................................ 14
  2.4 Linear Programming and Simplex Method ..................................................................... 16
  2.5 Related Work .................................................................................................................. 20
      2.5.1 Dynamic Workload Management on CPU/GPU ...................................................... 20
      2.5.2 Power Optimization on CPU/GPU ......................................................................... 21
          2.5.2.1 To Reduce Static Leakage Power .................................................................... 21
          2.5.2.2 To Reduce Dynamic Power ............................................................................ 22

CHAPTER 3: RUNTIME POWER GATING IN CACHES OF GPUS FOR LEAKAGE ENERGY SAVINGS ................................................................................................................... 24
  3.1 Motivation ....................................................................................................................... 24
  3.2 Proposed Microarchitectural Technique for Leakage Reduction on Caches
      Using Power Gating ......................................................................................................... 26
3.2.1 Power Gating L1 Caches of GPU ................................................................. 27
3.2.2 Power Gating L2 Caches of GPU ................................................................. 28
3.2.3 Analysis of Latency Hiding .............................................................................. 30
   3.2.3.1 Hiding Transition Latency Between off and active in L1 ....................... 30
   3.2.3.2 Hiding Transition Latency Between sleep and active in L1 .................... 31
   3.2.3.3 Hiding Transition Latency Between sleep and active in L2 ................. 32
3.3 Experimental Setup and Results ....................................................................... 32
   3.3.1 Modeling of Power Gated SRAM Array ...................................................... 32
   3.3.2 Benchmarks and Simulator Setup ................................................................. 33
   3.3.3 Evaluation and Analysis .............................................................................. 37
3.4 Conclusion .......................................................................................................... 39

CHAPTER 4: LOAD BALANCING ON THE RUNTIME FOR GPUS .............................. 41
4.1 Motivation ............................................................................................................ 41
4.2 Methodology and Implementation ...................................................................... 43
   4.2.1 Overview of Dynamic Workload Balancing .............................................. 44
   4.2.2 Microarchitecture Design .......................................................................... 45
   4.2.3 Dynamic Workload Balancing Algorithm ................................................. 48
      4.2.3.1 Memory Request Handling ................................................................... 49
      4.2.3.2 Synchronization and Shared Memory Access ....................................... 50
4.3 Experimental Results and Analysis .................................................................... 52
   4.3.1 Performance Improvement Result ............................................................... 53
   4.3.2 Choosing Optimal Values for threshold and fraction ................................. 55
   4.3.3 Result Analysis ............................................................................................ 56
      4.3.3.1 Warp-level Parallelism ......................................................................... 57
      4.3.3.2 Average Memory Access Time .............................................................. 58
   4.3.4 Overhead Analysis ....................................................................................... 59
4.4 Conclusion ............................................................................................................ 61

CHAPTER 5: DYNAMIC FREQUENCY SCALING ON GPUS ..................................... 63
5.1 PIDDFS - Dynamic Frequency Scaling Using PID Controlling Theory .......... 64
   5.1.1 Motivation .................................................................................................... 64
   5.1.2 PIDDFS Methodology ............................................................................... 66
      5.1.2.1 SMX Frequency Controller ................................................................. 67
      5.1.2.2 Memory Channel Frequency Controller .............................................. 68
   5.1.3 Implementation of PIDDFS ......................................................................... 68
      5.1.3.1 Prototype of Code Implementation ...................................................... 68
      5.1.3.2 Fixing the Activation Period ............................................................... 69
      5.1.3.3 Tuning the PID Parameters ................................................................. 71
5.2 LPDFS - Dynamic Frequency Scaling Using Linear Programming ................. 71
   5.2.1 Motivation .................................................................................................... 72
5.2.2 Methodology of LPDFS ................................................................. 73
5.2.3 Implementation of LPDFS ......................................................... 76
5.3 Simulation and Evaluation .............................................................. 78
  5.3.1 Evaluation of Memory Stall Reduction via PIDDFS ..................... 79
    5.3.1.1 Result ........................................................................ 79
    5.3.1.2 Analysis .................................................................... 80
  5.3.2 Evaluation of Performance Improvement ...................................... 82
    5.3.2.1 Result ........................................................................ 83
    5.3.2.2 Analysis .................................................................... 83
  5.3.3 Evaluation of Power Saving ....................................................... 84
    5.3.3.1 Result ........................................................................ 84
    5.3.3.2 Analysis .................................................................... 85
5.4 Conclusion .................................................................................... 86

CHAPTER 6: CONCLUSIONS AND FUTURE DIRECTIONS ............................ 88

REFERENCES ..................................................................................... 90

ABOUT THE AUTHOR ........................................................................ END PAGE


**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table 3.1</th>
<th>Three Leakage Modes of Cache</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.2</td>
<td>Analysis of Transition Latency Hiding</td>
<td>30</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Leakage Power in Different Cache Modes</td>
<td>34</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Benchmark Description</td>
<td>35</td>
</tr>
<tr>
<td>Table 3.5</td>
<td>Architecture Comparison</td>
<td>36</td>
</tr>
<tr>
<td>Table 3.6</td>
<td>Leakage Saving Result</td>
<td>37</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Performance Improvement Comparison</td>
<td>53</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Percentage of Extra Misses and AMAT</td>
<td>61</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Parameters of LPDFS and Acquisition Means</td>
<td>77</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Performance Improvement of PIDDFS and LPDFS</td>
<td>82</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Power Saving of PIDDFS and LPDFS</td>
<td>85</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 2.1  Sketch of GPU architecture........................................................................................................ 7
Figure 2.2  Modules within a streaming multiprocessor.................................................................................. 9
Figure 2.3  Power gating a circuit block with footer sleep transistor ............................................................. 13
Figure 2.4  Closed-loop control system with PID controller ........................................................................ 14
Figure 2.5  Graphical interpretation of the example of LP problem ............................................................... 17
Figure 3.1  Percentage of idle cycles in caches .............................................................................................. 25
Figure 3.2  Circuit implementation of power gated cache array .................................................................. 26
Figure 3.3  Sketch of power gate controlling mechanism ............................................................................. 27
Figure 3.4  Power gating strategy ................................................................................................................ 27
Figure 3.5  Power gated SRAM array that we model in HSPICE ................................................................. 33
Figure 4.1  Example of finishing time deviation .......................................................................................... 42
Figure 4.2  Coefficient of variation $c_v$ of SMXs original finishing times .................................................. 43
Figure 4.3  Sketch of hardware implementation of DWB ............................................................................ 46
Figure 4.4  Workload balancing between SMXs ......................................................................................... 48
Figure 4.5  Redirecting memory request to avoid deadlock ....................................................................... 49
Figure 4.6  Comparison of $c_v$ with and without DWB .............................................................................. 54
Figure 4.7  Average total execution times under different combinations of $T_{complete}$ and $\alpha$ ............. 55
Figure 4.8  Number of unfinished warps comparison for BFS .......................................................... 56

Figure 4.9  Number of unfinished warps comparison for HIS .......................................................... 56

Figure 4.10  Performance improvement tends to be large when $c_v$ is large, and when AMAT is small ........................................................................................................................................ 58

Figure 5.1  Ratio of stalled cycles over total execution cycles in MSHRs, ICNT-Mem and Mem-ICNT queues ........................................................................................................................................ 65

Figure 5.2  Sketch of the mechanism of PIDDFS .................................................................................. 66

Figure 5.3  Flowcharts of the frequency controlling .............................................................................. 67

Figure 5.4  Block diagram of PIDDFS code implementation ................................................................... 69

Figure 5.5  Average congested cycles in MSHRs and memory channels while using different PIDDFS activation frequencies .................................................................................................................................. 70

Figure 5.6  Average total execution cycles when using different LPDFS activation frequencies ....................................................................................................................................................... 77

Figure 5.7  The ratio of memory time over total execution time, L1 miss rate and L2 miss rate of benchmarks ........................................................................................................................................ 79

Figure 5.8  Ratio of reduced stalled cycles in MSHRs, ICNT-Mem queues and Mem-ICNT queues (with dotted lines showing average values) ..................................................................................... 80

Figure 5.9  Correlation between the percentage of reduced stalled cycles and the ratio of memory time over total execution time ........................................................................................................... 81

Figure 5.10  Correlation between the performance improvement (using y-axis on the left) and the ratio of memory time over total execution time (using y-axis on the right) ................... 83

Figure 5.11  Correlation between the power saving (using y-axis on the left) and the ratio of memory time over total execution time (using y-axis on the right) ........................................... 86
ABSTRACT

Power-performance efficiency has become a central focus that is challenging in heterogeneous processing platforms as the power constraints have to be established without hindering the high performance. In this dissertation, a framework for optimizing the power and performance of GPUs in the context of general-purpose computing in GPUs (GPGPU) is proposed. To optimize the leakage power of caches in GPUs, we dynamically switch the L1 and L2 caches into low power modes during periods of inactivity to reduce leakage power. The L1 cache can be put into a low-leakage (sleep) state when a processing unit is stalled due to no ready threads to be scheduled and the L2 can be put into sleep state during its idle period when there is no memory request. The sleep mode is state-retentive, which obviates the necessity to flush the caches after they are woken up, thereby, avoiding any performance degradation. Experimental results indicate that this technique can reduce the leakage power by 52% on average. Further, to improve performance, we redistribute the GPGPU workload across the computing units of the GPU during application execution. The fundamental idea is to monitor the workload on each multi-processing unit and redistribute it by having a portion of its unfinished threads executed in a neighboring multi-processing unit. Experimental results show this technique improves the performance of the GPGPU workload by 15.7%. Finally, to improve both performance and dynamic power of GPUs, we propose two dynamic frequency scaling (DFS) techniques implemented on CPU host threads, one of which is motivated by the significance of the pipeline stalls during GPGPU execution. It applies a
feedback controlling algorithm, Proportional-Integral-Derivative (PID), to regulate the frequency of parallel processors and memory channels based on the occupancy of the memory buffering queues. The other technique targets on maximizing the average throughput of all parallel processors under the dynamic power constraints. We formalize this target as a linear programming problem and solve it on the runtime. According to the simulation results, the first technique achieves more than 22% power savings with a 4% improvement in performance and the second technique saves 11% power consumption with 9% performance improvement. The contributions of this dissertation represent a significant advancement in the quest for improving performance and reducing energy consumption of GPGPU.
CHAPTER 1:
INTRODUCTION

Graphics Processing Units (GPUs) are widespread in traditional computers and portable devices nowadays. With hundreds of pipelines integrated on a single chip, GPUs have demonstrated their powerful parallel computing capability in high quality graphic processing. Moreover, as the interest in general-purpose parallel computing grows, the newly-released generation of GPUs are designed with full programmability, hence more flexibility for parallel applications, regardless of graphic or non-graphic. Especially in recent years, GPUs have emerged as an effective platform for running non-graphic applications that have traditionally been handled by central processing units (CPUs). This computing paradigm, also known as general-purpose computing in GPUs (GPGPU), is being used to harness the massive computation power of GPUs for domains such as molecular science, weather prediction and cryptography [Stone et al. 2009][Govett et al. 2010][Tzeng and Wei 2008]. While Nvidia’s CUDA [NVIDIA Corporation 2007] provides a GPGPU framework for their GPUs, OpenCL [Tsuchiyama et al. 2010] provides a similar framework across a more heterogeneous set of platforms, including GPUs, CPUs, DSPs (digital signal processors), and other processors. To address the memory wall [NVIDIA 2010], GPUs are equipped with on-chip caches and high performance off-chip or main memories such as GDDR5 [NVIDIA 2010].
1.1 Power and Energy Concern in GPU Computing Systems

As the computing capability of GPUs is growing tremendously, and mobile platforms that adopt GPUs are popularized, the demand on GPU energy and heat dissipation begins to catch more attention on GPU design. Specifically, for the GPGPU, power management is of great necessity and importance due to the memory access characteristics of general-purpose applications. The profiles of memory usage between graphic applications and general-purpose applications are quite different [Pharr and Fernando 2005]. Graphic applications have much higher sequential memory access rate. Read operations are dominant among these sequential accesses. A special kind of read-only cache, texture cache, is designed for locality in two dimensions in order to support such sequential access pattern. The texture cache has a high hit rate, hence the heavy reuse rate between neighboring pixels. However, general-purpose applications tend to have high random memory access rate. The data involved in calculations are mostly not texture-structured, but general data. So the texture cache is seldom used in general-purpose applications. Moreover, the general-purpose applications permit programmers to write algorithms that have more interactions with data values. Write accesses are more significant than graphic applications. Therefore, the cache missing rate is much higher and the idle time of cache arrays is also much longer. Reducing the leakage power of caches and the dynamic power of processing units during pipeline stalls for cache misses resolving become important for power management of GPUs for GPGPU workload.

1.2 Performance Concern in GPU Computing Systems

GPUs achieve high throughput for graphics workload by performing fine-grained multithreading on a large number of threads. As the graphics programming is becoming increasingly versatile,
developers constantly seek to incorporate more sophisticated algorithms and leverage more configurable graphics pipelines. A major challenge in the evolution of GPU programming involves preserving GPU performance levels while increasing the generality and expressiveness of application interfaces. The frameworks of GPGPU interfaces, such as NVIDIA’s CUDA [NVIDIA Corporation 2007] and AMD’s OpenCL [Tsuchiyama et al. 2010], abstract computation as large batch operations that involve many invocations of a kernel function operating in parallel. The computations execute on GPUs efficiently only under the condition of massive data parallelism. Programs that attempt to implement non-data-parallel algorithms perform poorly.

However, the scale of parallelism that fits in GPUs is not easy to achieve in general-purpose application. GPGPU programming models only permit well-written programs to make good use of both GPU programmable cores and other resources. The current GPGPU programming frameworks are suitable for general-purpose computations that have balanced workload distribution among the processing units so that the computing power of the processing cores can be efficiently utilized.

1.3 Contributions of This Dissertation

The main contribution of this dissertation is a set of methodologies proposed at the architecture level to effectively reduce leakage power in the cache arrays, balance the workload among processing units and improve dynamic power utilization and overall throughput by frequency regulation.

A brief description of the contributions is:

- **Runtime power gating in caches of GPUs for leakage energy savings** - In this work, a novel run-time microarchitectural technique is proposed to achieve savings in leakage energy in
the caches of GPUs when they are idle during workload execution. The sleep mode is state-retentive, which obviates the necessity to flush the caches after they are woken up, thereby, avoiding any performance degradation.

- **Load balancing on the runtime for GPUs** - In this work, a runtime microarchitectural technique is proposed to improve the GPGPU performance by balancing the computing tasks dynamically among cores of the GPU. This technique does not necessitate any code recompilation. Thus, legacy application binaries are able to readily benefit from this technique. Two control parameters can be properly adjusted for different GPU platforms. A microarchitecture redesign based on shared structures is proposed, storing the scheduling queues and warp context of two multiprocessing units grouped together, so that the workload balancing and task scheduling can happen simultaneously.

- **Dynamic frequency scaling on GPUs** - In this work, we propose two DFS techniques for GPGPU in order to optimize the power and performance. Both techniques are implemented in the CPU host thread, giving control to frequency regulation modules of the GPU when they are activated periodically. So the overhead introduced to GPU execution is minimized. PIDDFS technique regulates the frequencies for each processor and memory channel based on the utilization of the memory access buffering queues using a feedback controlling model, the leading phase of which helps to achieve in-advance control. LPDFS technique is based on linear programming theory, maximizing the average throughput of all multiprocessing units in the GPU under a dynamic power budget by regulating the frequency for each processing unit. Overall dynamic power budget is given as an input,
which let the user have direct and flexible control over the total power consumption of the GPU, especially in battery-powered systems.

1.4 Outline of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides the background and discusses the research work that is related to the topic of thesis dissertation. Specifically, the background of GPU architecture, including scheduling mechanism and memory hierarchy, is presented, followed by the introduction of the power gating technique, dynamic frequency scaling, and Proportional-Integral-Derivative (PID) controller. In Chapter 3, we propose a novel microarchitectural technique for run-time power gating caches of GPUs to save leakage energy. The low power mode of caches is state-retentive, which precludes the necessity to flush the caches after they are woken up. The methodology of the technique and the discussion of latency hiding is presented. In Chapter 4, a microarchitectural technique for workload balancing is investigated in the context of general-purpose computing in GPUs (GPGPU). It monitors the workload on each multi-processing unit and redistributes it by having a portion of its unfinished threads executed in a neighboring multi-processing unit in order to improve performance. A microarchitecture design of this technique based on shared structures is proposed. In Chapter 5, we propose two dynamic frequency scaling (DFS) techniques for GPGPU. PIDDFS technique applies Proportional-Integral-Derivative (PID) feedback controlling theory and regulates the frequency of parallel processors and memory channels based on the occupancy of the memory buffering queues. LPDFS technique formalizes the problem of maximizing the average throughput of all parallel processors under the dynamic power constraints as a linear programming
problem and solves it on the runtime, then output the adjusted frequency for each processor. Finally, some concluding remarks and the challenges going forward are discussed in Chapter 6.
CHAPTER 2:
BACKGROUND AND RELATED WORK

In this chapter, some fundamental concepts that related to this research work are presented. Specifically, we respectively discuss the GPU terminology and architecture details, basics of power consumption in digital CMOS circuit, power gating technique for reducing subthreshold leakage in CMOS circuits, basics of dynamic frequency scaling and the Proportional-Integral-Derivative (PID) algorithm. A discussion of the various related works for power and performance optimization of processors on the architecture and microarchitecture level is also presented in this chapter.

