Intelligent Scheduling and Memory Management Techniques for Modern GPU Architectures

by

Shin-Ying Lee

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved July 2017 by the
Graduate Supervisory Committee:

Carole-Jean Wu, Chair
Chaitali Chakrabarti
Fengbo Ren
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

August 2017

©2017 Shin-Ying Lee

All Rights Reserved

ABSTRACT

With the massive multithreading execution feature, graphics processing units (GPUs) have been widely deployed to accelerate general-purpose parallel workloads (GPGPUs). However, using GPUs to accelerate computation does not always gain good performance improvement. This is mainly due to three inefficiencies in modern GPU and system architectures.

First, not all parallel threads have a uniform amount of workload to fully utilize GPU's computation ability, leading to a sub-optimal performance problem, called warp criticality. To mitigate the degree of warp criticality, I propose a Criticality-Aware Warp Acceleration mechanism, called CAWA. CAWA predicts and accelerates the critical warp execution by allocating larger execution time slices and additional cache resources to the critical warp. The evaluation result shows that with CAWA, GPUs can achieve an average of 1.23x speedup.

Second, the shared cache storage in GPUs is often insufficient to accommodate demands of the large number of concurrent threads. As a result, cache thrashing is commonly experienced in GPU's cache memories, particularly in the L1 data caches. To alleviate the cache contention and thrashing problem, I develop an instruction-aware Control Loop Based Adaptive Bypassing algorithm, called Ctrl-C. Ctrl-C learns the cache reuse behavior and bypasses a portion of memory requests with the help of feedback control loops. The evaluation result shows that Ctrl-C can effectively improve cache utilization in GPUs and achieve an average of 1.42x speedup for cache sensitive GPGPU workloads.

Finally, GPU workloads and the co-located processes running on the host chip multiprocessor (CMP) in a heterogeneous system setup can contend for memory resources in multiple levels, resulting in significant performance degradation. To maximize the system throughput and balance the performance degradation of all co-located

applications, I design a scalable performance degradation predictor specifically for heterogeneous systems, called HeteroPDP. HeteroPDP predicts the application execution time and schedules OpenCL workloads to run on different devices based on the optimization goal. The evaluation result shows HeteroPDP can improve the system fairness from 24% to 65% when an OpenCL application is co-located with other processes, and gain an additional 50% speedup compared with always offloading the OpenCL workload to GPUs.

In summary, this dissertation aims to provide insights for the future microarchitecture and system architecture designs by identifying, analyzing, and addressing three critical performance problems in modern GPUs.

ACKNOWLEDGMENTS

Exploring the darkest world to discover new opportunities, doing research is one of the toughest adventures. It is a lonely, challenging, stumbling, and endless journey. Yet, it is also the most exciting and joyful achievement when we see a beam of light in the deep darkness. I am truly glad I have had a chance to enjoy this great moment, watching a warm ray of light shining the ground.

I am grateful to my research advisor, Dr. Carole-Jean Wu. With her passion in research, Carole opened a door for me to this amazing research space. With her patience in teaching, Carole guided me overcoming all the challenges I had experienced. This thesis would not have been possible without her enthusiasm for mentoring students.

I would like to thank Dr. Chaitali Chakrabarti, Dr. Fengbo Ren, and Dr. Aviral Shrivastava for serving on my dissertation committee and helping me improve my research work.

I would like to thank my writing instructor, Gregory Fields, and all tutors from the ASU writing center for reviewing and polishing my research papers as well as my PhD dissertation.

I would like to thank my colleagues at AMD and Apple, Cyril de Chanterac, Dr. Jin Chen, Dr. Ying Chen, Michael Christman, Michael Chung, Dr. Anas Lasram, Dr. Timour Paltashev, Dhruv Saksena, Dr. Dana Schaa, Dr. Churayev Sergey, Dr. Stephen Somogyi, and Charles Tan, for broadening my vision in industry and inspiring me to find out new research ideas.

I would like to thank all my labmates and classmates in ASU, Akhil, Amrit, Benjamin, Chia-Wen, Davesh, Dhinakaran, Digant, Duo, Hsing-Min, Jhe-Yu, Jian, Jeevan, Ke, Moslem, Nishant, Shail, Vignesh, and Yooseong for assisting me in setting up my experiment infrastructure and reviewing my code.

I would like to thank Yen-Shao and Chung-Ying for preparing a geat apartment for me before I arrived in Arizona.

I would like to thank Chi-Han and Yu-Ying for generously providing me a cozy home in a foreign country.

Most importantly, I would like to thank for my parents and my old brother, Yu-Rey, for supporting, encouraging, and motivating me finishing this adventure.

Without all your kindly encouragement, unlimited support, valuable suggestions, as well as rigorous criticisms, I would never be able to arrive at a destination of such a long journey. After four and a half years, now, it is the time to share this wonderful moment with all of you in my life to sincerely express my best gratitude.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 INTRODUCTION 1