2.1 GPU Architecture and Terminology

Figure 2.1. Sketch of GPU architecture [NVIDIA Corporation 2007].
Figure 2.1 depicts a high-level overview of GPU architecture. As shown in the figure, the GPU is a single instruction, multiple threads (SIMT) engine that comprises of *streaming multiprocessors* (SMXs) in the figure and *memory channels* that communicate through an interconnection network. The current generation of GPUs can have multiple clock domains. Generally, there are three clock domains in the GPU: (1) the SMX clock domain, which applies to all the SMXs (2) the interconnection network clock domain (3) the memory channel clock domain, which applies to L2 arrays and DRAMs in all the memory channels. The memory request buffering queues, such as MSHRs, ICNT-Mem queues and Mem-ICNT queues, also have clock resynchronization circuits and therefore, serve the purpose of synchronizers between different clock domains. Meanwhile, clock frequencies can have any arbitrary value, instead of multiples of each other. According to [NVIDIA 2013], the DVFS software in NVIDIA Tegra GPU can increase or decrease frequency in steps as small as 1 MHz.

2.1.1 Single Instruction Multiple Thread (SIMT) Architecture

SMXs are clusters of pipelined, in-order processors, as shown in Figure 2.2. The CUDA framework [NVIDIA Corporation 2007] provides extensions to the C programming language to support a coprocessor model (with the CPU as the host and the GPU as a device). In this framework, users can write C functions known as *kernels*. Each kernel, when called, gets executed \( n \) times by \( n \) CUDA threads. In order to capture the variation in dimensionality of data structures, the threads can be grouped to form linear, 2D, or 3D *thread blocks*. For instance, a kernel that adds two vectors can be represented by linear thread blocks, while one that adds two matrices can be represented by 2D thread blocks. Each thread block is assigned to one SMX, which is a light-weight, pipelined, in-order processor, as shown in Figure
2.2. When an SMX receives one or more thread blocks to execute, it partitions them into *warps*, which are groups of fixed number of threads within a thread block that are treated as a basic units that are managed, scheduled, and executed concurrently on an SMX. Each thread block gets executed in its residing SMX independently and has no correlation with other thread blocks.

![Diagram of Warp Scheduler and Execution Pipeline]

**Figure 2.2.** Modules within a streaming multiprocessor [NVIDIA Corporation 2007].

2.1.2 **Warp Scheduling**

A thread block will be issued to an SMX from the kernel if there are enough resources free on that SMX. As introduced earlier, a *warp* is a group of fixed number of threads within a thread block that
are treated as a basic units that are managed, scheduled, and executed concurrently on an SMX. The warp scheduler in an SMX selects and issues the ready warps into the pipelines in a round-robin fashion. Non-ready warps, such as those waiting on memory accesses, will be skipped by the scheduler. At every instruction issue stage, the warp scheduler selects a warp that is ready to execute from the I-cache and issues the next instruction from the active threads of the warp. A warp executes one common instruction at a time. If threads of a warp diverge via a conditional branch, the warp will serially execute each branch path taken, disabling threads that are not on that path. The threads converge back to the same path when all branches complete. Branch divergence only occurs among threads within a warp. Different warps execute independently and do not have branch divergence coordination regardless of whether they are executing common or disjointed code paths. If any thread inside a warp incurs a long latency operation, all threads in the same warp are switched out along with it until the long latency operation is over. Meanwhile, other ready warps will be scheduled for execution.

### 2.1.3 Memory Hierarchy

Each SMX has a private L1 cache, which is partitioned into *shared memory, data cache, constant cache* and *texture cache*. Any content in the shared memory can be accessed by all threads within a block, while the contents in the rest of the partitions are exclusively owned by the thread requesting for it. Data cache stores the data loaded from L2 cache. Constant cache is read-only and used to store data that does not change over the course of a kernel execution. Texture cache was originally designed for special purposes, such as OpenGL and DirectX rendering. But it also can be used for general-purpose caching. If some of the threads are having L1 misses, the whole warp will be switched
out and memory requests will be generated. Those memory requests are temporarily stored in a module
called miss status holding register (MSHR) and merged if some of them are requesting for the content in
the same cache block. Each SMX has its own MSHR, interacting with the interconnection network so
that the memory requests can be transmitted to lower-level storage units, such as L2 cache and DRAM,
in the memory channels. All the partitions of L1 cache can be accessed within the SMX, which it resides
in, while the unified L2 cache in each memory channel is shared among all the SMXs.

The interconnection network connects the SMXs to the memory channels, passing memory
requests that are generated by SMXs to memory channels and sending fetched data back from memory
channels to the requesting SMXs. The memory channel has three major components – a memory
controller, a unified L2 cache, and an off-chip DRAM (GDDR3/GDDR5). Within a memory channel,
buffering queues (ICNT-Mem queues and Mem-ICNT queues) for both memory requests and fetched
data connect the memory channel and interconnection network through a memory controller. When there
are too many memory requests flowing in a memory channel, or the interconnection network is not fast
enough to pull out the returning data from a memory channel, the buffering queues will be filled up and
congestions will happen. Then the memory controller in the congested memory channels will stall the
channel until the congested buffering queue has an available entry.

2.2 Power Consumption in CMOS Circuit

There are two main components that constitute the power used by a CMOS integrated circuit:

static power and dynamic power. Static power consists of the power used when the transistor is not in
the process of switching. Due to the fact that transistors are not perfect insulators, the current is
conducted through from the power source to the ground. Static power is determined by the formula:

\[ P_{\text{static}} = I_{\text{static}} \cdot V_{dd} \]

where \( V_{dd} \) is the supply voltage and \( I_{\text{static}} \) is the total current flowing through the CMOS circuit. Although
the static power of CMOS circuit is normally very low (because \( I_{\text{static}} \) is low), the leakage current
becomes larger as the size of CMOS device grows and the gate oxide thickness decreases.

Dynamic power is the sum of transient power consumption \( P_{\text{transient}} \) and capacitive load power
consumption \( P_{\text{cap}} \). \( P_{\text{transient}} \) is the amount of power consumed when the device changes logic states from "0" to "1" or vice versa. Capacitive load power consumption \( P_{\text{cap}} \) represents the power used to charge and discharge the load capacitance. Together the dynamic power can be expressed as:

\[ P = a \cdot C \cdot V^2 \cdot f \]

where \( a \) is the activity factor (the fraction of the circuit that is switching), \( C \) is the switched capacitance,
\( V \) is the supply voltage, and \( f \) is the clock frequency.

### 2.2.1 Power Gating Technique

Power gating is an effective technique to reduce leakage power remarkably when the circuit is idle. It is used as a routine power management technique in the commercial products to eliminate subthreshold leakage of a CMOS circuit. As shown in Figure 2.3, \textit{sleep transistor}, which is a high threshold voltage transistor, is inserted between the actual ground and the circuit ground, like in [Agarwal et al. 2003][Johnson et al. 1999]. When the circuit block is idle, the sleep transistor can be shut off. The leakage path between \( V_{dd} \) and ground is cut off consequently. Then the circuit block is put to the
A tradeoff for staying in sleep mode is that some of the circuit functions might be restrained. Another tradeoff would be performance degradation if the mode transition latency is significant. The mode transition latency is the time period for switching the circuit block between active and sleep mode. Given that the mode switching costs extra energy, a circuit block should stay in the sleep mode for a minimum period in order to let leakage savings break even with the dynamic energy overhead that it incurs. The minimum time period is called breakeven period [Kim et al. 2003]. The effectiveness of power gating is influenced by the length of mode transition latency and the ratio of the power gated period over the breakeven period.

![Diagram](image_url)

**Figure 2.3.** Power gating a circuit block with footer sleep transistor [Agarwal et al. 2003][Johnson et al. 1999].

### 2.2.2 Dynamic Frequency Scaling

Dynamic Frequency Scaling (DVS) is one of the most commonly used power-aware techniques in modern processors. DFS regulates the frequency in each domain of the system at runtime, either to decrease power consumption while preserving high performance or to increase performance of the processor under a power constraint. The dynamic power of a CMOS circuit is:

\[ P = a \cdot C \cdot V^2 \cdot f \]
where $P$ is the dynamic power consumed, $a$ is the activity factor (the fraction of the circuit that is switching), $C$ is the switched capacitance, $V$ is the supply voltage, and $f$ is the clock frequency. Therefore, when the supply voltage is held constant and the frequency is scaled down, the dynamic power is also scaled down. In reality, some specific consideration has to be made in order to implement DFS on a circuit, such as division of the chip into multiple clock domains and the synchronization among the different domains.

2.3 Proportional-Integral-Derivative Controlling Algorithm

The discrete Proportional-Integral-Derivative (PID) controller gives control to any measurable variable that can be affected by manipulating some other variables in a process. The PID controller is widespread in the industry applications, such as motor control, control of speed, force, temperature, or pressure. It has become a standard due to its good performance and simplicity. Figure 2.4 presents a schematic of closed loop PID controlled system. A reference setpoint value $y_0$ is given at the beginning. Then, the PID controller measures process value $y$ and compares it with $y_0$. The controller calculates a new process input $u$ based on the error $e$, the difference between $y$ and $y_0$. The input $u$ is supplied to the system in order to adjust the measured process value $y$ back to the reference setpoint $y_0$.

![Figure 2.4. Closed-loop control system with PID controller.](image)
The mathematical expression of a PID controller has three terms: the proportional term, integral term and derivative term respectively. The transfer function of a parallel-formed PID controller can be written as:

\[
\frac{u}{e}(s) = K_p + \frac{K_i}{s} + K_d s
\]

where \(K_p\) is the proportional gain, \(K_i\) the integral gain, \(K_d\) the derivative gain, or in the form of:

\[
\frac{u}{e}(s) = K_p \left(1 + \frac{1}{T_s s} + T_d s\right)
\]

where \(T_i\) is the integral time constant and \(T_d\) the derivative time constant. In the time domain, the output of the PID controller can be expressed as:

\[
u(t) = K_p \left[e(t) + \frac{1}{T_i} \int_0^t e(\sigma)d\sigma + T_d \frac{de(t)}{dt}\right]
\]

In order to get the discrete form of the transfer function, the integral and derivative terms need to be approximated as below:

\[
\int_0^t e(\sigma)d\sigma \approx T \sum_{k=0}^n e(k)
\]

\[
\frac{de(t)}{dt} \approx \frac{e(n) - e(n-1)}{T}
\]

where \(T\) is the period of sampling and \(n\) is the number of sampling operations during time interval \(t\). In this way, the discrete PID controller can be described as:

\[
u(t) = K_p e(t) + K_i \sum_{k=0}^n e(k) + K_d \left[e(n) - e(n-1)\right]
\]

where
With these three terms, the PID controller is capable of manipulating the process inputs based on the history and the rate of change of the signal. The functionalities of the three terms are as follows:

- The proportional term ($P$) - this term is proportional magnification or minification over error $e$, which has no phase-leading or phase-lagging characteristic towards the system. A too large $P$ term may result in instability of the system.

- The integral term ($I$) - this term sums up the previous errors. With integral term, the system should have no stationary error when the system out is stable. However, an $I$ term that weighs too much will render slow response to the signal changes.

- The derivative term ($D$) - this term represents the changing rate of the error $e$. A rapid change in the error will lead to a large derivative term, which will lead a significant $D$ term. Essentially, the $D$ term improves the response to sudden changes in the system output. It can be considered as a high-pass filter, which makes the system more sensitive to noise.

### 2.4 Linear Programming and Simplex Method

A linear programming problem is defined as the problem of maximizing or minimizing a linear function subject to linear constraints. The constraints may be equalities or inequalities. For example: find numbers $x_1$ and $x_2$ that maximize the sum $x_1 + x_2$ subject to the constraints, $x_2 \geq 0$ and

\[
\begin{align*}
2x_1 + x_2 &\leq 4 \\
x_1 + 2x_2 &\leq 3
\end{align*}
\]
In this problem there are two variables, and five constraints. All the constraints are inequalities and they are all linear in the sense that each involves an inequality in some linear function of the variables. The function to be maximized (or minimized) is called the objective function. We can solve this problem by graphing the set of points in the plane that satisfies all the constraints and then finding which point maximizes the value of the objective function. In the present example, the constraint set is the plotted in Figure 2.5. We can easily see that \((5/3,2/3)\) is the solution for the problem, and the value of the objective function is \(7/3\).

![Figure 2.5. Graphical interpretation of the example of LP problem.](image)

In general, the objective function always takes on its maximum (or minimum) value at a corner point of the constraint set, if the constraint set is bounded. Occasionally, the maximum occurs along an entire edge or face of the constraint set.

Now, to solve this problem via simplex method, we need to convert the problem into standard form by adding slack variables \(x_3 \geq 0\) and \(x_4 \geq 0\). The converted problem is: maximize \(x_1+x_2\) under the constraints:
Let $z = x_1 + x_2$. We are now dealing the following system of linear equations:

\[
2x_1 + x_2 + x_3 = 4 \\
x_1 + 2x_2 + x_4 = 3
\]

The goal is to maximize $z$. Here, $x_1$ and $x_2$ are called *nonbasic variables*, and $x_3$ and $x_4$ are called *basic variables*. A *basic solution* is obtained from the system of equations by setting the nonbasic variables to zero, which yields: $x_1 = x_2 = 0$, $x_3 = 4$, $x_4 = 3$. Now we need to see if this is an optimal solution or we can still increase $z$. By looking at Row 0, we see that we can increase $z$ by increasing $x_1$ or $x_2$. This is because these variables have a negative coefficient in Row 0. This rule can be formalized as:

**Rule 1**  *If all variables have a nonnegative coefficient in Row 0, the current basic solution is optimal. Otherwise, pick a variable with a negative coefficient in Row 0.*

The variable chosen by Rule 1 is called the *entering variable*. Here we choose $x_1$ as the entering variable. The idea is to *pivot* in order to make the nonbasic variables become basic variables. The change of basis is done by *Gauss-Jordan procedure*. Next, we need to choose $x_1$ as the *pivot element*. Then, try the pivot element in Row 1:

\[
\begin{align*}
z - x_1 - x_2 &= 0 & \text{Row 0} \\
(2x_1) + x_2 + x_3 &= 4 & \text{Row 1} \\
x_1 + 2x_2 + x_4 &= 3 & \text{Row 2}
\end{align*}
\]

which yields:
with basic solution $x_2=x_3=0$, $x_1=2$, $x_4=1$, $z=2$. Actually, there is a rule that determines which row we should choose to pivot.

**Rule 2**  For each Row $i$, where there is a strictly positive "entering variable coefficient", compute the ratio of the Right Hand Side to the "entering variable coefficient". Choose the pivot row as being the one with MINIMUM ratio.

In the example, the ratio is 4/2 if we choose Row 1 and 3/1 if we choose Row 2. Therefore, Row 1 is our choice. If we choose Row 2 to pivot, we will end up having $x_3=-2$ in the basic solution, which violate the constraint $x_3 \geq 0$. After this Gauss-Jordan pivot, we have a new basic solution. Then we iterate between Rule 1 and 2 and pivot until Rule 1 guarantees that the current basic solution is optimal. According to Rule 1, $x_2=x_3=0$, $x_1=2$, $x_4=1$, $z=2$ is still not optimal. According to Rule 2, we should pivot the Row 2 on $\frac{3}{2}x_2$, which yields:

$$z + \frac{1}{6}x_3 + \frac{1}{3}x_4 = \frac{7}{3} \quad \text{Row 0}$$

$$x_1 + \frac{2}{3}x_3 - \frac{1}{3}x_4 = \frac{5}{3} \quad \text{Row 1}$$

$$x_2 - \frac{1}{3}x_3 + \frac{2}{3}x_4 = \frac{2}{3} \quad \text{Row 2}$$

with basic solution $x_3=x_4=0$, $x_1=5/3$, $x_2=2/3$, $z=7/3$. Now Rule 1 tells us this basic solution is optimal because there are no more negative entries in Row 0.
2.5 Related Work

In this section, existing research works on dynamic workload management and power optimization on processors are briefly summarized and discussed.

2.5.1 Dynamic Workload Management on CPU/GPU

Before GPU becomes a mature product as it is today, researchers have been working on scheduling algorithms custom-designed for multicore CPUs integrated with heterogeneous chip multiprocessors (CMPs) that specifically target on the workload balancing among the multiprocessors on runtime. As in [Arora et al. 2001] and [Blumofe and Leiserson 1999], a user-level thread scheduler with non-blocking work-stealing algorithm is modelled and thoroughly analyzed. As described in the algorithm, a process can “steal” threads from the ready queues of other processes in a non-blocking manner. Becchi and Crowley [Becchi and Crowley 2006] simulate a runtime mechanism for thread migration between the cores within a CMP. They argue that the dynamic assignment outperforms the static one on the heterogeneous CMP system. In [Jenks and Gaudiot 2003], researchers exploit thread migration for the purpose of remote memory access reduction in multithreading system. However, thread migration can also cause performance degradation because of the loss of cache state, stated by Brown et al. [Brown et al. 2011], who propose a technique that can prefetch appropriate data into the new caches for the migrated threads.

In the research of GPUs, Aila and Laine [Aila and Laine 2009] explore the efficiency of work distribution of ray tracing application on GPUs. The main idea of theirs is balancing workload on the warp level, using persistent threads to replace the terminated threads. Cederman et al. [Cederman and
Tsigas 2008] implemented four load balancing techniques, three centralized and one non-centralized, in the context of octree partitioning on GPUs. In their centralized techniques, a monolithic task queue is maintained and accessed either in blocking manner or non-blocking manner. But none of the three centralized techniques, according to their comparison, outperforms their non-centralized technique implemented based on Arora et al.’s CPU-based task stealing method [Arora et al. 2001] with respect to performance and scalability. In [Tzeng et al. 2010], Tzeng et al. also point out the significance of dynamic scheduling in managing irregular tasks on GPUs. They implement task donation technique, featuring less memory overhead than task stealing.

2.5.2 Power Optimization on CPU/GPU

2.5.2.1 To Reduce Static Leakage Power

Power gating, an architecture-level technique, is commonly applied in VLSI design. Powell et al. [Powell et al. 2001] propose an I-cache design that can dynamically resize using power gating technique to adapt to an application’s required size. In their design, the tradeoff on execution time exists and is analyzed. Flautner et al. [Flautner et al. 2002] put currently unused cache lines into a state-retentive, low-power drowsy mode, taking advantage of the phenomenon that the activity in a cache is only centered on a small subset of lines during a fixed period of time. But it still incurs around 1% runtime increase as its tradeoff. Processor manufacturers like Intel and AMD are also interested in incorporating power gating techniques into their products. Rusu et al. [Rusu et al. 2009] described the power gating implementation for SRAM array in Intel’s Xeon processor. They put two sleep transistors in parallel between $V_{dd}$ and SRAM array to create multiple low-leakage modes. Weiss et al. [Weiss et al. 2011]
proposed a design of L3 cache in 32nm with power gating technique integrated. All of the wordline and column select drivers in data macros are power gated in groups of eight. Performance tradeoff, however, is not discussed in either of their techniques.

2.5.2.2 To Reduce Dynamic Power

Researchers exploit DVFS schemes on several directions. Some are based on hardware [Semeraro et al. 2002][David et al. 2011]. Some are based on OS time interrupt [Lim et al. 2006][Choi et al. 2005], while some are based on compiler techniques [Isci et al. 2006][Wu et al. 2005]. Semeraro et al. [Semeraro et al. 2002] describe and evaluate an online, dynamic DFS algorithm for multiple clock domain processors that uses Attack/Decay technique to reduce the frequency for energy savings. The algorithm is claimed to be able to maintain consistent performance levels across different applications using only modest hardware resources. A slight degradation in performance, however, exists as the cost of the saving in energy. David et al. [David et al. 2011] aim at reducing the frequency of the memory system. Many workloads, as they discovered, suffer minimal performance impact when memory frequency is reduced. Lim et al. [Lim et al. 2006] present a message passing interface runtime system that dynamically reduces CPU performance during communication phases in MPI programs in order to save energy. Both methods in [David et al. 2011] and [Lim et al. 2006] are designed as “reactive” approaches, which directly respond to the application behavior. Therefore, the control latency in those methods has an impact on the performance. Similar as the method in [Lim et al. 2006], the technique that Choi et al. [Choi et al. 2005] propose aims at power saving during the CPU idle time due to the domination of memory accesses. However, a performance penalty is still involved. On the direction of
DVFS through compiler optimization, Wu et al. [Wu et al. 2005] present a prototype of DVFS mechanism that is implemented and integrated into a dynamic compilation system. Although their technique is more fine-grained and more code-aware, both hardware and code have to be modified. In [Isci et al. 2006], Isci et al. demonstrate a runtime phase predictor that analyses the history of branch predication of the running applications, and cooperates with the DVFS unit. However, keeping track of branch prediction results and making predictions require extra hardware support.
CHAPTER 3:

RUNTIME POWER GATING IN CACHES OF GPUS FOR LEAKAGE ENERGY SAVINGS

In this chapter, we propose a novel microarchitectural technique for run-time power gating caches of GPUs to save leakage energy. The L1 cache (private to a core) can be put in a low-leakage sleep mode when there are no ready threads to be scheduled, and the L2 cache can be put in sleep mode when there is no memory request. The sleep mode is state-retentive, which precludes the necessity to flush the caches after they are woken up. The primary reason for the effectiveness our technique lies in the fact that the latency of detecting cache inactivity, putting a cache to sleep and waking it up before it is accessed, is completely hidden microarchitecturally. The technique incurs insignificant overheads in terms of power and area. Experiments were performed using the GPGPU-Sim simulator on benchmarks that was set up using the CUDA framework. The power and latency modeling of the cache arrays for measuring the wake-up latency and the break-even periods is performed using a 32-nm SOI IBM technology model. Based on experiments on 16 different GPU workloads, the average energy savings achieved by the proposed technique is 52%.

3.1 Motivation

In the simulations of general-purpose benchmarks, the idleness of the GPU caches comes into our attention. In Figure 3.1, some data of the cache idle period is gathered. Red bars represent the
fraction of L1 idle time over total execution time, while the blue bars represent the fraction of L1 long (greater than continuous five cycles) idle time over total execution time. Figure 3.1(b) shows the statistics of L2 caches with the same format. It can be seen from these two figures that the idle cycles of L1 and L2 caches take up a considerable part of total execution time. Furthermore, many of the idle cycles (44% for L1, and 90% for L2) are within idle intervals longer than five cycles. The data in Figure 3.1 reveals that the general-purpose applications incur a considerably high cache missing rate on both L1 and L2. Similar as in the traditional CPU applications, random memory accesses are dominant in general-purpose GPU applications, which has a great tendency of cache misses.

**Figure 3.1.** Percentage of idle cycles in caches. (a) Percentage of cycles that L1 caches are idle and percentage of cycles that L1 caches are idle for >5 cycles; (b) Percentage of cycles that L2 caches are idle and percentage of cycles that L2 caches are idle for >5 cycles.

In order to save leakage power of caches for GPUs, our idea is to apply power gating technique on L1 and L2 cache arrays, especially during the long idle periods, because the power saving will be more significant and the mode switching activities and overhead will be less.
3.2 Proposed Microarchitectural Technique for Leakage Reduction on Caches Using Power Gating

![Cache Array Diagram]

**Figure 3.2.** Circuit implementation of power gated cache array [Rusu et al. 2009].

**Table 3.1. Three Leakage Modes of Cache**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Control Signal</th>
<th>Cache Status</th>
<th>Leakage Power Saved</th>
<th>State Retentive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cs1 cs2</td>
<td>Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>active</td>
<td>1 1</td>
<td>Normal</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>sleep</td>
<td>0 1</td>
<td>Stall</td>
<td>High</td>
<td>Yes</td>
</tr>
<tr>
<td>off</td>
<td>0 0</td>
<td>Stall</td>
<td>Maximum</td>
<td>No</td>
</tr>
</tbody>
</table>

A sketch of circuit implementation of our CPG technique is shown in Figure 3.2, two equally-sized sleep transistors, $M_{s1}$ and $M_{s2}$ are placed between the cache array and the ground. Based on the control signals: $cs1$ and $cs2$ supplied to the gates of the two transistors, the cache can be put into 3 modes: *active*, *sleep* and *off*. As shown in Table 3.1. In active mode, $M_{s1}$ and $M_{s2}$ are both turned on. Cache array works normally while the leakage is maximum. In *sleep* mode, $M_{s2}$ is turned off to narrow down the conducting path to the ground. It reduces a portion of leakage current and the contents in the cache arrays are still retained for later access. In *off* mode, both the sleep transistors are turned off. In
this mode, the leakage savings are maximum but the contents of the cache are lost. To make the decision of mode switching and supply the control signals, power gate controllers are added into GPU as shown in Figure 3.3. Our power gating strategy is briefly illustrated in Figure 3.4.

![Figure 3.3. Sketch of power gate controlling mechanism.](image)

![Figure 3.4. Power gating strategy. (a) Strategy for L1. (b) Strategy for L2.](image)

### 3.2.1 Power Gating L1 Caches of GPU

Each SMX privately owns an L1 cache array. Normally, the L1 cache is frequently accessed by the ALU of each core within the same SMX, except for 2 scenarios:

- **No Ready Warp to Be Scheduled** - A thread warp will get switched out by scheduler and wait for its next turn if one or more threads within this warp incur an L1 miss. A ready warp instead will be switched in for execution. Warps in waiting status will become ready again
once its required data gets fetched from lower level storage (such as L2 cache, or DRAM). If none of the warps within an SMX is ready at the current cycle, a *Stall* instruction will be issued into the pipelines. Functional blocks of SMX including L1 cache array will be idle once it encounters one or a series of *Stall* instructions. During the idle time, L1 is switched from *active* to *sleep* mode so that the leakage power can be saved while the contents are still kept for later accessing after the *Stall* ends.

- *Streaming Multiprocessor Finishes All Its Workload* - A SMX has one or more thread blocks to process during a kernel execution. Each SMX works on its own share of work distribution. Most of the time, GPU programs assign computation tasks fairly among SMXs. Even that, the SMXs still finish at different wall-clock times. When an SMX finishes all its assigned threads, it simply becomes idle and waits for other SMXs to finish. During that period of time, the L1 cache array of the finished SMX is switched from active to off mode to get the optimal power saving result. The loss of stored cache content is acceptable in this scenario because the content will not be used any more.

### 3.2.2 Power Gating L2 Caches of GPU

Compared with L1, L2 cache is less frequently accessed. So power gating L2 cache theoretically should yield better result on leakage saving. Similarly, we also use two sleep transistors to power gate L2, yet with one of them always on. It means only two modes are allowed: *active* and *sleep*. *off* mode is not applicable for L2 because L2 arrays are shared among all the SMXs. They can not be turned off as long as there is an SMX still running.
An L2 cache array is switched into *sleep* mode when its memory controller issues a *Nop* (no operation) command, which happens mainly because of three reasons:

- **The L2-DRAM Queue Is Full** - A memory request for data is sent by SMX on an L1 miss and it reaches L2 through interconnection network. When an L2 miss also happens, the memory request will be passed further down to DRAM, temporarily pushed into the L2-DRAM queue. If unfortunately the L2-DRAM queue is already filled up, the memory controller will keep issuing *Nop* until one of the queued requests gets response from DRAM and frees up a slot in the L2-DRAM queue.

- **The L2-Memory Controller Queue Is Full** - Once a memory request is passed all the way to DRAM and gets served, the data fetched from a DRAM will be sent to the corresponding L2 array, which is responsible for passing it up to the L2-memory controller queue. Then the data will be placed onto the interconnection network by the memory controller for transferring until it reaches the SMX that requests for it. However, if the L2-memory controller queue is full, L2 array can not place the current data fetched from DRAM. In this case the memory controller will issue *Nop* until that data can be pushed into the L2-memory controller queue.

- **There Is No Memory Request to Be Served** - Among all the reasons for issuing *Nop*, this is the one most commonly seen. Since L2 cache is a low level storage, the request is considerably infrequent and L2 array is idle because of this for most of the execution cycles.
3.2.3 Analysis of Latency Hiding

As the power gating technique saves us leakage power, its overhead in the performance aspect also raise a problem. There will be latency when a cache array is switching its mode among active, sleep and off. In our HSPICE simulation, we managed to limit the mode transition latency down to two nanoseconds through adjusting the size of SRAM arrays controlled by every sleep transistor pair. Given that the current generation of GPU products is usually working on a clock of giga hertz, the mode transition latency should be less than two cycles. Then the problem left is how to hide the 2-cycle latency in L1 and L2 caches when a mode transition happens. A brief analysis is shown in Table 3.2.

<table>
<thead>
<tr>
<th>Mode Transition</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>active→off</td>
<td>Transition latency does not negatively affect the performance because SMX has finished all its workload.</td>
</tr>
<tr>
<td>off→active</td>
<td>Transition latency overlaps with the grid initialization, which takes much more cycles.</td>
</tr>
<tr>
<td>active→sleep</td>
<td>Transition latency exists. But no performance degradation because the SMX stalls when this transition happens.</td>
</tr>
<tr>
<td>sleep→active</td>
<td>Transition latency overlaps with the last 2 cycles of the Stall.</td>
</tr>
<tr>
<td>active→sleep</td>
<td>Transition latency exists. But no performance degradation because the L2 is idle when this transition happens.</td>
</tr>
<tr>
<td>sleep→active</td>
<td>Transition latency overlaps with 1st 2 cycles of queue accessing (1st step L2 is on when back to active).</td>
</tr>
</tbody>
</table>

3.2.3.1 Hiding Transition Latency between off and active in L1

L1 cache is switched from active to off when the SMX it is located in finishes all its workload.

So there will be no performance degradation since the L1 array will not be active again. If a new kernel
starts its execution, the L1 cache can get woken up from off mode to active in the middle of grid initialization, which takes far more than two cycles. For the transitions between active and sleep, we may discuss it respectively in L1 and L2.

### 3.2.3.2 Hiding Transition Latency between sleep and active in L1

L1 is switched from active to sleep mode once a Stall is issued into the pipeline. The power gate controller will get the information directly from the warp scheduler whether a Stall is issued. There are three pipeline stages: instruction dispatch, register file read and execute between warp scheduling and memory stage (in which L1 cache should be active). It means there is at least a three-cycle interval between knowing whether a Stall is issued and power gate controllers sending out the control signal.

Our power gating strategy for L1 is: if a Stall of three or less than three cycles is issued into the pipelines, the power gate controller chooses not to power gate L1 during such Stall, since the stall interval is not long enough for switching to sleep mode and switching back to active. If a Stall of four cycles is issued, the power gate controller will go ahead switching the mode of the L1 cache without knowing the actual length of the stall interval. The L1 cache array can still be woken up in time, as long as the power gate controller puts L1 array back to active mode in the last two cycles of that Stall. However, the leakage power may not be saved much because we barely complete two mode switching operations within that Stall, so the L1 cache does not completely enter sleep mode. If the Stall intervals are greater than four cycles, the power gate controller will switch L1 to sleep, the same as it does for a four-cycle Stall. Only in this case, L1 can stay in sleep mode for some cycles, during which the leakage power of L1 cache will be effectively saved.
3.2.3.3 Hiding Transition Latency between *sleep* and *active* in L2

The activities of L2 and DRAM on the same channel are controlled by a memory controller. If the memory controller detects that its associated L2 array is not ready to process any data during a cycle, then a *Nop* command will be issued. L2 power gate controller keeps watching the commands from the memory controller. If the current command issued by memory controller is *Nop*, the L2 array is put into *sleep* and if not a *Nop*, the L2 stays on active mode.

L2 arrays obviously need to be operational at the first non-*Nop* command right after a *Nop*, yet the wakeup latency is two cycles. It won’t be a problem for our mechanism because when L2 is back to *active*, the first thing it is supposed to do is not reading/writing data in the power gated memory cells, but accessing buffering queues (either popping a memory request from L2-memory controller queue, or popping a fetched data from the L2-DRAM queue). This step normally takes much longer than 2 cycles. During this step, the corresponding queue will be accessed first, and then the L2 cache array will be accessed. Therefore, L2 array can be switched back to *active* in time.