1.1 GPGPU Architecture and Computation 2

1.2 The Inefficiencies of Current GPGPU and Accelerator-rich System Designs 4

1.2.1 Execution Time Disparity and Warp Criticality 5

1.2.2 Resource Contention and Cache Thrashing 6

1.2.3 Data Movement Overheads and Memory Interference 7

1.3 Contributions 9

2 RELATED WORK 11

2.1 Warp Criticality 11

2.2 Cache Thrashing Problem and Cache Management 14

2.3 Memory Interference in Heterogeneous Systems 18

3 WARP CRITICALITY AND CRITICALITY-AWARE WARP SCHEDULING 23

3.1 The Baseline GPGPU Architecture and its Computation Model 23

3.2 GPU Latency Hiding Ability 25

3.2.1 Factors Stalling Warp Execution 25

3.2.2 Latency Attribution Algorithm 27

3.2.3 Characterization Methodology 31

3.2.4 Latency Characterization 33

3.3 Warp Criticality 35

3.4 Warp Scheduler Design Exploration for Critical Warp Acceleration . 39

3.4.1 CAWS Algorithms 39

3.4.2 CAWS Implementation 40

3.5 Evaluation and Analysis 42

3.6 Chapter Summary 44

4 COORDINATED CRITICALITY-AWARE WARP ACCELERATION . . . 48

4.1 Source of Execution Time Disparity 48

4.1.1 Workload Imbalance 49

4.1.2 Diverging Branch Behavior 51

4.1.3 Contention in the Memory Subsystem 52

4.1.4 Latency Introduced by the Warp Scheduler 54

4.2 Coordinated Criticality-Aware Warp Acceleration Design 55

4.2.1 Critical Warp Identification with Criticality Prediction Logic 56

4.2.2 greedy Criticality-Aware Warp Scheduler 59

4.2.3 Criticality-Aware Cache Prioritization 60

4.3 Evaluation and Analysis 65

4.3.1 Experimental Environment and Methodology 65

4.3.2 Performance Overview 66

4.3.3 Performance Analysis for CPL 69

4.3.4 Performance Analysis for gCAWS 70

4.3.5 Performance Analysis for CACP 72

4.4 Chapter Summary 75

5 INSTRUCTION-AWARE CONTROL LOOP BASED ADAPTIVE CACHE BYPASSING 76

5.1 GPU Cache Access Behavior Characterization 76
5.2 Control-Loop Based Adaptive Cache Bypassing 83
5.2.1 Design Overview of Ctrl-C 83
5.2.2 Cache Line Reuse Prediction and iReuse Table 84
5.2.3 Feedback Control Loop 85
5.2.4 The Ctrl-C Cache Bypassing Algorithm 87
5.3 Evaluation and Analysis 88
5.3.1 Experimental Environment and Methodology 88
5.3.2 Performance Improvement 90
5.3.3 MPKI and Interconnect Traffic Reduction 91
5.3.4 Fraction of Zero-reuse Lines 92
5.3.5 Hardware Implementation Overhead 93
5.4 Chapter Summary 93
6 PERFORMANCE CHARACTERIZATION AND PREDICTION FOR HETEROGENEOUS COMPUTER SYSTEMS WITH GPUS 97
6.1 Heterogeneous Systems and the OpenCL Framework 97
6.2 Methodology 100
6.2.1 Experiment Infrastructure and Configurations 100
6.2.2 Workload Construction 101
6.3 Motivation for an Intelligent Execution Target Scheduler 101
6.3.1 Performance Characterization 102
6.3.2 Optimal Execution Target in the Presence of Memory Interference 103
6.3.3 Performance Degradation with Different Co-location Scenarios 105

6.3.4 Performance Degradation with Different Scheduling Priorities 106
6.4 Performance Degradation Predictor for Heterogeneous Systems 108
6.4.1 The HeteroPDP Prediction Scheme Overview 109
6.4.2 OpenCL Kernel Execution Time Prediction for *alone* 110
6.4.3 OpenCL Kernel Execution Time Prediction for *co-located* . . . 111
6.4.4 Performance Model Training for OpenCL Kernels 111
6.4.5 Performance Degradation Prediction for Native CPU Applications 112
6.5 Evaluation and Analysis 113
6.5.1 Execution Time and Execution Target Prediction Accuracy . 113
6.5.2 Evaluation for System Performance 115
6.5.3 HeteroPDP with Varying Scheduling Priorities 116
6.5.4 HeteroPDP Scalability Analysis 118
6.6 Chapter Summary 120
7 CONCLUSIONS 126
REFERENCES 129
APPENDIX
A REGRESSION MODELS AND COEFFICIENTS FOR HETEROPDP . . 142

LIST OF TABLES

Table Page

3.1 GPGPU-sim Configurations for Latency Characterization 31

3.2 Benchmarks for GPGPU Latency Hiding Ability Characterization 46

3.3 The Speedup and Frequency of Criticality Inversion within a Threadblock for `BFS` 47

4.1 GPGPU-sim Simulation Configurations for CAWA 66

4.2 Benchmarks for CAWA Evaluation 67

5.1 GPGPU-sim Simulation Configurations for Ctrl-C 89

5.2 Default Configurations for the Ctrl-C Control Loop Design 90

5.3 Benchmarks for Ctrl-C Performance Evaluation 96

6.1 Memory Interference Infrastructure Setup and Configurations 122

6.2 CPU Workloads for the Characterization Studies and Design evaluation 123

6.3 OpenCL Workloads for the Characterization Studies and Design Evaluation 124

6.4 The OpenCL Kernel Features Used for Execution Time Prediction 125

A.1 Coefficients for Predicting OpenCL Kernel Execution Time *alone* on the Intel Core i7-3770 CMP 143

A.2 Coefficients for Predicting OpenCL Kernel Execution Time *alone* on the AMD FirePro S9150 GPU 144

A.3 Coefficients for Predicting OpenCL Kernel Execution Time *co-located* on Intel Core i7-3770 CMP 145

A.4 Coefficients for Predicting OpenCL Kernel Execution Time *co-located* on the AMD FirePro S9150 GPU 146

LIST OF FIGURES

Figure | Page

1.1 An Example of an Accelerator-rich Heterogeneous Computer System . . . 2
1.2 An Overview of the Modern GPGPU Microarchitecture 3
3.1 The Execution Order with the Baseline RR Scheduler 24
3.2 Latency Breakdown for GPGPU Applications 32
3.3 An Example of Warp Criticality from the GPGPU Application BFS 37
3.4 Latency Breakdown for the BFS Application 38
3.5 The Speedup Comparison for Different Warp Scheduling Policies on BFS 42
3.6 Latency Breakdown for the BFS Application under the Oracle CAWS-avg Scheduling Policy 44
4.1 Warp Execution Time Disparity Caused by Workload Imbalance for BFS 50
4.2 Warp Execution Time Disparity Caused by Diverging Branch Behavior for BFS 52
4.3 Warp Execution Time Disparity Caused by Memory Subsystem Delay for BFS 53
4.4 L1 Data Cache Reuse Distance for the Critical Warps in BFS 54
4.5 Warp Execution Time Disparity Caused by Warp Scheduling Delay for BFS 55
4.6 The CAWA Architecture 56
4.7 The Instruction Count Disparity Caused by Branches 57
4.8 The Criticality-aware Cache Prioritization Scheme 60
4.9 Reuse Behavior of Different PCs for BFS 61
4.10 Performance Improvement of CAWA 68
4.11 L1 Data Cache MPKI Reduction of CAWA 69
4.12 The Prediction Accuracy of CPL 70

Figure | Page

4.13 The Performance Improvement of gCAWS 71
4.14 L1 Data Cache Critical Warp Hit Rate of CAWA 72
4.15 L1 Data Cache MPKI Reduction of CACP with Different Warp Scheduling Policies 73
4.16 L1 Data Cache Performance Improvement of CACP with Different Warp Scheduling Policies 74
5.1 Speedup of Different L1 Data Cache Configurations 77
5.2 An Example of Thrashing in GPU Caches 79
5.3 The Distribution of L1 Data Cache Reuse Distance 80
5.4 The Distribution of L1 Data Cache Reuse Distance per Insertion PC of `BFS` 81
5.5 Speedup with Varying an Instruction's Insertion/Bypassing Ratio 82
5.6 The System Diagram of Ctrl-C Design 87
5.7 The Performance Improvement of Ctrl-C 91
5.8 The L1 Data Cache MPKI Reduction of Ctrl-C 92
5.9 The L1 to L2 Caches Interconnect Traffic Reduction of Ctrl-C 93
5.10 The Fraction of Zero-reuse Cache Lines with Ctrl-C 94
6.1 An Example of a Heterogeneous Computer System with Multiple OpenCL Enabled Devices 99
6.2 The Average Execution Time Speedup and Slowdown Fairness of Running OpenCL Applications 104
6.3 The Execution Time Speedup of an OpenCL Application 105
6.4 The Fairness Ratio between Running an OpneCL Kernel on the CMP versus on the GPU. 107

Figure Page

6.5 The Fairness Ratio of Running an OpneCL Kernel on the CMP versus on the GPU with Varying Scheduling Priorities 108

6.6 System Diagram of the HeteroPDP Prediction Scheme 109

6.7 The Prediction Accuracy of Selecting the Optimal Execution Target Device to Run an OpneCL Kernel 114

6.8 The CDF of Prediction Errors for Predicting OpenCL Kernel Execution Time 115

6.9 The System Speedup of HeteroPDP when Running an OpenCL Application *alone* 116

6.10 The Speedup and Fairness of HeteroPDP when an OpenCL Application is *co-located* with a Native CPU Application 117

6.11 The Prediction Accuracy of Selecting the Optimal Target to Run an OpneCL Kernel Co-located with a Native CPU Application Having Varying Scheduling Weights 118

6.12 The Speedup of HeteroPDP when Running Workloads Consisting of an OpenCL Application and a Native CPU Application with Varying Scheduling Weights 119

6.13 The Prediction Accuracy of Selecting the Optimal Target Device to Run an OpneCL Kernel *co-located* with Two Native CPU Applications . 120

6.14 The Speedup and Fairness of HeteroPDP when Running Workloads Consisting of Two Native CPU Applications and One OpenCL Application 121

Chapter 1

INTRODUCTION

Modern computer systems are accelerator-rich, equipped with many types of hardware accelerators or sensors, e.g., graphics processing units (GPUs), tensor processing units (TPUs) [56], digital signal processors (DSPs), image processors, audio processors, and field-programmable gate arrays (FPGAs) to speed up computation and/or reduce energy consumption [25, 47, 105, 123]. Figure 1.1 exhibits an example of an accelerator-rich heterogeneous system architecture which integrates a variety of execution devices in a single computer machine. The advantage of having such kinds of heterogeneous systems is that workloads can be dynamically distributed to run on different devices based on their characteristics to maximize the overall system throughput.

GPUs are a type of hardware accelerators in modern computer systems. They are pervasively deployed to high performance computing clusters (HPCs). GPUs were initially devised to perform graphics related computations, specifically frame rendering, 3D modeling, video codec, and digital image processing. Nevertheless, the capability of performing massive multithreading and fast context-switching has been the forte of modern GPU architectures, which enables GPUs to accelerate general-purpose parallel workloads such as scientific computation, weather forecasting, as well as machine learning workloads. Therefore, it is getting more and more notice today to offload and execute general-purpose GPU (GPGPU) workloads on the highly-parallel, throughput-oriented architecture.

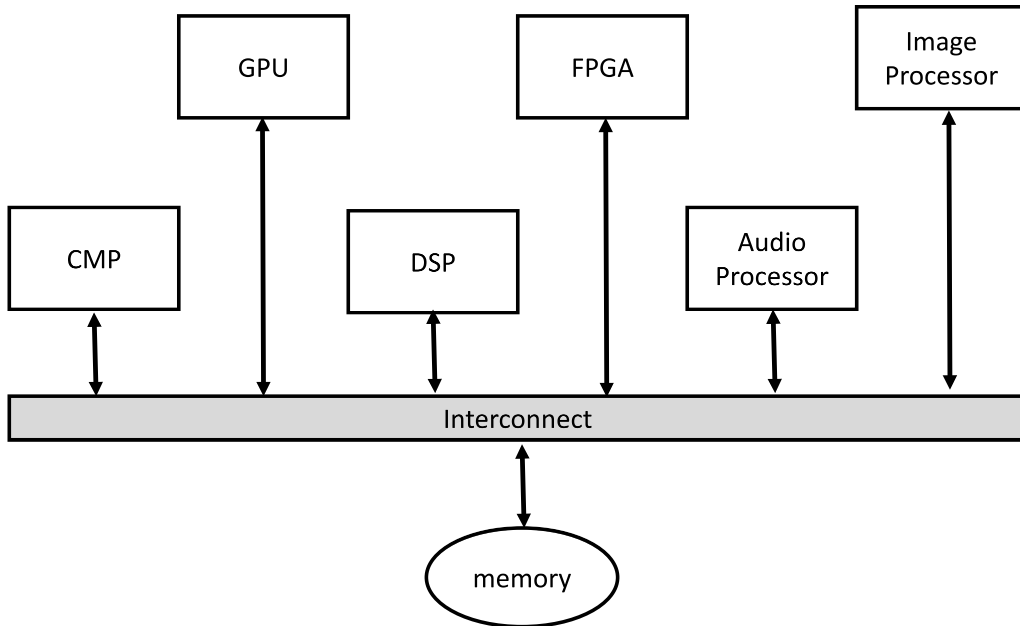

Figure 1.1: An example of an accelerator-rich heterogeneous computer system.

1.1 GPGPU Architecture and Computation

GPUs are based on the single instruction multiple thread (SIMT) computation paradigm where multiple threads are grouped together to form a warp or wavefront. Threads in a warp are mapped to a single instruction multiple data (SIMD) execution unit such that all threads execute the same instructions, but with different data.

The benefit of the large number of warps and fast context-switching is latency-hiding—whenever the execution of a warp stalls, e.g., facing a cache miss and waiting for the data to be ready, it can be swapped out and another warp can be swapped in for immediate execution to maximize resource utilization without paying much context-switching overhead.

A modern GPU consists of multiple streaming-multiprocessors (SMs) or computation units (CUs). Each SM is similar to a SIMD processor, which has vector

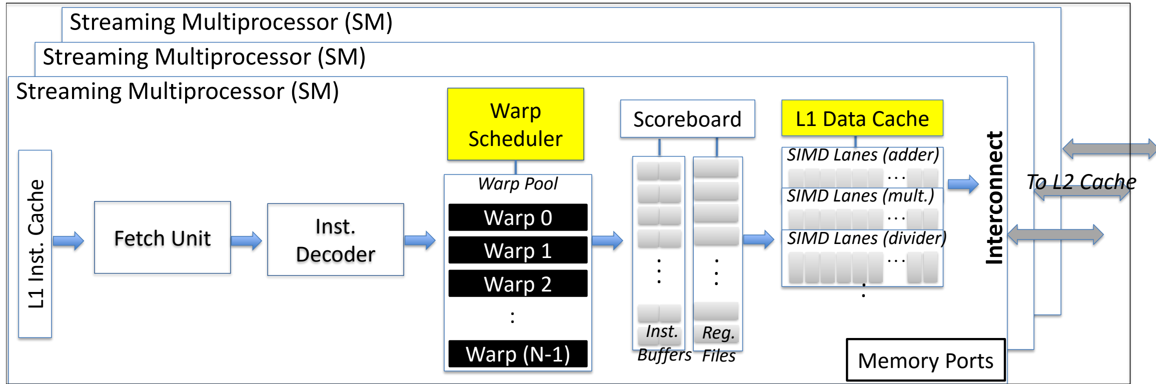

Figure 1.2: An overview of the modern GPGPU microarchitecture.

functional units, register file, cache memories, and instruction fetch/decode units as what Figure 1.2 illustrates [79]. Additionally, an SM also has a warp pool to record the context of all running threads for performing fast-context switching. To order the execution of the large number of parallel warps, an SM employs a hardware warp scheduler to dispatch and allocate computation resources for warp execution. Note that, the components highlighted in yellow in Figure 1.2 are the main components in the GPU microarchitecture which this thesis work focuses on.