3.3 Experimental Setup and Results

In order to test our power gate controlling strategy on transistor level, a timing and power model of the caches is built in HSPICE using 32-nm IBM technology [Service 2013].

3.3.1 Modeling of Power Gated SRAM Array

The power saving of a full-sized L1 and L2 array are estimated based on an HSPICE model built under 32-NM IBM technology [Service 2013]. According to the simulation of mode transition activity in
HSPICE, the latency of $\text{off} \rightarrow \text{active}$ transition, which is the worst case for mode switching latency, is 1.954 ns. Given that the current generation of GPU works at a clock frequency of 1GHz or higher, we assume the mode transition latency as two-cycle long in our technique. One pair of sleep transistors, as shown in Figure 3.5, controls a cache array of 32 bytes. Another thing we noticed is that the extra power cost introduced by cache entering and exiting low-leakage mode is much smaller than the power saving during the two-cycle mode transition. Therefore, the *breakeven period* of our technique is actually less than two cycles, which means our power gating technique always yields positive power saving even during mode transitions. As shown in Table 3.3, the leakage power due to subthreshold conduction in *active*, *sleep* and *off* modes is measured through HSPICE testing. In *sleep* mode, the leakage power of cache arrays is only 22.8% of that in *active* mode, while in *off* mode, the leakage power is reduced even more significantly down to 1.5% of that in *active* mode.

![Figure 3.5. Power gated SRAM array that we model in HSPICE.](image)

### 3.3.2 Benchmarks and Simulator Setup

The GPGPU-Sim [Bakhoda et al. 2009] simulator only provides us the total number of execution cycles, the total number of *Stall* and *Nop* cycles. To get more precise details, some other statistic data are needed, such as the length of each idle interval of L1, L2 cache arrays and the finishing
time of each SMX. All these statistics above are essential for the analysis of our power gating scheme. So we implement our own functions that work as performance counters during the simulation of execution and add them into the source code of GPGPU-Sim.

<table>
<thead>
<tr>
<th>Cache Mode</th>
<th>Leakage Power in SRAM Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>active</td>
<td>100%</td>
</tr>
<tr>
<td>sleep</td>
<td>22.8%</td>
</tr>
<tr>
<td>off</td>
<td>1.5%</td>
</tr>
</tbody>
</table>

Table 3.3. Leakage Power in Different Cache Modes

GPUWattch [Leng et al. 2013] is an energy model based upon McPAT and integrated with GPGPU-Sim. Driven by performance counters in GPGPU-Sim, GPUWattch is able to capture both program and microarchitectural level interactions. The power model we build inherits the empirical modules from GPUWattch, as shown in the Equation below:

\[
P_{\text{overall}} = P_{\text{static}} + P_{\text{dyn}}
\]

\[
= (P_{\text{proc, leak}} + P_{\text{mem, leak}} + P_{\text{VRM}} + P_{\text{peripherals}}) + (P_{\text{proc, dyn}} + P_{\text{mem, dyn}})
\]

The overall power of GPGPU is the sum of static power \( P_{\text{static}} \) and dynamic power \( P_{\text{dyn}} \). The \( P_{\text{static}} \) includes processor leakage power, memory leakage power, voltage regulation module power and the power consumed in all peripheral circuits. \( P_{\text{dyn}} \) consists of the dynamic power of both the processors and the main memory, which is proportional to the frequency. We make no modification on the power models of most of the components above, which are carefully built and validated in GPUWattch. Only
Table 3.4. Benchmark Description

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Encrypt a 256KB picture using 128-bit encryption. Memory access is intensive and cache misses are many.</td>
</tr>
<tr>
<td>RAY</td>
<td>Render graphics with near photo-realism. Thread behavior is mainly subjected to branch divergence. Memory access is frequent but cache misses are not many.</td>
</tr>
<tr>
<td>CP</td>
<td>Used in the field of molecular dynamics. Loops are manually unrolled to reduce loop overheads. L1 cache is intensively accessed yet miss rate is not high.</td>
</tr>
<tr>
<td>STO</td>
<td>A library that accelerates hashing-based primitives designed for middleware. Memory accesses are not significantly many. L2 miss rate is high.</td>
</tr>
<tr>
<td>LPS</td>
<td>A highly parallel finance application. Branch divergence is significant and takes up most of the execution time.</td>
</tr>
<tr>
<td>MUM</td>
<td>It matches query strings consisting of standard DNA nucleotides to a reference string. Cache misses are significant and branch divergence hurts the performance.</td>
</tr>
<tr>
<td>BS</td>
<td>A sorting algorithm. Programmed to run on a single core of the GPU.</td>
</tr>
<tr>
<td>HIS</td>
<td>Used for image processing and data mining. Memory access is highly intensive and cache miss rate is high.</td>
</tr>
<tr>
<td>MM</td>
<td>Illustrate GPU performance for matrix multiply. Cache miss rate is high.</td>
</tr>
<tr>
<td>SP</td>
<td>The memory access amount is fair. Cache miss rate is high.</td>
</tr>
<tr>
<td>ST</td>
<td>Performs a trivial 2D coordinate transformation using texture memory. L1 cache miss rate is high.</td>
</tr>
<tr>
<td>MT</td>
<td>A parallel version of Mersenne Twister for random number generation. Cache misses rate is fair, but number of memory accesses is large.</td>
</tr>
<tr>
<td>NN</td>
<td>Uses a convolutional neural network to recognize handwritten digits. Memory accesses and cache miss rate is low.</td>
</tr>
<tr>
<td>CS</td>
<td>A separable convolution filter implemented in NVIDIA CUDA. Memory accesses are very intensive and both L1 and L2 cache miss rate are high.</td>
</tr>
<tr>
<td>TRAN</td>
<td>Transpose of a matrix of float numbers. Both memory access amount and cache miss rate are fair.</td>
</tr>
<tr>
<td>BFS</td>
<td>Performs breadth-first search on a random graph with 65,536 nodes and an average of 6 edges per node. L1 miss rate is high. Memory access intensity and L2 miss rate are low.</td>
</tr>
<tr>
<td>FWT</td>
<td>Naturally-ordered Fast Walsh Transform for batched vectors of arbitrary eligible (power of two) lengths. Memory access intensity and L2 miss rate are low.</td>
</tr>
<tr>
<td>ID</td>
<td>Removes white noise from an image. The memory access amount is fair. L1 miss rate is high.</td>
</tr>
<tr>
<td>JPEG</td>
<td>A wavelet-based image compression algorithm. Computationally intensive. The memory access amount is small.</td>
</tr>
<tr>
<td>SHA1</td>
<td>A multi-cycle hardware version of SHA-1 that takes one 64-bit input. Computation-bound. Memory accesses are not many.</td>
</tr>
</tbody>
</table>

for the $P_{mem\_leak}$ part, we integrate our results from the tests on the HSPICE cache model. Whenever a cache array enters sleep, or off mode, leakage power of it will be scaled down to the corresponding
percentage in Table 3.3. The dynamic power that is involved in cache entering and exiting sleep mode is not added into the memory dynamic power due to its insignificance that is discovered in HSPICE cache model tests.

In our simulations, twenty CUDA programs are used as our benchmarks, most of which are from NVIDIA’s CUDA software development kit (SDK) [NVIDIA 2010]. Nine of them are used by the developer of GPGPU-Sim in [Bakhoda et al. 2009]. Two of them are from [Chang et al. 2010]. A brief description of the benchmarks is shown in Table 3.4.

To simulate the architecture of NVIDIA GTX480, configurations of GPGPU-Sim are set up as in Table 3.5.

**Table 3.5. Architecture Comparison**

<table>
<thead>
<tr>
<th>Specification</th>
<th>GTX 480</th>
<th>Our Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Clock</td>
<td>700MHz</td>
<td>700MHz</td>
</tr>
<tr>
<td>Memory Speed/Clock</td>
<td>924MHz</td>
<td>L2: 700MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DRAM: 924MHz</td>
</tr>
<tr>
<td>Number of SMX’s</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Number of cores per SMX</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>Register size per SMX</td>
<td>128KB</td>
<td>128KB</td>
</tr>
<tr>
<td>L1 size per SMX</td>
<td>48KB</td>
<td>48KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-way set assoc.</td>
</tr>
<tr>
<td>Shared memory size</td>
<td>16KB</td>
<td>16KB</td>
</tr>
<tr>
<td>Memory bus width per channel</td>
<td>64 bits</td>
<td>64 bits</td>
</tr>
<tr>
<td>Number of memory channels</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>L2 total size</td>
<td>768KB</td>
<td>768KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8-way set assoc.</td>
</tr>
</tbody>
</table>

36
3.3.3 Evaluation and Analysis

Table 3.6 shows the result of benchmark simulation. We list the average number of L1 cache arrays that are solidly in sleep mode per cycle, the average number of L1 cache arrays that are solidly in off mode per cycle, and the average number of L2 cache arrays that are solidly in sleep mode per cycle based on our power gating mechanism. Then the percentage of leakage power saved by power gating L1 and L2 is calculated. Table 3.6 reveals us some meaningful information:

Table 3.6. Leakage Saving Result

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>L1 (15 arrays in total)</th>
<th>L2 (6 arrays in total)</th>
<th>Total Leakage Saving (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cache Arrays in sleep per Cycle</td>
<td>Cache Arrays in off per Cycle</td>
<td>Leakage Saving (%)</td>
</tr>
<tr>
<td>AES</td>
<td>1.7862</td>
<td>0.8473</td>
<td>14.08</td>
</tr>
<tr>
<td>RAY</td>
<td>0.4112</td>
<td>0.9883</td>
<td>8.21</td>
</tr>
<tr>
<td>CP</td>
<td>0.0371</td>
<td>4.1368</td>
<td>26.18</td>
</tr>
<tr>
<td>STO</td>
<td>0.0499</td>
<td>0.2862</td>
<td>2.06</td>
</tr>
<tr>
<td>LPS</td>
<td>0.0637</td>
<td>2.7018</td>
<td>17.32</td>
</tr>
<tr>
<td>MUM</td>
<td>3.6539</td>
<td>1.8387</td>
<td>30.53</td>
</tr>
<tr>
<td>BS</td>
<td>0.0034</td>
<td>14.2061</td>
<td>95.70</td>
</tr>
<tr>
<td>HIS</td>
<td>0.3384</td>
<td>0.0273</td>
<td>1.92</td>
</tr>
<tr>
<td>MM</td>
<td>0.9703</td>
<td>5.4954</td>
<td>37.82</td>
</tr>
<tr>
<td>SP</td>
<td>3.3971</td>
<td>3.1300</td>
<td>36.39</td>
</tr>
<tr>
<td>ST</td>
<td>0.3278</td>
<td>0.2900</td>
<td>3.52</td>
</tr>
<tr>
<td>MT</td>
<td>0.2033</td>
<td>10.6738</td>
<td>73.05</td>
</tr>
<tr>
<td>NN</td>
<td>0.4222</td>
<td>0.4437</td>
<td>4.83</td>
</tr>
<tr>
<td>CS</td>
<td>0.5096</td>
<td>0.0449</td>
<td>2.83</td>
</tr>
<tr>
<td>TRAN</td>
<td>2.9683</td>
<td>5.0556</td>
<td>45.75</td>
</tr>
<tr>
<td>BFS</td>
<td>1.6131</td>
<td>4.5683</td>
<td>37.61</td>
</tr>
<tr>
<td>FWT</td>
<td>3.8006</td>
<td>0.9541</td>
<td>24.59</td>
</tr>
<tr>
<td>ID</td>
<td>0.6197</td>
<td>0.6660</td>
<td>7.46</td>
</tr>
<tr>
<td>JPEG</td>
<td>2.8643</td>
<td>2.2502</td>
<td>28.17</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.0014</td>
<td>13.9707</td>
<td>88.54</td>
</tr>
<tr>
<td>Average</td>
<td>1.2021</td>
<td>3.6288</td>
<td>29.2778</td>
</tr>
</tbody>
</table>
● On average, more than 8% of L1 cache arrays (1.2021 out of total 15) can be kept in sleep mode and more than 24% of L1 cache arrays (3.6288 out of total 15) can be kept in off mode during execution. The two parts altogether render a 29% of leakage power saving, which is a considerable amount for L1 cache alone.

● The portion of L1 cache arrays that are in off mode during execution is greater than 24%, which indicates a significant variation of the SMXs’ finishing times. This phenomenon justifies the necessity of having an off mode in our technique to power gate L1 cache arrays in early-finished SMXs.

● L2 cache, as a lower-level storage device, is idle in most of the execution time. Approximately 82% (4.9066 out of total 6) of L2 cache arrays on average can be placed into sleep mode at any time of the execution, which gives us a 63% of leakage power saving on L2. The benefit of power gating L2 cache arrays is therefore much greater than that of power gating L1 cache arrays.

The main targets of our power gating technique are the GPU programs with long periods of memory stalls. In some extreme cases like BS and SHA1, the program itself does not even require all the SMXs to work. The leakage power saving for them will be very significant. Some other examples, such as CP and MT, have a lot of memory stalls during execution and most of the stalls are long ones. Therefore, the leakage saving they yield is also considerably large. However, if the long intervals of stall are not the majority of cache idle periods, such as in BFS and FWT, the leakage saving will not be so decent because the power gating operation is triggered only if the stall is long enough.
The design of power gate controllers in this technique is fairly simple in terms of the logical complexity: the input signal of power gate controller of L1 cache is a 1-bit Stall flag indicating whether a Stall is issued in the current cycle. To keep track of the length of Stall between warp scheduling and memory stage, a counter is needed that simply counts from 0 to 4. Therefore, a 1-bit input, 1-bit output (cs1) sequential logic circuit with a 2-bit counter is enough to control the transitions between active and sleep. To control the transitions between active and off, 1-bit output is enough for both cs1 and cs2, since their values are always the same. Then a 1-bit signal is needed to indicate whether an SMX finishes its workload and another one bit to indicate whether grid initialization starts. For L2 cache arrays, We use 1-bit signal to indicate whether the currently-issued command is Nop or not. A 1-bit output will be used to set and clear the cs1. The power consumption of our power gate controllers is negligible due to the small scale of the circuit compared with the power gated circuit block (size of one L1 cache array is 48Kb and size of one L2 cache array is 96Kb).

3.4 Conclusion

In this chapter, we propose a cache power gating technique to save the leakage power of L1 and L2 caches in the GPU. We design three working modes for caches. The strategy of manipulating the sleep transistors is based on the functional properties of L1, L2 and other relevant modules in GPUs. Analysis is presented, from which we conclude that our latency hiding scheme ensures no negative effect on performance. Simulation results show that the average savings of leakage power is 52%. For future work, we want to extend our power gate controlling mechanism into a more fine-grained version. Currently, we are power gating the L1 or L2 by the unit of the whole array. If we can manage to bring
the granularity of power gating technique down to each individual cache line within an L1 or L2 cache array, we will have opportunities to save more leakage power by only waking up one or a few cache lines that will be accessed in the next moment, while the rest of cache lines can still stay in power gated mode.
CHAPTER 4:
LOAD BALANCING ON THE RUNTIME FOR GPUS

In this chapter, an effective microarchitectural technique, Dynamic Workload Balancing (DWB), for optimizing performance of GPUs is investigated in the context of general-purpose computing in GPUs (GPGPU). It monitors the workload on each multi-processing unit and redistributes it by having a portion of its unfinished threads executed in a neighboring multi-processing unit in order to improve performance. To accomplish this, a microarchitecture redesign based on shared structures is proposed. DWB also includes careful considerations for preventing deadlock scenarios and avoiding potential errors during the execution that involves synchronization and shared variable access. Experimental results of twenty GPGPU applications indicate that DWB improves their performance of by 15.69%.

4.1 Motivation

To understand the motivation for this technique, consider the hypothetical example shown in Figure 4.1. In the figure, execution profiles of two types of workload on a GPU with two SMXs are presented. In the case of workload 1, SMX A finishes execution in 50 cycles and SMX B finishes execution in 100 cycles. In the case of workload 2, SMX A finishes execution in 70 cycles and SMX B finishes execution in 80 cycles. Even though both types of workload require equal amounts of total compute time (of 150 cycles), the latter finishes faster because the two SMXs are loaded in a more
balanced fashion than the former. In other words, workload 2 makes a more efficient use of the SMXs than workload 1 does.

![Figure 4.1](image)

Figure 4.1. Example of finishing time deviation. (a) GPU finishes workload 1 on cycle 100. cv of SMX finishing times is 47.2%; (b) GPU finishes workload 2 on cycle 80. cv of SMX finishing times is 9.5%.