At runtime, a GPGPU application first copies data from the host machine to a GPU's memory space. The application then spawns a massive number of threads that execute the same piece of code in a kernel to process the data. Multiple threads are grouped into a small batch, called a thread-block (TB) or a cooperative thread array (CTA). Threads from a thread-block have the same life-cycle and are dispatched to an SM for concurrent execution. For the threads to be executed by the vector functional units in an SM, the threads in a thread-block are split into several warps. All threads within a warp are executed simultaneously by the vector functional units. At every cycle, the warp scheduler selects a ready warp for execution. When a warp stalls, GPUs can perform fast context-switching to process another ready warp without introducing any additional latency. By interleaving the execution with a

large number of parallel warps, GPUs can hide execution latency to maximize the pipeline utilization and achieve a considerable throughput.

1.2 The Inefficiencies of Current GPGPU and Accelerator-rich System Designs

Although modern GPUs can process a large number of threads in parallel, it has been shown that directly employing GPUs to accelerate parallel workloads does not always gain good performance improvements. This is mainly due to three reasons:

1. Not all warps have the same amount of workload. Warps have different number of instructions for execution, and thereby there is an execution time disparity between parallel warps. This execution time disparity can raise a sub-optimal performance problem, called warp criticality.

2. The shared hardware resources are limited, in particular the cache storage and memory bandwidth. It is difficult to fairly accommodate the demands of all running threads. Concurrent running threads compete the shared computation resources that may introduce additional stall cycles, lowering the pipeline utilization.

3. To perform computation on a GPU often requires to copy a large amount of data between the host CMP and the GPU back and forth for communication and synchronization. Because of the limited bandwidth capacity of the system bus and the host main memory, these data transfer operations can incur significant execution time overhead and memory interference, resulting in lower average throughput than always using the host CMP to process the same workload.

1.2.1 Execution Time Disparity and Warp Criticality

In the GPGPU computation paradigm, a thread-block is the basic computation unit dispatched onto an SM. A thread-block contains a number of warps that have the same life-cycle and are bounded to the same synchronization barrier. In other words, warps from the same thread-block start execution at the same time, and are blocked at the same synchronization barrier until all warps finish the associated computation workloads.

However, not all warps have the same amount of workload, and therefore warps do not always finish at the same time. A significant execution time disparity is observed between warps in a thread-block for GPGPU workloads. As a result, fast running warps have to wait at a synchronization barrier or kernel exit point until the slowest running warp, or the so-called critical warp, finishes. This raises two problems that significantly degrade the performance of GPUs. First, the execution time of a thread-block is determined by the execution time of the critical warp. Although faster running warps finish the assigned workloads, they are suspended at a synchronization barrier without performing any meaningful computation. Consequently, it occupies and wastes precious shared hardware resources such as the register file. Second, when faster running warps finish execution and are idle, the number of active warps decreases. In such a scenario, the GPU may not have enough ready warps to hide latency stalls. When a warp, especially the critical warp, stalls, its execution latency will be exposed and thereby the GPU pipeline is not fully utilized. This performance problem is called the *warp criticality* problem.

To address the warp criticality problem, in Chapter 3, I discuss the important factors that contribute to GPU pipeline stall and present the results that quantify the severity of the warp criticality problem for GPGPU applications [68, 69]. I identify

the sources of warp execution time disparity in Chapter 4 [72]. While the average execution time disparity between the warps that share a synchronization point in a thread-block can result in reduced pipeline utilization and lower throughput, the performance of GPGPUs is also significantly constrained by the memory subsystem. I propose a coordinated warp scheduler and cache prioritization scheme, called **C**riticality-**A**ware **W**arp **A**cceleration (**CAWA**), to effectively reduce the degree of warp criticality [72].

1.2.2 Resource Contention and Cache Thrashing

Modern GPUs are often equipped with cache memories to filter out the interconnect bandwidth demands as well as to reduce the average memory access latency. However, because of the massive multithreading computation paradigm, cache capacities, especially the L1 data caches, of GPUs are relatively small. For instance, an SM of the NVIDIA Maxwell GPU can process up to 2048 concurrent threads and has a 24kB L1 data cache shared across all running threads [92]. Namely, on average, each thread can only obtain a few bytes of the data cache storage. Threads contend for the cache storage with each other, resulting in a severe cache thrashing problem, namely, cache lines are frequently swapped in/out without receiving any reuse. Consequently, GPGPU applications do not utilize cache memories efficiently.

The cache inefficiency in GPUs incurs two critical problems which often limit the performance of GPUs. First, due to the cache thrashing problem, many GPGPU applications have high data cache miss rates. GPU caches are not able to effectively reduce the average memory access latency, leading to additional pipeline stalls. Second, a large amount of adjacent data elements brought into the cache with the demanded data is never referenced before being evicted. This injects additional data traffic to the interconnect and can increase the queuing latency in the interconnect.

Due to the unnecessary data traffic, applying caches to preserve spatial localities significantly degrades the performance in some GPGPU workloads instead [51].

Many prior works proposed to apply cache bypassing techniques to alleviate the degree of cache thrashing in GPUs. A widely used approach to bypass memory requests from caches is to employ compilers to perform off-line analysis and identify data that are unlikely to receive any reuse in the near future [51, 76, 77, 121]. However, the compiler-based approaches are not flexible for input dependent applications. In addition to the static compiler based schemes, a number of prior works proposed to use additional hardware components to count and predict the reuse distances of cache lines at runtime [74, 109]. However, the reuse distances of GPGPU cache lines can be extremely long and exhibit a disperse distribution. It is challenging to accurately predict reuse characteristics of GPGPU cache lines with low storage requirement. These dynamic prediction algorithms, therefore, require a large number of hardware counters and incur significant implementation overhead.

To tackle the cache inefficiency problem in GPUs, Chapter 5 of this thesis explores the cache access behavior of GPGPU applications. I propose a low hardware implementation overhead cache bypassing algorithm—**C**on**tr**ol-**L**oop Based Adaptive **C**ache Bypassing (**Ctrl-C**)—for GPGPUs to accurately predict the cache reuse behavior without the need of off-line analysis and dynamically bypass memory requests to prevent cache lines from early eviction [70]. Ctrl-C significantly improves the overall performance of GPGPUs and outperforms other state-of-the-art GPU cache bypassing schemes.

1.2.3 Data Movement Overheads and Memory Interference

In a heterogeneous system, GPUs are usually attached to the host machine via the PCIe or AGP bus interface. When offloading computation onto a GPU card, the

system has to frequently copy data between the host main memory and the GPU internal memory via the bus to synchronize the data. Since the system bus and main memory bandwidth is a limited resource, the data movement operations frequently become a critical performance bottleneck of GPUs and dominate the total execution time [13, 37, 81, 93, 108]. As a result, exploiting GPUs to accelerate computation does not always exhibit better overall throughput than processing on CMPs directly.

In order to eliminate the performance impacts of data movement operations, numbers of prior works, such as [6, 114, 115, 120], developed performance prediction schemes to dynamically make offloading decisions. These works focused on adopting machine learning or compiler techniques to profile and analyze the characteristics of a GPGPU application to understand whether offloading the computation is able to receive performance or energy benefits.

However, apart from the GPU application itself, in a realistic computer system, there are many concurrent processes co-located on the same machine, sharing the system bus and main memory bandwidth. For example, in an on-demand cloud computing environment such as Amazon Web Service (AWS) [3], Google Cloud [36], and Microsoft Azure [24], compute nodes are simultaneously servicing multiple applications or hosting multiple virtual machines with native CPU applications as well as GPU acceleratable applications. In such execution environment, co-located applications contend for shared resources in the memory subsystem. Consequently, existing task scheduling schemes that only consider the characteristics of an application itself but do not take into account memory interference from co-located workloads are not robust and provide sub-optimal performance gain.

To understand the need for an intelligent scheduler that can make an accurate decision for which optimal execution target an application should be executed on in the presence of memory interference, Chapter 6 of this thesis provides a detailed

performance characterization study for accelerate-rich heterogeneous systems. Based on the observations, I design a scalable **Hetero**geneous **P**erformance **D**egradation **P**rediction (**HeteroPDP**) scheme to accurately predict the system performance degradation when an application is running on different execution targets with memory interference [71]. With the prediction outcomes of HeteroPDP, a workload can be dynamically to be dispatched to run on the optimal execution target device based on the optimal goal.

1.3 Contributions

The goal of this thesis is to design architectural- as well as system-level solutions to address the inefficiencies of GPGPU microarchitectures and system architectures of accelerator-rich computers equipped with GPUs. Specifically, the thesis focuses on discussing the warp criticality, cache contention, and memory interference problems in GPGPUs. Besides, this thesis also provides detailed characterization studies and new insights of GPGPU architecture designs from different aspects, including warp scheduling algorithms, memory management techniques, and performance predictions. Overall, this work makes the following key contributions:

1. Providing a detailed characterization of the latency hiding ability of GPGPUs.
2. Identifying the warp criticality problem and providing an in-depth study of the warp execution time disparity in the massive multithreading computation of GPGPUs.
3. Designing a coordinated warp scheduling and cache prioritization solution to efficiently eliminate the warp criticality problem in GPGPUs.
4. Developing a control loop based cache bypassing algorithm to intelligently mitigate the cache contention problem in GPGPUs.

5. Analyzing the system performance degradation in the presence of memory interference in a CPU-GPU multiprogrammed computing environment.

6. Proposing a performance degradation mechanism to balance the execution time slowdown and maximize the overall system throughput for accelerator-rich computer systems.

The following chapters of this thesis present my research accomplishments in detail. The rest of this thesis is organized as follows:

1. Chapter 2 discusses prior studies related to this thesis work.

2. Chapter 3 describes the warp criticality problem and shows the characterization results for its impact on the performance of GPGPU workloads.

3. Chapter 4 presents a solution that accelerates the execution of critical warps, called Criticality-Aware Warp Acceleration (CAWA).

4. Chapter 5 demonstrates a control loop based adaptive cache bypassing (Ctrl-C) algorithm to effectively mitigate cache contention in GPGPUs.

5. Chapter 6 presents a performance degradation prediction (HeteroPDP) scheme to accurately predict and balance the system performance degradation in the presence of memory interference in a heterogeneous system setup.

6. Chapter 7 summarizes this thesis work and makes the conclusions.

Chapter 2

RELATED WORK

To better understand the context and novelty of my thesis work, this chapter focuses on reviewing and discussing prior studies in the related areas.

While the goal of this thesis work is to solve three inefficiencies in modern GPGPU microarchitectures and accelerator-rich heterogeneous systems (i.e., the warp criticality, the cache contention, and the system memory interference problems), I will first review the prior studies relevant to warp criticality. I will then present the works regarding cache management in CPUs and GPUs. Finally, I will introduce the designs related to the shared system resource management as well as the task scheduling in heterogeneous computer systems.

2.1 Warp Criticality

Thread Criticality in CMPs. The concept of thread criticality in CMPs is similar to the warp criticality problem in GPUs. A multithreading application often applies barriers to synchronize between threads. However, not all threads arrive at a barrier at the same time. Fast running threads are idle at a barrier to wait for the slowest running thread. The execution time of a parallel application is dominated by the execution time of the critical thread on CMPs. In order to improve the system performance, it is important to identify the critical thread in advance and accelerate the critical thread execution.

Li et al. pointed out that some threads in CMPs are often idle to wait for slower running threads, resulting in energy waste [75]. In order to save energy, Liu et al. presented a probability model to estimate the thread running time and guide the dy-

namic voltage and frequency scaling (DVFS) of CMPs [80]. Cai et al. proposed using compilers to insert check points in the parallel regions that all threads execute to evaluate the execution speed of each thread [17]. By monitoring the time a thread reaches the check points, the critical thread can be detected. Bhattacharjee and Martonosi observed that the critical threads often encounter more cache misses and have longer average memory access latency [14]. Bhattacharjee and Martonosi designed a thread criticality predictor (TCP) by monitoring the per-thread cache access behavior. TCP is then used to guide the task stealing as well as DVFS of CMPs. Ebrahimi et al. exploited the degree of resource contention at a spin lock as the metric to predict the critical thread [30]. If a lock is frequently held by a particular thread, this thread has likelihood to be the critical thread. Bois et al. proposed a stack based approach to measure thread criticality by monitoring the number of waiting threads in a certain time interval [15]. A thread has a higher degree of criticality if there are more threads waiting at a spin lock when this particular thread performs computation. Turakhia et al. observed that the number of instructions in a code section (i.e., the code between two consecutive barrier instructions) has locality [111]. In other words, two consecutive code sections often have similar number of instruction counts. Based on this observation, Turakhia designed a thread progress equalization (TPEq) scheme to predict the degree of thread criticality by predicting and calculating the distance to reach a barrier.

Although the concept of thread criticality in CMPs is similar to warp criticality in GPUs, due to the distinct difference between CPU and GPU architectures and computation paradigms, the effects introduced by the critical threads and critical warps vary as well. Because GPU has a large number of parallel warps and frequently switches the execution between the parallel warps, there are more factors that can

lead to warp criticality as I will present in Chapter 4. It is still difficult to accurately predict the critical warp with these thread criticality prediction algorithms for CMPs.

GPGPU warp scheduling. While the warp criticality problem can substantially limit the performance of GPGPU workloads, this thesis work proposes using a criticality-aware warp scheduling algorithm to eliminate the warp execution time disparity. Next, I present the state-of-the-art warp scheduling algorithms to better understand the design of warp schedulers.

Many prior works focused on improving the performance of GPUs by modifying warp scheduling algorithms to prevent warps from stalling. Gebhart et al. and Narasiman et al. designed a 2-Level scheduler to split warps into different subgroups and keep only one group of warps active at a time [35, 87]. The warp scheduler is only able to issue instructions from the active subgroup of warps, so that the resource contention problem can be alleviated. Jog et al. further improved the 2-Level scheduler by assigning warps with continuous IDs to different subgroups [54, 55]. Because memory requests from continuous warps have higher probability to fall into the same L2 cache or DRAM bank, resulting in bank conflicts and longer memory access latency. With this warp grouping algorithm, the GPU performance can be improved by avoiding bank conflicts at the L2 cache and DRAM. Rogers et al. proposed a cache conscious mechanism to monitor and measure the degree of memory contention by a loose locality score (LLS) [101, 102]. The warp scheduler then dynamically modulates the number of active warps based on the degree of LLS value. If the cache controller detects cache lines in the L1 data cache are frequently evicted due to the interference from inter-warp accesses (LLS value is high), the warp scheduler will decrease the number of active warps. While applications might have different preferences of warp scheduling policies, Awatramani et al. proposed a phase-aware warp scheduling algorithm which applies compilers to analyze the GPGPU

kernel source code and select the optimal warp scheduling policy, whereas Lee et al. designed an adaptive algorithm to dynamically select the optimal scheduling algorithm based on the instruction issue pattern at runtime [10, 67].