In order to capture such load imbalance in workload, we use the coefficient of variation, $c_v$, of the SMXs’ finishing times. $c_v$ is defined as:

$$c_v = \frac{\sigma}{\mu}$$

where $\sigma$ is the standard deviation and $\mu$ is the mean value of SMXs’ finishing times. In the case of the illustrative example, with $\mu = 75$ for both types of the workload,

$$c_v_{\text{of workload 1}} = \frac{\sqrt{(50-75)^2 + (100-75)^2}}{75} = \frac{2-1}{75} = 47.2\%$$

$$c_v_{\text{of workload 2}} = \frac{\sqrt{(70-75)^2 + (80-75)^2}}{75} = \frac{2-1}{75} = 9.5\%$$

In real world GPGPU applications, the imbalance of workload distribution across SMXs can be significant. A profile of the $c_v$ of the finishing times of SMXs for a high-end discrete GPU model across
a set of CUDA benchmarks is shown in Figure 4.2. This profile is created with GPGPU-Sim [Bakhoda et al. 2009] and the details of the CUDA benchmarks and the discrete GPU model are described earlier in Section 3.2.2. As plotted in the figure, the $c_v$ of 10 out of the 20 benchmarks is greater than 20%. For 3 of the benchmarks – MM, BFS and JPEG – $c_v$ is even around 40%. DWB monitors such workload imbalance during the execution of an application and redistributes the unfinished warps from an overloaded SMX to the neighboring one, thereby making more efficient use of the SMXs.

![Coefficient of variation $c_v$ of SMXs original finishing times.](image)

**Figure 4.2.** Coefficient of variation $c_v$ of SMXs original finishing times.

### 4.2 Methodology and Implementation

In this section, we first present an overview of the proposed DWB technique. A microarchitecture design for DWB based on shared structures is then discussed. After that, the algorithm of DWB is elaborated, with consideration for preventing deadlock scenarios and avoiding potential errors during the execution that involves synchronization and shared variable access.
4.2.1 Overview of Dynamic Workload Balancing

A high-level overview of the approach at the center of DWB is as follows:

- SMXs are grouped into pairs. For each pair of SMXs, a DWB controller monitors the number of unfinished warps on both SMXs.
- If the number of unfinished warps on an SMX less than a threshold, while the same on the other SMX is more than the threshold, reassign a fraction of the unfinished warps assigned from the latter SMX to the former as long as doing so does not violate the correctness of program semantics.
- Repeat steps 1 and 2 until there are fewer than threshold number unfinished warps on both SMXs.

For this approach to be effective, the parameters, threshold and fraction, above need to be selected so that the load balancing can be improved, thereby achieving overall performance improvement. These parameters should not make DWB so conservative that it misses out a large number of opportunities to reassign warps across SMXs that would be beneficial. At the same time, it is important to make sure that the warps are not being continuously shuffled between the SMXs so aggressively that the technique reaches the point of diminishing returns.

In this work, the optimal values of $T_{\text{complete}}$ and $\alpha$ are determined empirically by running all the GPGPU benchmarks on GPGPU-Sim for various values of these parameters. This is described later in Section 4.3.2. Further, the technique needs to make sure that any changes to the original assignment of warps to SMXs continue to maintain the semantic correctness of the programs. Such scenarios can occur
when the workload has synchronization code, or any accesses to the shared variables. This is described in more detail in Section 4.2.3.

4.2.2 Microarchitecture Design

Each SMX within GPU has its independent hardware unit for warp scheduling, including the SRAM that stores the scheduling queue and the context of all its warps. To reassign warps between SMXs, we need to either add extra connections between the warp schedulers of the two SMXs, which introduces much overhead on synchronizing the communication, or let the two neighboring SMXs share the same storage for scheduling queue and warp context, with dual-port SRAM applied.

Synchronous dual-port SRAMs (DPRAMs) [Nii et al. 2009][Ishii et al. 2010], unlike single-ported ones, allow two access operations to occur at the same time. They are designed to serve the need for faster devices and simpler interfaces in the next generation of high-speed applications. The state-of-the-art DPRAMs have two separated sets of address decoding and I/O controlling systems, which are independent of each other. It ensures the true simultaneous accesses from both ports. An interrupt semaphore arbitration block is used to solve the data collisions when a read and a write or two writes happen in the same location of the SRAM array at the same cycle. In order to support high-speed scheduling operations, DPRAMs need to have short accessing time. Fortunately, in GPU’s SIMT architecture, similar as in SIMD, the access rate of warp scheduling queue need not to be as high as the access rate of data. Current researches on DPRAM have brought the write access time down to as low as 3.0 nanoseconds with 65 nm CMOS [Nii et al. 2009], and 1.4 nanoseconds with 28 nm CMOS technology [Ishii et al. 2010]. With GDDR5 high-performance, low-power SRAMs [NVIDIA 2010]
hitting the market, data transfer rate is even doubled compared to DDR3 RAMs, hence the conventional
memory clock frequency can be brought down to another level, which is in favor of the usage of
multi-ported SRAMs.

**Figure 4.3.** Sketch of hardware implementation of DWB. (a) When DWB controller is not activated, 
ordinary scheduling takes places on both SMXs. (b) When DWB controller is activated, DWB controller 
reassign warps from one SMX to the other.
As shown in Figure 4.3, we use a dual-port content-addressable memory (CAM) to store the warp scheduling queues for the two neighboring SMXs. Each entry of the warp scheduling queue contains many fields, among which there are several fields that are relevant to DWB: sid field indicates which SMX this warp belongs to. R field indicates whether all the threads of this warp are ready to be scheduled. S field is used to mark if the threads of this warp access the shared variable in their original SMX. warp index field is the address the corresponding warp context. The warp context of both SMXs is also stored together in a dual-port SRAM. After the warp scheduler of SMX a or b locates the next ready warp, the corresponding warp context will be found in the dual-port SRAM based on the warp index and output to the scheduler. The DWB controller is in charge of reassigning the warps from one SMX to the other when triggered by the unfinished warp counter of unfinished warps in either SMX.

- **When DWB controller is not activated** - The DWB controller is not activated when the number of the unfinished warps in neither SMX a nor b has reached the threshold $T_{\text{complete}}$. During that period, the ordinary warp scheduling takes place in both SMX a and b. For instance, the scheduler from SMX a requests the next ready warp in dual-port CAM via a search word with sid field "0" and R field "1". After the entry that meets this requirement is found, the warp index will be used to locate the warp context in the dual-port SRAM. Then the warp context is loaded by the scheduler of SMX a. The dual-port CAM and dual-port SRAM provide read and write ports for both SMXs and support the access to the scheduling queue and warp context for SMX a and b simultaneously.

- **When DWB controller is activated** - If the number of the unfinished warps of SMX a reaches $T_{\text{complete}}$, while the number of the unfinished warps in SMX b does not. The DWB
controller is then activated to reassign a certain number of unfinished warps from SMX \( b \) to \( a \). Specifically, the warp scheduling in SMX \( a \) will be paused and the read and write ports for SMX \( a \) are taken over by DWB controller. Then, the DWB controller goes through the dual-port CAM, changing the \( sid \) fields of the warps that are selected to be reassigned. Warp context in dual-port SRAM remains unchanged.

### 4.2.3 Dynamic Workload Balancing Algorithm

![Figure 4.4. Workload balancing between SMXs. (a) SMX A is approaching its completion. (b) \((n-m)\alpha\%\) of unfinished warps are reassigned from SMX B to SMX A.](image)

In the context of the detailed microarchitectural view proposed for DWB, the algorithm for workload balancing is as follows: warp scheduler in each SMX dynamically monitors its warp scheduling queue. As shown in Figure 4.4, if the total number of unfinished warps on SMX \( A \) is less than the pre-set threshold \( T_{\text{complete}} \), which means it is approaching to completion of execution, the DWB controller gets signaled. If the number of unfinished warps in SMX \( B \) is not less than \( T_{\text{complete}} \), then \((n-m)\alpha\%\) (\( n \) being the number of unfinished warps in SMX \( B \), \( m \) being the number of unfinished warps in SMX \( A \), \( \alpha \) being a factor parameter) unfinished warps will be reassigned from the tail of SMX \( B \)’s
scheduling queue to the tail of SMX A’s scheduling queue. Given that only the tails of scheduling queues are modified, the normal scheduling operations, which take place on the head of scheduling queue, will not be affected.

### 4.2.3.1 Memory Request Handling

![Diagram](image)

**Figure 4.5.** Redirecting memory request to avoid deadlock. (a) Memory request is generated via warp \(i\). (b) Warp \(i\) is reassigned from SMX \(a\) to SMX \(b\). (c) Deadlock threat exists when the reservation of resource cannot be released. (d) Resolving the memory request via redirection.

The handling of memory requests that are already sent by the threads before reassignment is depicted in Figure 4.5. The format of a memory request, as shown in Figure 2.2, contains SMX id, warp id, the address of requested data or the fetched data values. Once a memory request sent by a reassigned warp has the data fetched back to the original SMX, SMX looks up its scheduling queue and learns that the requesting warp is reassigned. Then the original SMX will change SMX id of that memory request to the id of the correct SMX and send the request back down to interconnection network. The request will
finally be received by the SMX where the requesting warp is currently located, as shown in Figure 4.5. This memory request redirect mechanism ensures that the previously-generated memory request will be handled and the reassigned warps will not have to send redundant memory requests from their newly-residing SMXs. So the control messages between cache and memory will not be redundantly doubled or left unanswered.

It is beneficial not only for the traffic efficiency of control messages, but also for the avoidance of deadlock situation caused by such warp reassignment. Warps waiting in the scheduling queue of an SMX are mostly in such cache-miss waiting status. If a warp on this status is reassigned, it is considered deadlock-unsafe, compared with the waiting warps due to computation-time-exhausted, in which case either the resource needed by threads is already acquired and backed up in thread control blocks, or the reservations for the requested resource are all released before the thread warp is switched out. Once a warp generates a memory request and is put to waiting status, the memory request will travel through several hardware domains and return after a relatively long period of time. During that time, the requested data are logically considered as reserved by the requesting thread of the warp. By redirecting the result of the request to the new location of the reassigned warp, we can make sure that the resource reservation can be resolved in the end, instead of having it suspended and posing a threat of deadlock.

4.2.3.2 Synchronization and Shared Memory Access

It is possible that mapping a warp to a new SMX may not guarantee program correctness any more and, in such cases, the original warp-to-SMX mapping needs to be restored. We have identified two scenarios that fall under this category and they are described below:
- **Thread Synchronization** - The CUDA instruction `__syncthreads()` acts as a barrier at which all threads within a thread block must wait before they proceed. From the hardware perspective, it synchronizes all the threads within an SMX. In the context of CUDA, `__syncthreads()` calls in source code get compiled into instruction bar.sync on the SMX-level of execution. Therefore, when a thread warp reassigned from one SMX is having bar.sync as the next instruction, it has to wait for the threads from its original SMX at that synchronization point. Given that the hardware taking care of synchronization mechanism exists only locally within each SMX, we send the reassigned warps back to their original SMXs where the coordination with other threads can be done conveniently and at a low cost. Warps that are currently waiting at a synchronization point will certainly not be selected to reassign.

- **Shared Variable Accessing** - The shared memory and L1 cache are different partitions of the same high-speed storage unit within an SMX. As introduced in the background section, threads can access data in shared memory loaded by other threads within the same thread block. Specifically, in the application code, variables declared using `__shared__` quantifier, are stored in the shared memory space. For any of the threads from a reassigned warp, once a decoded instruction turns out to be a shared variable accessing one, it makes more sense to restore the whole warp to its original SMX and access the shared memory locally, other than copying the content of its original shared memory to the newly-residing SMX, which definitely involves more expense on hardware and tradeoff on performance.
Note that the warps restored due to synchronization purpose can still become candidates for reassigning between SMXs when the synchronizing point is passed by all threads in other warps. However, the warps restored due to accessing the shared variables are marked as *dirty* in our DWB mechanism and will not be selected to reassign again. It is because the shared variables, once accessed during the execution of those threads, may just be accessed again. Given the characteristic of shared memory, variables declared as `__shared__` are mostly used in a loop for caching a continuous piece of global memory or unified operations towards themselves, where the shared memory will probably be accessed frequently for a large number of times. Therefore, letting the warps that access shared variables stay locally within their SMXs should yield better efficiency.

Branch divergence will not be a problem for the DWB at all. As CUDA architecture is defined [NVIDIA Corporation 2007], branch divergence occurs only among the threads within a warp. In other words, different warps execute independently regardless of whether they are executing common or disjointed code paths. Hence we are free to reassign warps to other SMXs without worrying about the coordination of execution paths with other warps.

### 4.3 Experimental Results and Analysis

Same as in chapter three, the timing simulations are set up on GPU simulator GPGPU-Sim. It performs NVIDIA parallel thread execution (PTX) on a massively parallel architecture with highly programmable pipelines. Twenty CUDA programs used in chapter three are used as our benchmarks here. Configurations of GPGPU-Sim are set up as in Table 3.5 to simulate the architecture of NVIDIA GTX480. In order to simulate the DWB technique, custom-made functions, which monitor each SMX of
the GPU and take care of the warp reassigning operations, are added to the loop body of the GPU execution. In our model, the latency of reassigning one warp between queues of two SMXs within a dual port SRAM is comprised of one read latency and one write latency.

### Table 4.1. Performance Improvement Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>DWB technique applied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Warps per 1000 Cycles</strong></td>
<td><strong>Warps per 1000 Cycles</strong></td>
</tr>
<tr>
<td>AES</td>
<td>133.08</td>
<td>137.211</td>
</tr>
<tr>
<td>RAY</td>
<td>12.591</td>
<td>13.248</td>
</tr>
<tr>
<td>CP</td>
<td>6.13</td>
<td>7.652</td>
</tr>
<tr>
<td>STO</td>
<td>30.75</td>
<td>31.070</td>
</tr>
<tr>
<td>LPS</td>
<td>6.349</td>
<td>7.710</td>
</tr>
<tr>
<td>MUM</td>
<td>1.886</td>
<td>2.070</td>
</tr>
<tr>
<td>BS</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>HIS</td>
<td>0.9534</td>
<td>0.9537</td>
</tr>
<tr>
<td>MM</td>
<td>19.603</td>
<td>21.033</td>
</tr>
<tr>
<td>SP</td>
<td>22.043</td>
<td>26.500</td>
</tr>
<tr>
<td>ST</td>
<td>3.684</td>
<td>3.695</td>
</tr>
<tr>
<td>MT</td>
<td>0.046</td>
<td>0.068</td>
</tr>
<tr>
<td>NN</td>
<td>3.042</td>
<td>3.793</td>
</tr>
<tr>
<td>CS</td>
<td>0.05373</td>
<td>0.05376</td>
</tr>
<tr>
<td>TRAN</td>
<td>4.924</td>
<td>5.186</td>
</tr>
<tr>
<td>BFS</td>
<td>0.0928</td>
<td>0.154</td>
</tr>
<tr>
<td>FWT</td>
<td>17.217</td>
<td>18.049</td>
</tr>
<tr>
<td>ID</td>
<td>5.925</td>
<td>6.409</td>
</tr>
<tr>
<td>JPEG</td>
<td>74.813</td>
<td>101.186</td>
</tr>
<tr>
<td>SHA1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Mean Value</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.1 Performance Improvement Result

In Table 4.1, we demonstrate the performance improvement brought by DWB in the form of how many warps can finish execution within 1000 cycles when DWB technique is applied with $T_{complete}$.
= 20 and \( \alpha = 20 \). Along with the performance improvement, the ratio of reassigned warps is also shown. As we can see, there are 12 out of total 20 benchmarks having more than 5% of performance improvement with DWB applied. The average performance improvement among 20 benchmarks is 15.69%. BS and SHA1 hashing are programmed to have only one SMX execute the kernel, so there is no performance improvement for either of them.

![Comparison of \( c_v \) with and without DWB.](image)

In Figure 4.6, the \( c_v \) values of each benchmark with and without DWB applied are presented. Red bars, same as in Figure 4.2, are the finishing time \( c_v \) without DWB, which reflect how balanced the workload is distributed originally. The data with DWB applied is represented by blue bars, which indicates that the finishing time \( c_v \) is greatly reduced since DWB aims at having SMXs get the similar amount of work in total. One thing needs to be clarified: improvement on \( CV \) with DWB applied does not necessarily lead to proportional improvement in performance. The finishing time \( c_v \) depends on the finishing times of all SMXs, while the performance improvement, shown in Table 4.1, only depends on the last finished SMX.

54
4.3.2 Choosing Optimal Values for threshold and fraction

The two parameters, threshold $T_{\text{complete}}$ and fraction $\alpha$, are determined by the computing power of each SMX as well as the type of computation load the GPU mainly deals with. To get the $T_{\text{complete}}$ and $\alpha$ that fit a specific scenario, it requires statistical tests on a vast number of benchmarks, ideally with all possible combinations of $T_{\text{complete}}$ and $\alpha$ tested. Then apply the combination that yields the largest average saving of total cycles. For the purpose of demonstration, in our simulation, the test of this kind is carried on using 20 benchmarks that we have. Meanwhile, instead of testing all the combinations of $T_{\text{complete}}$ and $\alpha$, some combinations of typical values are chosen. The results are plotted in Figure 4.7.

![Graph showing total execution times under different combinations of $T_{\text{complete}}$ and $\alpha$.]

**Figure 4.7.** Average total execution times under different combinations of $T_{\text{complete}}$ and $\alpha$.

The trend of these curves can be comprehended as: when $\alpha$ is too small, the number of warps reassigned during each reassignment is less than supposed to. Therefore, more reassignments will be triggered later on to make it up. The overhead of those reassignments will accumulate and make the total execution time long. When $\alpha$ is too large, reassigned warps are more than the helper SMX can handle. Eventually, the helper SMX may have to call for the other SMX to help with its excessive workload, which adds up to the negative effect upon total execution time. As shown in Figure 4.7, the curve of
average execution time reaches its lowest pole point at $T_{\text{complete}} = 20$, $\alpha = 20$ in our test. To tune $T_{\text{complete}}$ and $\alpha$ in real GPUs, GPU vendors need to carry on such tests more thoroughly with much more benchmarks.

4.3.3 Result Analysis

As shown in both Table 4.1 and Figure 4.6, DWB has different improvement effect upon these benchmarks. We look into each benchmark from several aspects and make comparisons among them. We found that there are two characteristics of GPU programs that make this improvement difference happen.

Figure 4.8. The number of unfinished warps comparison for BFS. (a) BFS execution without DWB; (b) BFS execution with DWB applied.

Figure 4.9. The number of unfinished warps comparison for HIS. (a) HIS execution without DWB; (b) HIS execution with DWB applied.
4.3.3.1 Warp-level Parallelism

Some of the benchmarks, due to the ways they are programmed, tend to have more balanced workload during execution while the others do not. In Figure 4.8 and Figure 4.9, we present some dynamic workload information of benchmark BFS and HIS, which respectively yields the best and the worst saving on the total execution time among all 20 programs (BS and SHA1 are excluded because of their non-parallel execution mode). The two charts within the same figure demonstrate the amount of unfinished workload on each SMX during the benchmark execution with and without DWB applied. Workload status in both charts is captured at the same cycle after which no more warp reassignment has occurred in the DWB-applied scenario. From the chart (b) of Figure 4.8, we can see that not only the workload is more balanced among all the 15 SMXs, the execution progress of each SMX is also ahead of time, compared with the chart (a). For HIS, the execution is fairly balanced among the SMXs in the original case, as shown in Figure 4.2. So there is not that much improvement space left for DWB. In Figure 4.9, chart (b) does show a more balanced workload distribution and less unfinished workload than chart (a), however not significant. An interesting point is that for HIS, the execution is far from completion when the warp reassignment stops from happening, which indicates that the execution of HIS on each SMX progresses in a very balanced manner from then on. In BFS, warp reassignment stops happening when the unfinished warps are around 80 on each SMX, close to our pre-set $T_{\text{complete}} = 20$. It means that DWB is in effect for most of the execution time of BFS, and the reason why warp reassignment stops happening in BFS is not because the execution of BFS is originally balanced, but the threshold $T_{\text{complete}}$ is reached by most of the SMXs, meaning the SMXs are near completion and need no help from others anymore.
### 4.3.3.2 Average Memory Access Time

![Figure 4.10](image)

**Figure 4.10.** Performance improvement tends to be large when \( c_v \) is large, and when AMAT is small. (a) \( c_v \) Original and \( c_v \) with DWB applied; (b) AMAT Original.

Some statistics about average memory access time (AMAT) are collected and shown in Figure 4.10(b), together with performance improvement and \( c_v \) of each benchmark in Figure 4.10(a) (BS and SHA1 are removed because they are single-SMX applications and have no performance improvement). A rough trend would be: as the AMAT goes down, the performance improvement gradually increases up; as the original \( c_v \) goes up, the performance improvement goes up. The relation between \( c_v \) and performance improvement has been explained at the end of Section 4.3.3.1. The original finishing time generally indicates how balanced their workload originally is. The more unbalanced it is, the more likely it can be improved a lot with DWB applied. About the AMAT trend, we can see those benchmarks that greatly benefit from our balancing algorithm, such as BFS, MT and NN, tend to have shorter average memory access time than the others. Benchmark HIS, CS and FWT, for example, with longer average memory access time do not gain much improvement from DWB. The reason is that workload balancing yields the optimal result when those reassigned warps are not currently waiting for data to be fetched from memory. If warps reassigned to a new SMX can get issued and processed instantly, the reassignment will be more meaningful and total execution time will be reasonably saved. However, if
unfinished warps are not ready to be scheduled because of cache misses, then reassigning them to another SMX simply means letting them wait in a new location and does no good to the overall performance.

To put in a nutshell, the original degree of workload balance and AMAT are the two factors that have impact on the performance improving effect brought by DWB algorithm. But it is difficult to say which one is dominant. For example, RAY and AES have a similar workload balance degree, indicated by their original $c_i$ values. But AMAT of AES is much higher than RAY, which explains why the performance improvement of RAY is more than that of AES. However, in the comparison between AES and SP, although their AMAT are similar, the workload in AES is originally way more balanced than in SP. Therefore, SP takes a great deal of advantage from DWB while AES does not.

4.3.4 Overhead Analysis

The overhead introduced by DWB is small, yet not negligible. From the hardware perspective, the utilization of DWB controllers and the upgrade of SRAMs from single-port to multi-port brings about more expenses, although the total number of ports between registers files and instruction cache is not changed at all. Theoretically, the size of the multi-port instruction cache should be $n$ times as the original, if it is shared by $n$ SMXs.

To address the content in such a space, $\log(n)$ extra bits are needed in the addressing register. However, it will not require any extra addressing register. Given that the register files in current GPUs are 32-bit wide and the length of an instruction in the CUDA instruction set is normally 4 to 8 bytes [NVIDIA Corporation 2007], one register file can hold an address for a cache space up to 16~32 giga
bytes, yet the size of instruction cache for the current generation of GPUs is only measured by kilo byte. So the mechanism for original instruction addressing should have the capacity for increased cache space without extra hardware support. From the performance aspect, the warp reassigning operations and the corresponding maintenance of scheduling queues causes the scheduling of one of the SMXs to pause. In our algorithm, the execution in the helper SMX is interrupted. Fortunately, the helper SMX having some extra delay does not directly or even definitely leads to a delay in total execution of GPU because total execution time is determined by the last finished SMX. In one of the columns of Table 4.1, for each benchmark, we show the ratio of the number of warps that are subjected to reassignment during execution over the total number of warps that are scheduled to run. As we can see, the percentages are all less than 10%. It indicates that on average, at the cost of 6.73% extra burden on scheduling queue accessing, our DWB manages to improve the performance of these benchmarks by 15.69%.

Another overhead of DWB technique is extra cache misses. Once warps get reassigned by DWB, they will access the new L1 cache array associated with the new SMX. It is more likely than not that the new L1 cache array does not have the data required by these warps. Therefore, L1 cache miss rate should be higher than the original. In Table 4.2, the percentage of extra cache misses brought by DWB is shown, which is insignificant compared with the original cache misses, so is the extra AMAT. It is because in the current generation of GPUs, there are intra-warp and inter-warp memory coalescing techniques [Bakhoda et al. 2009] taking effect to merge cache misses, including the ones caused by warp reassignment. So the outstanding cache misses left to be seen do not increase too much.
Table 4.2. Percentage of Extra Misses and AMAT

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Extra L1 Misses</th>
<th>Extra L2 Misses</th>
<th>AMAT Ori. (cycles)</th>
<th>AMAT with DWB (cycles)</th>
<th>Extra AMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>1.09%</td>
<td>0.55%</td>
<td>12.25</td>
<td>12.63</td>
<td>3.06%</td>
</tr>
<tr>
<td>RAY</td>
<td>0.28%</td>
<td>0.80%</td>
<td>7.06</td>
<td>7.28</td>
<td>3.14%</td>
</tr>
<tr>
<td>CP</td>
<td>0.57%</td>
<td>7.42%</td>
<td>2.06</td>
<td>2.13</td>
<td>3.27%</td>
</tr>
<tr>
<td>STO</td>
<td>0.47%</td>
<td>0.50%</td>
<td>8.94</td>
<td>9.23</td>
<td>3.22%</td>
</tr>
<tr>
<td>LPS</td>
<td>0.53%</td>
<td>13.67%</td>
<td>8.79</td>
<td>8.93</td>
<td>1.62%</td>
</tr>
<tr>
<td>MUM</td>
<td>3.72%</td>
<td>7.27%</td>
<td>5.57</td>
<td>5.82</td>
<td>4.49%</td>
</tr>
<tr>
<td>HIS</td>
<td>2.49%</td>
<td>0.72%</td>
<td>11.70</td>
<td>12.06</td>
<td>3.04%</td>
</tr>
<tr>
<td>MM</td>
<td>0.00%</td>
<td>3.04%</td>
<td>4.89</td>
<td>5.07</td>
<td>3.55%</td>
</tr>
<tr>
<td>SP</td>
<td>0.16%</td>
<td>0.51%</td>
<td>14.46</td>
<td>14.71</td>
<td>1.72%</td>
</tr>
<tr>
<td>ST</td>
<td>0.15%</td>
<td>0.09%</td>
<td>10.42</td>
<td>10.67</td>
<td>2.40%</td>
</tr>
<tr>
<td>MT</td>
<td>35.82%</td>
<td>15.55%</td>
<td>5.92</td>
<td>6.11</td>
<td>3.11%</td>
</tr>
<tr>
<td>NN</td>
<td>7.36%</td>
<td>48.07%</td>
<td>8.31</td>
<td>8.99</td>
<td>8.14%</td>
</tr>
<tr>
<td>CS</td>
<td>0.30%</td>
<td>0.00%</td>
<td>11.98</td>
<td>12.13</td>
<td>1.22%</td>
</tr>
<tr>
<td>TRAN</td>
<td>1.77%</td>
<td>0.27%</td>
<td>4.54</td>
<td>4.65</td>
<td>2.35%</td>
</tr>
<tr>
<td>BFS</td>
<td>3.94%</td>
<td>25.34%</td>
<td>4.18</td>
<td>4.42</td>
<td>5.73%</td>
</tr>
<tr>
<td>FWT</td>
<td>0.71%</td>
<td>0.10%</td>
<td>10.22</td>
<td>10.53</td>
<td>2.99%</td>
</tr>
<tr>
<td>ID</td>
<td>0.45%</td>
<td>1.18%</td>
<td>6.61</td>
<td>6.82</td>
<td>3.27%</td>
</tr>
<tr>
<td>JPEG</td>
<td>0.41%</td>
<td>6.72%</td>
<td>8.69</td>
<td>8.94</td>
<td>2.90%</td>
</tr>
<tr>
<td>Mean Value</td>
<td>3.17%</td>
<td>6.94%</td>
<td></td>
<td></td>
<td>3.29%</td>
</tr>
</tbody>
</table>

*This is the data of outstanding misses after memory coalescing.*

4.4 Conclusion

In this chapter, we propose a load balancing technique to improve the performance. It reassigns some of the unfinished warps from original SMXs to other near completion SMXs, where they may get processed instead of waiting idly. When the algorithm is activated, it incurs a little more dual-port SRAM accesses as a tradeoff. From the test results, the time saving gained overwhelms the delays caused by these extra accesses. A microarchitecture design based on shared structures is proposed, which includes solutions for preventing deadlock scenarios and avoiding potential errors during the execution.
that involves synchronization and shared variable access. A possible improvement to be made for DWB is to have GPGPU be self-adaptive on setting the parameters $T_{\text{complete}}$ and $\alpha$. In this work, simply for demonstrating the technique, the two parameters are selected by based on statistical tests on benchmarks, which is time-consuming. A better solution would be to let the GPU set up these parameters based on its specifications, and adjust them dynamically based on the speed of warps entering and exiting the scheduling queue in each SMX.
CHAPTER 5: DYNAMIC FREQUENCY SCALING ON GPUS

In this chapter, we propose two dynamic frequency scaling (DFS) techniques for general-purpose computing in GPUs (GPGPU). PIDDFS technique is motivated by the significance of the pipeline stalls during GPGPU execution, in which frequency can be reduced to save power without incurring performance loss. A feedback controlling algorithm, Proportional-Integral-Derivative (PID), is used to regulate the frequency of the processing units and memory channels based on the occupancy of the memory buffering queues. LPDFS technique targets on maximizing the average throughput of all parallel processors under the dynamic power constraints. We formalize this target as a linear programming problem and solve it on the runtime, then output the adjusted frequency for each processor. According to the results of the experiments, both techniques ensure energy savings as well as the performance improvement at the same time. Implemented on CPU host threads, both techniques involve no overhead in GPU execution. A prototype implementation of our work and the simulation are accomplished in a cycle-level GPU simulator, GPGPU-Sim. Data of power consumption are provided by the power modelling tool, GPUWattch. The simulation results from CUDA benchmarks show that PIDDFS achieves more than 22% power savings with a 4% improvement in performance and LPDFS saves 11% power consumption with 9% performance improvement, compared with the unmanaged system.
5.1 PIDDFS - Dynamic Frequency Scaling Using PID Controlling Theory

The PIDDFS technique we propose carries out DFS operations via two PID controllers that are pre-tuned to adapt to the GPGPU execution pattern. The SMX frequency controller regulates the frequency of each SMX based on the occupancy of its MSHR, while the memory channel frequency controller regulates the frequency of each memory channel based on the occupancy of the ICNT-Mem queue and Mem-ICNT queue on that channel. The straightforward purpose of PIDDFS is to maintain a proper level of utilization of these memory queues via maintaining the frequencies of SMXs and memory channels. In the meantime, power saving is achieved when lower frequencies are applied towards over-occupation of the queues, and performance improvement is achieved when higher frequencies are dispatched to make the queues fully utilized.

5.1.1 Motivation

Unlike the traditional graphic applications that mostly sequentially read the caches and memory for contiguous texture data, general-purpose applications tend to have a high rate of random memory access (both read and write), since the algorithm design is up to the programmers and can be in a pattern way different than graphic applications. Because of this, cache misses in general-purpose applications are more significant than in graphic ones. Therefore, pipeline stalls due to resolving cache misses are commonly seen, during which the power consumption needs to be saved.

To motivate our work on PIDDFS, we gather the data of stall cycles in MSHRs, ICNT-Mem and Mem-ICNT queues over total execution cycles for some general-purpose applications in Figure 5.1. The result is from simulation carried on in GPGPU-Sim. From the Figure 5.1, we can notice that the stalled
cycles in those memory buffering queues are significant. On average, the MSHR stalls take up around 25% of the total execution time. Stalls happening in ICNT-Mem and Mem-ICNT queues are almost 9% and 5%, respectively. If an MSHR is filled up by memory requests and stalls, the pipeline execution of the SMX that owns the MSHR will also stall until a vacant entry of the MSHR is available. Similarly, when an ICNT-Mem or Mem-ICNT queue congests and stalls, the corresponding memory channel will stall until the congestion is resolved. Both power and performance are lost during the stalled periods.

![Graph showing the ratio of stalled cycles over total execution cycles in MSHRs, ICNT-Mem and Mem-ICNT queues.]

**Figure 5.1.** Ratio of stalled cycles over total execution cycles in MSHRs, ICNT-Mem and Mem-ICNT queues.

PIDDFS technique adjusts the frequencies of each SMX and memory channel in order to avoid such congestions from happening. If the occupancy is higher than pre-set level in a memory buffering queue, PIDDFS will bring down the frequency of the module that fills in the queue, preserving the power consumption; or increase the frequency of the module that drains the queue, which is in favor of performance. By such frequency regulation, queue congestions will be reduced. Meanwhile, power-performance efficiency of the GPU will be improved.
5.1.2 PIDDFS Methodology

We implement two PID controllers in a CPU host thread and use it respectively to regulate the clock frequency for each SMX and each memory channel. Figure 5.2 demonstrates the structure of the controlling mechanism in outline. As discussed in the background section, the thread block that resides in each SMX is highly independent of each other. Each SMX can have a unique clock frequency and it will not affect the execution on the others. Here we use one PID controller to regulate the clock frequencies of the SMXs. During the active period, the SMX frequency controller processes the SMXs one by one it acquiring the occupancy in the MSHR of the current SMX as the measured process value, calculating the error and the output frequency based on the pre-tuned PID equation. Similarly, the memory channel frequency controller also works on each memory channel one after another, using the occupancy of the ICNT-Mem queue and the Mem-ICNT queue as its input. The aim of both SMX frequency controller and memory channel controller is to keep the occupancy in the memory buffering queues on a preset level. Flowcharts explaining how these two controllers work are shown in Figure 5.3.