These proposed warp scheduling algorithms aim to prevent all warps from stalling at the same time due to long latency memory operations to improve the pipeline utilization. These scheduling policies allow the warp scheduler to tolerate memory latency better by reducing the idle time of GPU pipeline. However, these warp scheduler designs do not take the impact of warp criticality into account. The warp criticality problem can still limit the performance of GPGPU workloads. In contrast, my proposed criticality-ware scheduling design (Chapter 3 and 4) in this thesis aims to resolve resource contention by ordering the warp execution based on warp criticality and allowing critical warps to execute with larger time slices. Therefore, the performance of GPGPU workloads can be significantly improved.

2.2 Cache Thrashing Problem and Cache Management

In addition to the warp criticality problem, memory contention and cache thrashing is another main problem limiting GPU's performance. To design a new cache management policy to improve the cache efficiency in GPGPUs, I intend to review prior studies that focused on lessening the degree of cache thrashing in CPUs and GPUs next.

CPU cache management. Many cache management policies have been proposed to mitigate cache thrashing in CPUs. Jiménez designed a tree-based pseudo LRU (pLRU) cache replacement policy, which exploits machine learning techniques to find out the optimal promotion and insertion position for cache lines on an LRU stack [53]. Qureshi et al. proposed the BIP cache insertion policy and set dueling mechanism to insert new cache lines at the LRU position to achieve the optimal hit rate when cache

thrashing occurs [97, 98]. Jaleel et al. designed an RRIP algorithm which predicts the reuse distance of a cache line by giving each cache line a re-reference prediction value (RRPV) and updating RRPV when a set is accessed [49]. Wu et al. proposed a signature based framework, SHiP, to predict the reuse distance of an incoming cache line based on the particular signature of a memory request, e.g., the insertion program counter value and memory address [119]. Arunkumar and Wu designed a reuse-and-cost aware memory access (ReMAP) scheme that takes the DRAM access latency into account to select the best cache eviction candidate [8]. Lai et al. and Khan et al. proposed dead block sampling algorithms to predict if a cache line will not be reused in the near future [61, 63]. The dead block then can be bypassed or evicted from the cache. However, all these reuse distance prediction works were built on top of CPU's last level caches, which usually have higher associativity with the capacity in MB scale. For instance, the Intel Core i7-2600 CPU is equipped with a 8MB L3 cache [44]. While the L1 data caches in GPUs are much smaller and have lower way-associative, the cache trashing problem is severer. These CPU cache management algorithms are not able to accurately predict the data reuse patterns in GPUs.

In order to prevent cache lines from early eviction, Dung et al. proposed a PDP protect algorithm to bypass part of memory requests [29]. In PDP, each cache line has a protection counter which is decremented by one when the corresponding set is accessed. A cache line can be evicted only when its protection counter reaches zero. If no line has a zero protection value, then the new incoming memory request will be bypassed. PDP guarantees a cache line will not be evicted within a short time period. However, the reuse distances for GPGPU workloads can be extremely long and often have a disperse pattern. It is difficult to predict and set up an optimal protection distance.

GPU memory and cache management. Stratton et al. conducted a detailed characterization study and suggested that resource contention in the memory subsystem is a critical performance problem limiting GPGPU performance [106, 107]. In order to alleviate the memory contention in GPUs, Lee et al. designed a compiler-based scheme to predict the per-thread working set size [66]. According to the prediction outcome, GPUs can then limit the number of active threads to regulate the degree of memory contention. Chatterjee et al. proposed a sub-channel architecture specifically for GPU DRAMs to mitigate the degree of contention in DRAM row buffers [18]. Choo et al. observed that using unified L1 data caches shared across multiple SMs can improve inter-warp locality and mitigate cache contention [23]. Sethia and Mahlke designed an Equalizer scheme that can dynamically monitor the demand of different shared resources in GPUs [104]. If Equalizer detects the warp execution time is dominated by memory access time, it throttles the warp execution by stopping creating new thread-blocks on an SM.

Cache bypassing is an approach for balancing cache capacity scaling and its utilization [85]. To effectively improve GPU cache utilization and mitigate the degree of cache contention, many cache bypassing algorithms have been proposed. Jia et al. designed a FIFO queue (MRPB) to reorder requests to reduce inter-warp contention [52]. Additionally, MRPB bypasses requests if intra-warp memory contention is detected. Chen et al. designed an adaptive resource management scheme that monitors cache contention and interconnect congestion [22]. If the degree of cache contention or bandwidth demand is too high, memory requests will be bypassed. Mahmoud et al. proposed using cache miss rate as a metrics to evaluate if an application is a streaming workload and make cache bypassing decision accordingly [60]. However, these prior designs do not distinguish reuse patterns among memory re-

quests. Cache lines with near reuse distances may be bypassed, losing an opportunity to improve the cache hit rate.

Xie et al. and Liang et al. modified compilers to analyze GPGPU applications source code and guide GPUs to bypass data which are unlikely to receive any reuse [76, 77, 121]. Li et al. proposed a valley model to guide compilers analyzing if a GPU application can benefit from cache bypassing [73] for varying number of spawned threads. However, these compiler-based schemes are not able to predict the reuse behavior of input dependent applications, e.g., applications with pointer chasing execution behavior.

Tian et al. proposed the PC-based Adaptive Bypassing that uses confidence counters to predict zero-reuse lines and bypasses all requests if detecting cache lines will not receive any reuse [109]. Lee et al. designed a region-aware caching mechanism (GREEN) [65]. GREEN dynamically tracks the degree of locality and selectively bypasses data that are located in the memory regions with poor locality. Li et al. suggested adding additional tag array entries to track the data reuse patterns [74]. Nevertheless, the reuse distance can be extremely long for GPUs. It is challenging to accurately predict the data reuse patterns with a limited number of confidence counters or tag array entries.

Zheng et al. designed an adaptive cache and concurrency allocation (CCA) scheme to dynamically trace the per-warp memory access footprint [125]. CCA then prevents cache lines from early eviction by limiting the number of warp allowed to access to the cache memory. Dai et al. developed a model-driven approach that dynamically estimates the cache hit rate as well as execution time speedup improvement if reducing the number of warps that can allocate the data cache storage [27]. The model-driven approach then intends to bypass memory requests issued by a set of designated warps to maximize the cache hit rate. However, with the imbalanced memory access time

created by these works, the model-driven approach may increase the degree of warp criticality, degrading GPU pipeline utilization.

Koo et al. developed an access pattern-aware cache management policy (APCM) that dynamically identify the locality type (inter-warp locality, intra-warp locality, and streaming) of each memory instruction [62]. If a memory instruction is predicted to have streaming access behavior, APCM will then bypass requests issued by this particular instruction. However, APCM bypasses all the memory requests if it predicts the memory access pattern is streaming. In such case, the hardware tracker loses the information to examine the cache access pattern. It is difficult to identify whether a bypassing decision is correct or not.