**Figure 5.2.** Sketch of the mechanism of PIDDFS.
Figure 5.3. Flowcharts of the frequency controlling. (a) SMX frequency controlling. (b) Memory channel frequency controlling.

5.1.2.1 SMX Frequency Controller

The SMX frequency controller is activated every a period of time. During the active period, it processes each SMX one after another. When the number of memory requests in the MSHR of the current SMX is more than the preset reference value, the controller calculates and outputs a lower frequency for the SMX based on how much the error is, saving the power consumption while giving the ICNT more time to handle the memory requests waiting in the MSHR. When the number of memory requests is smaller than the preset reference value, the controller increases the frequency of the current SMX, which accelerates the computation of SMX and improves the throughput of it. Potentially, it may also generate more memory requests to occupy the vacant slots of the MSHR, hence the reasonable utilization of the space resource of the MSHRs.
5.1.2.2 Memory Channel Frequency Controller

There are two major components in a memory channel: L2 cache and DRAM. Both of them work under the same clock domain. Similar as the SMXs, each memory channel can have its own clock frequency and the PID controller updates the clock frequency of each memory channel one by one. The input that the controller takes from the current memory channel is the sum of the number of memory requests in ICNT-Mem queue and the number of the vacant slots in the Mem-ICNT queue. When the sum exceeds the preset reference value, the controller increases the frequency to drain the ICNT-Mem queue faster. When the sum is smaller than the reference value, the frequency will be lowered for power saving and congestion prevention in Mem-ICNT queue as well.

5.1.3 Implementation of PIDDFS

The PID controllers are implemented in the form of CPU host thread. The frequency regulation operation is non-blocking, introducing no overhead on GPU performance perspective. The only tradeoff involved is extra computing resource in host CPU, and extra power for frequency adjustment in frequency regulation modules of the GPU. In this section, we present a prototype implementation of PIDDFS. Then the choice of the activation period and the tuning of the PID controllers are discussed.

5.1.3.1 Prototype of Code Implementation

A simplified block diagram of the code implementation is shown in Figure 5.4. Each PID controller uses a structure to store its status and parameters. These structures are initialized in function \textit{Init\_PID()} at the beginning of GPU execution. The \textit{SMX\_fctrl()} and \textit{MC\_fctrl()} functions are in charge
of the main calculation using the parameters from the corresponding structures. We put a timer for each controller to invoke it at the end of every invocation period. When a controller is activated, it reads the system process value (which is the queue occupancy), then outputs the calculated frequency. In our simulation, the reference occupancy value for SMX frequency controller is set to be 50% capacity of the MSHR, and the reference value for memory channel controller is set to be 50% of the total entries of a ICNT-Mem and a Mem-ICNT queue.

![Figure 5.4. Block diagram of PIDDFS code implementation.](image)

### 5.1.3.2 Fixing the Activation Period

We also test run all our benchmarks with different activation time intervals in order to decide the optimal activation time for each PID controller. The total number of congested cycles in the MSHRs, ICNT-Mem and Mem-ICNT queues is plotted in Figure 5.5. We can see from two subfigures that: when the activation frequency is low (time interval between two activations is long), the number of congestion cycles tends to be large. This is because the controllers are not sensitive enough to maintain the occupancy of the queues at a proper level in order to avoid the congestion. On the other hand, when the
Figure 5.5. Average congested cycles in MSHRs and memory channels while using different PIDDFS activation frequencies.

Activation frequency is high (the time interval is short), the number of congestion cycles is small and tends to be a constant value when the time interval is short enough. Theoretically, the shorter the interval is, the more sensitive our technique is. But in reality, the activation interval cannot be minimized beyond the time required for calculations in the controllers. When the activation interval is shortened close to the calculation time required by the controllers, the congestion-reducing effect tends to become invariant, as shown in Figure 5.5. Another issue reflected in Figure 5.5 is that PIDDFS cannot eliminate the queue congestions completely. It is because our controllers can only regulate the clock frequency within a range and cannot fatherly overclock the controlled modules when the frequency red line for overheating issue is reached. Limited by the factors above, as well as by the activation interval length, PIDDFS can only partially reduce the queue congestions. The turning-point activation frequencies in
Figure 5.5: 3.5 Mhz (200 times of the SMX default clock period) and 924 Khz (1000 times of the memory channel clock period) are proper enough to be used to invoke the SMX and memory channel controllers respectively. We stick to these interval parameters in the rest of our simulation.

5.1.3.3 Tuning the PID Parameters

The properness of $P$, $I$ and $D$ terms of a PID controller is crucial in maintaining the stability of the system. Furthermore, the stabilizing effect must appear and it has to appear within an acceptable time limit. Finding the terms for the PID controllers can be a challenging task. To calculate the needed PID parameters, the best way is to find the PID parameters from a mathematical model of the system. But a detailed mathematical description of the occupancy of the memory request buffering queues is unavailable. Therefore, experimental tuning of the PID parameters has to be performed. We use Ziegler-Nichols, a well-known online tuning method, to respectively acquire the PID parameters of the two PID controllers in PIDDFS technique. First, we set the $I$ and $D$ terms to zero, increasing the $P$ term until the buffering queue occupancy oscillates in a sustained and stable pattern. Then the current $P$ term, as known as the critical gain $K_c$, and the oscillation period $P_c$ is recorded and according to the Ziegler-Nichols tuning table, the $K_p$, $K_i$ and $K_d$ values adjusted to $0.6K_c$, $2K_pP_c$ and $0.125K_pP_c$, respectively.

5.2 LPDFS - Dynamic Frequency Scaling Using Linear Programming

LPDFS technique we propose in this section aims at maximizing the average throughput of SMXs under some power constraints given by the user, including the overall dynamic power threshold.
of all SMXs and the dynamic power limit of each SMX. This maximization target is formalized as a linear programming problem in LPDFS, which is also designed to work out the optimal solution - the frequency for each SMX. By applying a direct limit on the dynamic power of SMXs, LPDFS can also reduce the total power consumed in the GPU.

### 5.2.1 Motivation

Power management techniques for heterogeneous computing systems have been widely explored. Although a lot of techniques are effective to bring down the power consumption, to set an in-advance power budget before execution and then save the power towards it is a feature that is commonly missing in these techniques. The significance of this feature is revealed when it comes to the battery-powered systems. These systems require each module, including the processor, to work strictly under a clearly-defined power budget so that the total power budget determined by the battery life can be met. Therefore, an effective power-aware technique should not be able to only save power, but also to quantify the amount of power to be saved and enforce it during the execution.

Another issue that many power-aware techniques ignore is the asymmetry at the architecture level of parallel computing platforms caused by process variation. Due to the large scale of the integrated circuit, there is some divergence of the process parameters from their nominal specifications. According to [BorkarDAC03], process variation has great influence on power consumption. Therefore, it is not accurate to treat the power characteristics of each parallel processing unit the same, as most of the power-aware techniques do.
LPDFS technique is designed to have the total dynamic power budget as an input, so that the power saving goal is directly given by the users as one of the constraint in the linear programming problem before the GPGPU execution. Therefore, the fulfillment of the power saving goal in LPDFS is ensured, unlike other power-aware techniques, in which the power saving amount can only be estimated before the execution. Meanwhile, in consideration of the process variation issue, we model each processing unit in the GPU separately, adjusting the parameters of their characteristics dynamically.

5.2.2 Methodology of LPDFS

Linear Programming (LP) is a mathematical technique to find the best possible solution (such as maximum profit or lowest cost) for allocating resources under some requirements that are mathematically modeled in linear inequalities. The LP problem looking for maximum solution can be expressed in the following form: for $N$ independent variables $x_1, x_2, \ldots, x_N$ ($x_1 \geq 0, x_2 \geq 0, \ldots, x_N \geq 0$), maximize the objective function:

$$g = c_1 x_1 + c_2 x_2 + \ldots + c_N x_N$$

which is subject to $N$ primary constraints:

$$\vec{a}_1 x_1 + \vec{a}_2 x_2 + \ldots + \vec{a}_N x_N \leq \vec{b}$$

where scalars $c_{1,\ldots,N}$, $N$-by-1 vectors $\vec{a}_{1,\ldots,N}$ and $\vec{b}$ are problem-specific parameters.

The GPU performance optimization target can be defined as follows: given a set of $N$ SMXs $SMX_{1,\ldots,N}$, find the best selection of frequencies $(f_1, \ldots, f_N)$ for SMXs $SMX_{1,\ldots,N}$ that can maximize the overall throughput ($TP$) subject to the constraints on power perspective:

- The overall dynamic power for SMXs is less than a total power budget $P_{\text{dyn\_MAX}}$. 

73
The dynamic power of each SMX is less than a threshold \( P_{\text{SMX,MAX}} \) designated specifically for it.

Then we exploit the feasibility of formalizing the optimization problem into a linear programming problem. Mathematically, the objective function and the constraint inequalities have to be linear if an optimization problem is a linear programming problem. Our objective function is the average throughput of all SMXs:

\[
TP = \frac{tp_1 + tp_2 + \ldots + tp_N}{N}
\]

where \( tp_i \) is the throughput of SMX\(_i\), and \( N \) being the total number of SMXs. As is known to all, the throughput can be expressed as:

\[
tp_i = f_i \cdot ipc_i
\]

where \( f_i \) is the frequency of SMX\(_i\) and \( ipc_i \) is the instructions per cycle (IPC) of the thread block that is running on SMX\(_i\). Ideally, IPC is independent of frequency. In reality, however, it starts to go down as the frequency \( f \) goes up when \( f \) is higher than a certain level. This phenomenon is mainly caused by the inevitable latencies required by certain functional units, such as ALU, multiplier and off-chip memory. When \( f \) is high enough, stalls will be issued into the execution pipelines in order to supply the time needed by some certain units to complete their functions. The more stalled cycles there are, the lower the IPC of that SMX will be.

Therefore, when the stalled cycles are not significant, we can neglect the dependence between \( f \) and \( ipc \) so that the objective function can be expressed in a linear format:
\[
TP = \frac{ipc_1}{N} f_1 + \frac{ipc_2}{N} f_2 + \ldots + \frac{ipc_N}{N} f_N
\]

where \( f_1, \ldots, f_N \) are the set of frequencies we are trying to acquire via solving the optimization problem.

The first constraint inequality is that the total dynamic power consumed by all SMXs should be less than power budget \( P_{\text{dyn,MAX}} \). Same as in CPU, the dynamic power consumed in SMX\( i \) can be approximated as:

\[
p_i = a_i \cdot c_i \cdot v_i^2 \cdot f_i
\]

where \( a_i \) is the activity factor (the fraction of the circuit that is switching). \( c_i \) is the capacitance of SMX\( i \), \( v_i \) is the voltage supplied to SMX\( i \). If there are no voltage regulation techniques running in parallel with the LPDFS method, the voltage of each SMX \( v_i \) can be treated as constant. In practice, we assume \( a_i \cdot c_i \) to be invariant when the functional units on the execution pipelines are mostly active without major shutdown caused by long idleness. Therefore the \( P_{\text{dyn,MAX}} \) constraint can be written as follows:

\[
\left(a_1c_1v_1^2\right) \cdot f_1 + \left(a_2c_2v_2^2\right) \cdot f_2 + \ldots + \left(a_Nc_Nv_N^2\right) \cdot f_N \leq P_{\text{dyn,MAX}}
\]

The second constraint specifies that the dynamic power consumption of each SMX should be less than a threshold designated specifically for it:

\[
\left(a_ic_i^2\right) \cdot f_i \leq P_{\text{SMX,MAX}} \cdot i, \forall i \in 1, \ldots, N
\]

Now that the objective function and all the constraints are in linear format, we can formulate our problem as a linear programming problem. Factors \( a_1c_1v_1^2, a_2c_2v_2^2, \ldots, a_Nc_Nv_N^2 \) that are related to GPU characteristics can be acquired by linear regression, with the overall dynamic power of SMXs as the explanatory variable and the frequencies of each SMX as the dependent variables. Then, the inputs of
the LP problem are $\frac{ipc_1}{N}, \frac{ipc_2}{N}, \ldots, \frac{ipc_N}{N}$, (dynamically acquired during the GPU execution), as well as the dynamic power constraint for all SMXs $P_{\text{dyn}, \text{MAX}}$ and the power threshold $P_{\text{SMX,MAX}}$ for each SMX (provided by the users before GPU execution).

5.2.3 Implementation of LPDFS

Similar as the PIDDFS, the LPDFS is also implemented as a CPU host thread. We program our customized LP solver that solves the optimization problem using Simplex method [Dantzig et al. 1955], a fast-computing LP solving method that is broadly used in practice. Simplex algorithm involves a variable number of steps. The computation time of each step depends on the size of the problem, specifically in our case, the number of SMXs and the number of the constraints. The overhead of computation in LPDFS is larger than in PIDDFS because it involves more complicated calculations. However, by implementing LPDFS in the form of CPU host thread, this overhead will not affect the GPU execution. The performance counters for IPC of each SMX introduce some tradeoff in hardware perspective. But it is quite small and manageable.

The parameters needed in the formation of our LP problem are: the IPC of each SMX, the factor $a \cdot c \cdot v^2$ for each SMX, the total dynamic power constraint $P_{\text{dyn,MAX}}$ and the maximum allowable dynamic power of each SMX $P_{\text{SMX,MAX}}$. In Table 5.1, we briefly sum up how we acquire these parameters. The factor $a \cdot c \cdot v^2$ of each SMX can be calculated using the equation

$$P_{\text{dyn,SMXs}} = \sum_{i=1}^{N} a_i \cdot c_i \cdot v_i^2 \cdot f_i$$

with the overall dynamic power $P_{\text{dyn,SMXs}}$, and clock frequency $f_i$ of the moment acquired. The values of $P_{\text{SMX,MAX}}$ and $P_{\text{dyn,MAX}}$ can be the hardware thermal thresholds of the
Table 5.1. Parameters of LPDFS and Acquisition Means

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Acquisition Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \cdot c \cdot v^2$ of each SMX</td>
<td>Calculated via $P_{dyn, SMXs} = \sum_{i=1}^{N} a_i \cdot c_i \cdot v_i^2 \cdot f_i$ using linear regression with $P_{dyn, SMXs}$ and $f_i$ acquired from test runs of benchmarks. Then used as constant during simulation.</td>
</tr>
<tr>
<td>$P_{SMX, MAX}$ of each SMX</td>
<td>Up to the users, with upper bound determined by the overheating issue.</td>
</tr>
<tr>
<td>$P_{dyn, MAX}$</td>
<td>Up to the users, with upper bound determined by the overheating issue.</td>
</tr>
<tr>
<td>$ipc$ of each SMX</td>
<td>Obtained dynamically by performance counters.</td>
</tr>
</tbody>
</table>

GPU, or up to the users. In our simulations, we choose to use 90% of the maximum dynamic power of SMXs bounded by overheating issue, as our $P_{dyn, MAX}$. As for the $P_{SMX, MAX}$, we set the $P_{SMX, MAX}$ for all the SMXs to be $P_{dyn, MAX} / (3 \times N)$. That is to say, only one third of the SMXs are allowed to run at the maximum frequency at the same time. The IPC of each SMX, though assumed to be independent of the frequency and voltage, is updated dynamically by designated performance counters.

![Figure 5.6](image)

**Figure 5.6.** Average total execution cycles when using different LPDFS activation frequencies.
The activation period of LPDFS mechanism can impact the average SMX throughput $TP$ that we are trying to maximize. Specifically, if we use an interval that is too long between invocations, the parameters of each SMX may have varied, which affect the accuracy of the calculation. In Figure 5.6, we apply different activation frequencies of LPDFS and plot the average value of total execution times of all our benchmarks. The trend of the curve can be interpreted as: when the invocation interval is long, the total execution time is long and the variation between the plotted values is significant because the frequencies calculated in the previous invocation of LPDFS are not suitable for the system that is altered during the interval. As the interval is getting short, the execution time tends to remain invariable. The turning point of the activation frequency: 350 Khz (when the interval length is approximately 2000 times of the SMX default clock period), is used in our simulation.

5.3 Simulation and Evaluation

We simulate the GPU execution using GPGPU-Sim [Bakhoda et al.2009] simulator configured for the architecture of NVIDIA GTX 480, same as we did in previous work. For the range of frequency, we apply 150% of normal frequency as the frequency upper bound and 0 as the lower bound, in which case the execution completely stalls. The power consumption of each application is measured by GPUWattch [Leng et al. 2013]. In our simulations, eighteen CUDA programs are used as our benchmarks. In Figure 5.7, L1 cache miss rate, L2 cache miss rate and the ratio of memory time over total execution time are charted.
5.3.1 Evaluation of Memory Stall Reduction via PIDDFS

5.3.1.1 Result

The PIDDFS technique directly maintains the occupancy of the MSHRs, ICNT-Mem queues and Mem-ICNT queues at a preset level, on which the congestion or empty scenarios will be minimum, by the means of frequency regulation. Performance improvement and power saving are consequent results of the frequency increasing and lowering respectively. To analyse the direct outcome of PIDDFS technique, we gather the data that indicate the reduction of congestion in MSHRs, ICNT-Mem queues and Mem-ICNT queues in Figure 5.8. The overall reduction of stalled cycles due to queue congestion is significant. Specifically, there are 11 out of 18 benchmarks have their MSHR congestion cycles reduced over 40%, 13 benchmarks have both ICNT-Mem congestion and Mem-ICNT congestion reduced over 40%. It indicates that PIDDFS technique has a general and notable effect on reducing the congestions in these memory buffering queues. Benchmark BS has no value in Figure 5.8 because it originally has no congestion in any of the queues.
5.3.1.2 Analysis

Comparing the data in Figure 5.8 within each benchmark, the reduced portion of congestion cycles in MSHRs is smaller than that in ICNT-Mem and Mem-ICNT queues. This phenomenon is caused by the difference in functionalities of MSHR and the other two queues. In MSHRs, memory requests generated from concurrently-issued threads are buffered and checked for the purpose of coalescing into several accesses to small, contiguous memory regions. The possibility of such coalescing is high, especially for the memory requests generated by the threads within the same warp. Due to the delay for such coalescing check, the occupancy in MSHRs often becomes very intensive. Congestions are more commonly seen and harder to get rid of than in ICNT-Mem and Mem-ICNT queues, which serve the buffering purpose only.

We see that the effect that the PIDDFS technique has upon each benchmark also varies. Benchmarks like CS, FWT and TRAN have their MSHR stalls reduced over 60\%, and both their ICNT-Mem stalls and Mem-ICNT stalls are even reduced over 80\%. However, for HIS, LIB, MM and WP, the reduced portion of stalls is only around 20\% or less. A hypothesis is proposed that the
difference in the data of reduced congestion portion among benchmarks may be related to their memory accessing characteristics. To thoroughly analyse the correlation between the congestion-reducing effect of PIDDFS technique and the intensity of memory accessing for each benchmark, we rearrange the data in Figure 5.8 by the order of the memory time versus total execution time in Figure 5.7, and plot it in Figure 5.9. The trends of the curves in Figure 5.9 can be generally interpreted as: the reduction of the stalls in congested queues is more significant for those benchmarks that are less memory-bound. When memory accesses are small in quantity and scattered sparsely through the execution period, PIDDFS is effective in eliminating the queue congestions. However, when memory accesses are large in quantity, or intensively burst within a short period of time, the PIDDFS mechanism is not able to prevent the congestion from happening due to the length of the invoking period and the frequency adjusting capability.

![Figure 5.9](image-url)  
**Figure 5.9.** Correlation between the percentage of reduced stalled cycles and the ratio of memory time over total execution time.
5.3.2 Evaluation of Performance Improvement

We respectively simulate PIDDFS and LPDFS techniques over the CUDA benchmarks. In this section, the result of performance improvement is shown in Table 5.2. The GPU throughput for each benchmark is measured in MIPS. An analysis is made towards the different performance improving effects of PIDDFS and LPDFS among the benchmarks.

Table 5.2. Performance Improvement of PIDDFS and LPDFS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Total Throughput (MIPS)</th>
<th>PIDDFS</th>
<th>LPDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total Throughput (MIPS)</td>
<td>Improvement (%)</td>
</tr>
<tr>
<td>AES</td>
<td>425315</td>
<td>452011</td>
<td>6.28%</td>
</tr>
<tr>
<td>BFS</td>
<td>20220</td>
<td>21225</td>
<td>4.97%</td>
</tr>
<tr>
<td>BS</td>
<td>14598</td>
<td>15502</td>
<td>6.19%</td>
</tr>
<tr>
<td>CP</td>
<td>329146</td>
<td>345126</td>
<td>4.85%</td>
</tr>
<tr>
<td>CS</td>
<td>521012</td>
<td>552669</td>
<td>6.08%</td>
</tr>
<tr>
<td>FWT</td>
<td>284163</td>
<td>302119</td>
<td>6.32%</td>
</tr>
<tr>
<td>HIS</td>
<td>423189</td>
<td>433669</td>
<td>2.48%</td>
</tr>
<tr>
<td>LIB</td>
<td>127921</td>
<td>131577</td>
<td>2.86%</td>
</tr>
<tr>
<td>LPS</td>
<td>308553</td>
<td>320629</td>
<td>3.91%</td>
</tr>
<tr>
<td>MM</td>
<td>208299</td>
<td>214313</td>
<td>2.89%</td>
</tr>
<tr>
<td>NN</td>
<td>31658</td>
<td>32723</td>
<td>3.37%</td>
</tr>
<tr>
<td>NQU</td>
<td>23712</td>
<td>25053</td>
<td>5.65%</td>
</tr>
<tr>
<td>RAY</td>
<td>328974</td>
<td>343447</td>
<td>4.40%</td>
</tr>
<tr>
<td>SP</td>
<td>243853</td>
<td>255269</td>
<td>4.68%</td>
</tr>
<tr>
<td>ST</td>
<td>313543</td>
<td>326926</td>
<td>4.27%</td>
</tr>
<tr>
<td>STO</td>
<td>230736</td>
<td>241503</td>
<td>4.67%</td>
</tr>
<tr>
<td>TRAN</td>
<td>60932</td>
<td>64626</td>
<td>6.06%</td>
</tr>
<tr>
<td>WP</td>
<td>50681</td>
<td>51762</td>
<td>2.13%</td>
</tr>
<tr>
<td>Mean Value</td>
<td></td>
<td></td>
<td>4.56%</td>
</tr>
</tbody>
</table>
5.3.2.1 Result

In Table 5.2, we can see that LPDFS technique improves the throughput of 18 benchmarks by 9.3%, with a maximum value 16.5%. PIDDFS technique improves the throughput by approximately 4.6%, which is not as significant as LPDFS. This is because LPDFS exactly targets on the optimal overall throughput within a power expense, however, what PIDDFS directly targets on is getting rid of the queue congestions, which indirectly helps in improving the throughput when some of the SMXs are accelerated. Moreover, the acceleration of SMXs via PIDDFS may not lead to overall throughput improvement when the accelerated SMXs are not on the critical path of the GPGPU execution. Therefore, the effect of throughput improvement by PIDDFS method is not as obvious as LPDFS method.

![Graph showing Memory vs Total Execution Time and PIDDFS/ LPDFS Performance Improvement]

**Figure 5.10.** Correlation between the performance improvement (using y-axis on the left) and the ratio of memory time over total execution time (using y-axis on the right).

5.3.2.2 Analysis

To analyse the different performance improving effects that LPDFS and PIDDFS methods have upon each benchmark, we plot the data from Table 5.2 along with the percentage of memory time over total execution time in Figure 5.10. Performance improvement of both LPDFS and PIDDFS decreases
nearly monotonically when the ratio of memory time over total execution time increases, from which we can conclude that both LPDFS and PIDDFS methods are not in favour of the memory-bound benchmarks in performance improvement perspective. It is mainly because the intensity of memory requests can add negative impact upon LPDFS and PIDDFS. LPDFS only regulates SMX clock frequencies. However, for those memory-intensive applications, increasing the SMX clock frequencies may not improve the overall throughput very much. As for the PIDDFS, it deals with the memory request intensity in the buffering queues, but the frequency decreasing operations for power saving are dominant, instead of frequency increasing operations for the throughput improvement. Moreover, the load of memory requests can exceed the capability of PIDDFS frequency controllers, with queue congestions partially remained. In these cases, the more memory requests there are, the less the performance can be improved by PIDDFS.

5.3.3 Evaluation of Power Saving

In this section, the result of power saving is shown and discussed. In Table 5.3, the power data of each benchmark are averaged throughout the GPU execution time.

5.3.3.1 Result

The result of power saving for both PIDDFS and LPDFS is shown in Table 5.3. The power data of each benchmark are acquired from the GPU power model in GPUWattch. In Table 5.3, we can see that PIDDFS technique saves approximately 23% power on average for all the benchmarks. LPDFS technique has 11.4% of overall power saving on average, as we set the $P_{dyn,MAX}$ constraint to be 90% of the maximum dynamic power.
Table 5.3. Power Saving of PIDDFS and LPDFS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Average Power (W)</th>
<th>PIDDFS</th>
<th>LPDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Power (W)</td>
<td>Saving (%)</td>
<td>Average Power (W)</td>
</tr>
<tr>
<td>AES</td>
<td>83.17</td>
<td>65.20</td>
<td>21.61</td>
</tr>
<tr>
<td>BFS</td>
<td>41.68</td>
<td>32.43</td>
<td>22.18</td>
</tr>
<tr>
<td>BS</td>
<td>26.12</td>
<td>20.83</td>
<td>20.23</td>
</tr>
<tr>
<td>CP</td>
<td>133.96</td>
<td>104.00</td>
<td>22.37</td>
</tr>
<tr>
<td>CS</td>
<td>120.60</td>
<td>96.01</td>
<td>20.39</td>
</tr>
<tr>
<td>FWT</td>
<td>103.57</td>
<td>82.85</td>
<td>20.00</td>
</tr>
<tr>
<td>HIS</td>
<td>78.36</td>
<td>57.71</td>
<td>26.35</td>
</tr>
<tr>
<td>LIB</td>
<td>67.57</td>
<td>50.20</td>
<td>25.70</td>
</tr>
<tr>
<td>LPS</td>
<td>100.94</td>
<td>76.79</td>
<td>23.92</td>
</tr>
<tr>
<td>MM</td>
<td>45.65</td>
<td>33.93</td>
<td>25.66</td>
</tr>
<tr>
<td>NN</td>
<td>92.05</td>
<td>69.19</td>
<td>24.84</td>
</tr>
<tr>
<td>NQU</td>
<td>23.65</td>
<td>18.67</td>
<td>21.07</td>
</tr>
<tr>
<td>RAY</td>
<td>76.00</td>
<td>58.43</td>
<td>23.12</td>
</tr>
<tr>
<td>SP</td>
<td>84.86</td>
<td>65.64</td>
<td>22.65</td>
</tr>
<tr>
<td>ST</td>
<td>60.34</td>
<td>46.26</td>
<td>23.33</td>
</tr>
<tr>
<td>STO</td>
<td>32.74</td>
<td>25.32</td>
<td>22.68</td>
</tr>
<tr>
<td>TRAN</td>
<td>43.65</td>
<td>34.74</td>
<td>20.41</td>
</tr>
<tr>
<td>WP</td>
<td>35.91</td>
<td>26.24</td>
<td>26.94</td>
</tr>
<tr>
<td>Mean Value</td>
<td></td>
<td>22.97</td>
<td></td>
</tr>
</tbody>
</table>

5.3.3.2 Analysis

The relation between power saving percentage and memory access intensity is shown in Figure 5.11. The power saving via LPDFS is generally less for those memory bound benchmarks. The explanation for this phenomenon is that LPDFS only regulates the frequency for SMXs, not for the modules that handle memory requests. So the mechanism of LPDFS does not directly help in resolving the memory requests. Moreover, when the memory requests are intensive, stalls become significant in the pipeline execution, which may radically change the IPC of the SMXs. This will cause the inaccuracy
in the expression of the objective function of the LP problem, and affects the solution. PIDDFS technique, on the other hand, generally saves more power when the benchmark is more memory bound. We can see a steady growth on the percentage of power savings by PIDDFS in Figure 5.11 as the data point moves from the benchmark with lowest memory access intensity to the highest. It is because PIDDFS tends to lower down the frequency to deal with the bursts of memory requests. If the memory access intensity is high, the low frequency period certainly will be long and the power saved during that period will be more.

![Figure 5.11](image)

**Figure 5.11.** Correlation between the power saving (using y-axis on the left) and the ratio of memory time over total execution time (using y-axis on the right).

### 5.4 Conclusion

In this chapter, we propose two DFS techniques to optimize the power consumption and performance for GPGPU. PIDDFS technique aims at maintaining a proper level of occupancy in memory buffering queues in order to avoid execution stalls and save power. LPDFS maximizes the average throughput of the SMXs under dynamic power constraints. The methodologies of PIDDFS and LPDFS are described respectively. According to the benchmark simulation results, PIDDFS saves 22% of overall power consumption and improves the performance by 4% in the meantime. LPDFS yields approximately 11% power savings and 9% of performance improvement. Analysis of the results is made
in order to find out the correlation between memory access intensity of the applications and the effects
of the two techniques in power and performance aspects. In our future research, we plan to investigate
the coordination of the two techniques. We will explore the feasibility of combining the mechanisms of
the two techniques and having a comprehensive DFS module to regulate the frequency of each domain
of the GPU. The emphasis of the DFS regulations can be switched between power saving and
performance improvement.
CHAPTER 6:
CONCLUSIONS AND FUTURE DIRECTIONS

Performance and power optimization in heterogeneous computing system has become a challenging focus for the processor architecture and circuit designers. The research efforts shown in this dissertation represent a solid contribution to addressing the issue of improving performance and power consumption in GPU-accelerated computing system. Here we list some future directions to extend the works presented in this dissertation.

- **More fine-grained power gating in caches** – In the work described in Chapter 3, we propose a cache power gating technique that has with three working modes for caches in order to save the leakage power of L1 and L2 caches in the GPU. For future work, we want to extend our power gate controlling mechanism into a more fine-grained version. Currently, we are power gating the L1 or L2 by the unit of the whole array. If we can manage to bring the granularity of power gating technique down to each individual cache line within an L1 or L2 cache array, we will have opportunities to save more leakage power by only waking up one or a few cache lines that will be accessed in the next moment, while the rest of cache lines can still stay in power gated mode.

- **Be self-adaptive in workload balancing** - In the work described in Chapter 4, we introduce Dynamic Workload Balancing technique to improve the performance via reassigning some
of the unfinished warps from original SMXs to other near-completion SMXs, where they may get processed instead of waiting idly. A possible improvement to be made for Dynamic Workload Balancing is to have GPU be self-adaptive on setting the parameters $T_{\text{complete}}$ and $\alpha$. In this work, simply to demonstrate the technique, the two parameters are selected by based on statistical tests on benchmarks, which is time-consuming. A better solution would be to let the GPU set up these parameters based on its specifications, and adjust them dynamically based on the speed of warps entering and exiting the scheduling queue in each SMX.

- A comprehensive DFS mechanism – In Chapter 5, we propose two DFS techniques to optimize the power consumption and performance for GPGPU. PIDDFS technique aims at maintaining a proper level of occupancy in memory buffering queues in order to avoid execution stalls and save power. LPDFS maximizes the average throughput of the SMXs of the GPGPU under a dynamic power budget. In our future research, we plan to investigate the coordination of the two techniques. We will explore the feasibility of combining the mechanisms of the two techniques and having a comprehensive DFS module to regulate the frequency of each domain of the GPU. The emphasis of the DFS regulations can be switched by the users between power saving and performance improvement.
REFERENCES


NVIDIA Corporation 2007. NVIDIA CUDA programming guide (1.1 ed.). NVIDIA Corporation.


ABOUT THE AUTHOR

Yue Wang received both his Bachelor of Engineering degree in Computer Science and Bachelor of Engineering degree in Automation in 2009 from Shandong University of Science and Technology, Qingdao, China. In 2011, he received Master of Science degree in Computer Engineering from University of South Florida, Tampa, FL. He is currently working towards the Ph.D. degree in Computer Science and Engineering in University of South Florida. His research interests are in architecture level low-power design of microprocessors and graphic processing units, architecture level and instruction level power modeling. He is a student member of the IEEE and the IEEE Computer Society.