In contrast, in this work, I propose a low circuit implementation overhead design—Ctrl-C (Chapter 5)—to dynamically learn the cache line reuse behavior and perform selectively cache bypassing to alleviate the cache thrashing problem in GPGPUs without a need of off-line analysis.

2.3 Memory Interference in Heterogeneous Systems

Memory Interference and Management The shared resource contention in the CMP domain has been studied by an extensive amount of prior works. These works mainly focused on discussing managing the capacity and bandwidth of the shared memory subsystem. Mutlu and Moscibroda proposed a stall-time fair DRAM scheduling algorithm to reduce the performance degradation and improve system slowdown fairness caused by shared resource contention in the DRAM modules by dynamically assigning different DRAM access priorities to the co-scheduled threads [86].

In order to understand the effects of cache interference in a CMP system, Hsu et al. conducted a detailed characterization study to analyze the performance impacts with different shared cache partition strategies. To mitigate the shared last-level cache

contention, Jaleel et al. proposed a thread-aware dynamic insertion policy (TADIP) to monitor and select the insertion policy for co-located applications that share the last-level cache [48]. Qureshi and Patt designed a utility-based cache partitioning (UCP) algorithm to eliminate the shared cache interference in a multiprocessor system by allocating different size of cache storage to each co-located application based on the cache utilization [96]. In order to solve the underutilization problem and reduce the implementation overhead of UCP, Xie and Loh proposed a pseudo partitioning (PIPP) scheme that simulate the cache partitioning algorithm by inserting a new cache line at different positions [122]. Wang and Chen proposed a futility scaling (FS) mechanism that targets at partition the cache storage for co-located processes without losing the cache associativity [113]. Intra-application cache interference stemmed from operating system activities and hardware prefetching can occur and degrade an application's performance as well. Wu and Martonosi studied the intra-application cache interference problem and proposed an OS-aware cache insertion policy to eliminate the intra-application interference by prioritizing the memory requests asserted by kernel- and user- space processes [118]. In order to accommodate the shared cache resources for processes with different OS scheduling priorities, Wu and Martonosi developed a adaptive timekeeping replacement (ATR) policy to dynamically adjust the cache decay intervals based on the optimization target [117].

In addition to using the architectural-level solutions to alleviate the shared resource contention problem, many works targeted at designing software scheduling algorithms to allocate the shared resources. Mars et al. designed a low overhead algorithm, called Bubble-up, to predict the degree of shared resource contention and to schedule services to run on different computation nodes in data center execution environments [82]. The Bubble-up algorithm aimed to maximize the per-node loading without violation the real-time deadline or quality-of-service (QoS) constraints. On

the other hand, Jaleel et al. proposed a cache replacement and utility-aware scheduling (CRUISE) targeting at coordinating the OS scheduling and cache replacement policy to maximize the system throughput on a single CMP machine [50]. To schedule workloads running on a single-ISA heterogeneous multiprocessor system (e.g., the big-little core architecture [7]), Craeynest et al. proposed a performance impact estimation (PIE) algorithm to predict the performance when a program runs on the other core [26].

Nevertheless, all of these existing solutions looked at the homogeneous architecture domain only. In contrast, in my thesis, the proposed HeteroPDP scheme (Chapter 6) targets at predicting and mitigating the degree of shared resource contention specifically in heterogeneous computer systems.

Shared Resource Management for Heterogeneous Systems Since many commercial products have integrated CPU and GPU cores into one single die, how to efficiently manage the shared resources between the different types of processors is a significant research problem, especially for the shared last-level cache [41, 42]. Lee and Kim proposed a thread-level parallelism aware policy (TAP) to partition the shared cache storage for co-located CPU and GPU workloads [64]. Mekkat et al. developed an algorithm, called HeLM, to dynamically determine the priority of CPU and GPU cache accesses [83]. Kayıran et al. designed a concurrency management scheme that mitigates the memory bandwidth contention in a heterogeneous system by regulating the number of concurrent running on the GPU cores [57]. García et al. quantified the impact of shared virtual memory space between the CPU and GPU cores and suggested that developers have to redesign OpenCL programs to leverage the utilization between CPU and GPU cores to optimize the system throughput [34]. Ausavarungnirun et al. developed a staged DRAM controller that aims to improve the fairness of CPU-GPU shared DRAM by using dedicated CPU and GPU request

queues in the memory controller and treat the CPU/GPU requests with different priorities [9]. Seo et al. designed a memory-aware load balance algorithm (MLB) [103]. MLB aims to balance the performance degradation by allocating more DRAM bandwidth for data movement between the host CPU and the hardware accelerators. None of these works, however, addressed the shared resource contention problem from the aspect of task scheduling by taking into account the degree of memory interference from multiple levels of the memory hierarchy.

OpenCL Kernel Scheduling Many prior works have pointed out that employing GPUs to accelerate OpenCL kernels does not always lead to performance improvement, due to the data movement and synchronization overhead [13, 37, 42, 81, 93, 108, 124]. In order to identify the optimal execution target device to run an OpenCL kernel, many works proposed applying a variety of machine learning techniques to dynamically analyze and predict the behavior of an OpenCL kernel. Wu et al., designed a performance and power predictor for GPUs by adopting the K-means algorithm [120]. Wen et al., proposed using support vector machine (SVM) to model the performance gain of GPUs [115]. Ardalani et al. employed regression models to project the GPU kernel execution time by running the same kernel on CMPs [6]. Wen and O'Boyle designed a decision tree based algorithm to analyze the performance benefits that offloading an OpenCL to run on an accelerator [114]. Aji et al. designed a set of OpenCL API extensions enabling compilers to guide the OpenCL scheduler select the optimal target device at runtime [1, 2].

Instead of predicting the performance gain, a number of studies focused on minimizing the data transfer overhead. Lustig and Martonosi developed a fine-grained synchronization mechanism to early start the GPU kernel execution and hide the data transfer latency [81]. Ham et al. proposed a supply-compute framework (DeSc) which decouples the communication and compute engines to hide the data transfer overhead

by out-of-order executing data fetching and computation [38]. Belviranli exploited just-in-time (JIT) compilers to hide the communication overhead by automatically reordering the GPU application programming interface (API) calls to overlap the data transfer operations [13].

However, none of the prior works takes the shared resource interference introduced by the co-located applications into account in making a scheduling decision. They simply took an individual GPU kernel's characteristics to do performance prediction and optimization. While a realistic machine can service several processes or applications simultaneously, these designs are not robust. Instead, in this work the proposed HeteroPDP in Chapter 6 aims to optimize the performance for the entire system.

Chapter 3

WARP CRITICALITY AND CRITICALITY-AWARE WARP SCHEDULING

Modern GPUs achieve a high throughput by applying massive multithreading and fast context-switching to hide the execution latency. When the execution of a warp stalls, the warp can be swapped out and another ready warp can be swapped in for execution. These stalls could be caused by cache misses or pipeline hazards, e.g., data or structural hazards.

3.1 The Baseline GPGPU Architecture and its Computation Model

A GPGPU application is a highly multithreading program. Massive number of parallel threads execute the same program code, called a *kernel*, with different data. At runtime, a GPGPU application creates multiple thread-blocks to perform parallel computations, where a thread-block is an array of concurrent threads that are dispatched to run on a GPU shader core together. Threads from the same thread-block share global data and synchronize at barriers. A programmer can invoke barrier instructions (the API calls of ___*syncthread()* in CUDA [91] or *barrier()* in OpenCL [16] semantics) explicitly to block the thread execution and make threads synchronize. In addition to the explicit synchronization barriers, threads are automatically blocked to synchronize when they finish their own workloads as well, i.e., an implicit synchronization barrier can be observed at the end of kernel code. When reaching a barrier point, either explicit or implicit, a thread has to stop execution until all threads from the same thread-block reach the same barrier.

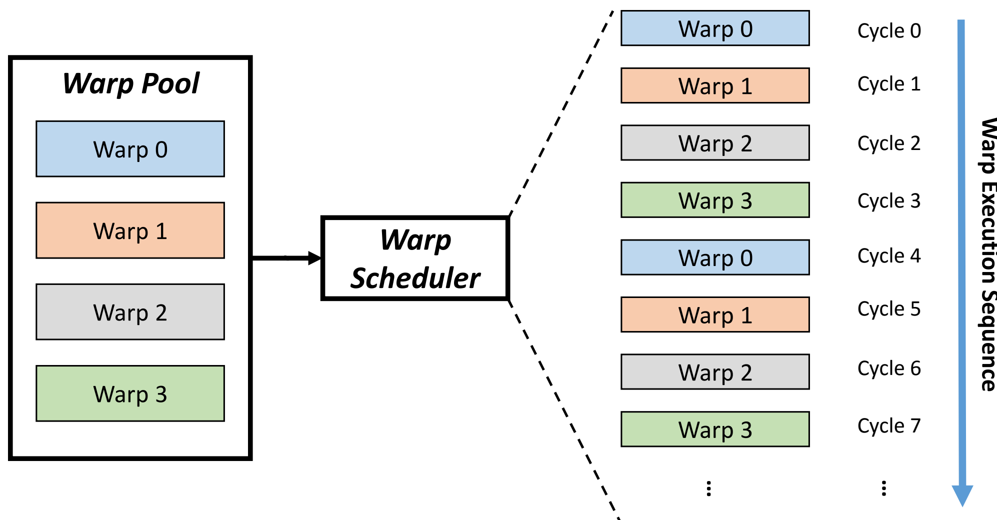

Figure 3.1: The execution order with the baseline RR scheduler.

A modern GPU is a cluster of multiple shader cores, called streaming multiprocessors (SMs) or computation units (CUs), where each SM is a unified graphics and computing processor that can execute graphics rendering or general-purpose computations as shown in Figure 1.2 [79]. An SM is a *single instruction multiple data* (*SIMT*) processor, which processes massive number of parallel threads from the same GPGPU program with vector functional units. In order to fit into the width of the vector functional unit, multiple parallel threads from the same thread-block are further grouped into a small batch, called a *warp*. A warp is the basic unit to be scheduled and executed in the GPU pipeline.

In order to efficiently execute the large number of parallel warps, an SM applies a warp pool and warp scheduler to manage the warp execution as highlighted in Figure 1.2. The warp pool records the context of all running warps. At each cycle, the warp scheduler selects a ready warp from the warp pool to be issued and executed. The warp scheduler orders the warp execution based on its warp scheduling algorithm.

For example, with the baseline round robin (RR) scheduling policy, the warp scheduler iteratively selects a warp from the warp pool to executed as illustrated in Figure 3.1. While the warp execution is interleaved, the warp execution latency can be hidden. For example, in Figure 3.1, a warp is selected to be execute every four cycles. Any operation with a latency fewer than 4 cycles will be hidden well and not affect the pipeline throughput.

3.2 GPU Latency Hiding Ability

At every cycle, GPU's warp scheduler selects an available warp from the warp pool for execution as discussed in Chapter 3.1. If there is a ready instruction in the selected warp's instruction buffer without any pipeline hazards, the warp is ready for execution; otherwise, the selected warp stalls. By interleaving the execution of warps, GPU can hide the warp stall cycles and maximize the computation throughput. However, GPUs are not always able to achieve a high throughput. To understand this sub-optimal performance problem, I delve deep to investigate GPU's latency hiding ability.

3.2.1 Factors Stalling Warp Execution

In order to evaluate the effectiveness of the modern GPU's latency hiding ability, I first need to identify the sources of warp execution time delays. In the following, I explain each of the potential factors that can delay a warp's instruction from getting executed:

1. **Warp Scheduling Delay.** At every cycle, the warp scheduler selects a warp for execution based on the warp scheduling policy. In the baseline GPU architecture, a fair round robin (RR) scheduler is employed to select warps in the warp pool. For instance, if there are 48 active warps in the warp pool, the RR

warp scheduler will iterate over the 48 warps cycle-by-cycle to issue one instruction per warp. A warp is selected for instruction issue and execution every 48 cycles and will have up to 47 cycles delay from the RR scheduling policy to hide any stall cycle from itself.

2. **Instruction Buffer and Instruction Cache Miss.** In order to store instructions fetched from the instruction cache, each warp has an instruction buffer. When a warp selected for execution has no valid instruction in its instruction buffer, additional latency penalty is paid due to the empty instruction buffer caused by instruction cache misses.

3. **Structural Hazard.** If there is an available instruction in the selected warp's instruction buffer, the instruction will be placed in the scoreboard while also accessing the source operands in the register file. This is when structural hazards caused by contentions in the register file banks and at the various functional units are examined. If the decoded instruction cannot proceed for execution due to the unavailability of the register file bank or due to the unavailability of the required functional unit, the warp stalls.

4. **Control Hazard.** Unlike CMPs that are often equipped with advanced branch predictors, modern GPUs do not currently implement branch prediction logics and rely on the massive multithreading feature to overlap the latency caused by control hazards. However, it is possible for a warp to experience additional control hazard stalls. For example, if a branch or function call instruction falls onto the *taken* path for a particular warp, and there is no other active warp in the pool at this time instance to help hide the branch address resolution latency, the particular warp then has to spend additional cycle(s) until its target address is calculated.

5. **Data Hazard.** In addition to the stalls coming from structural and control hazards, data dependency can introduce additional penalty to an active warp. Currently, there is no data forwarding logics implemented in GPUs. Therefore, warps have to wait until data dependency is resolved before it can proceed execution. If an instruction is dependent on an older load instruction experiencing a cache miss, this particular dependent instruction will spend a significant amount of stall cycles until the data hazard is cleared.

6. **MSHR and Data Cache Miss.** Memory load and store instructions can experience additional stall cycles if the miss status holding registers (MSHRs) for the data cache are highly contended or if a data cache miss is encountered. Depending on the availability of the MSHRs and where the requested data resides, the amount of latency can vary.

7. **Synchronization Primitives** In addition to pipeline hazards, cache miss, and scheduling latencies, implicit and explicit synchronization primitives in GPU programs can make warp execution stall as well. This delay mainly comes from how the communication between the parallel warps is structured.

3.2.2 Latency Attribution Algorithm

In order to understand how the different stall factors can contribute to a warp's execution time, I develop a latency attribution algorithm to count and reason about where stall cycles come from for warps. Because latencies caused by the various stall factors can be significantly overlapped in GPUs, it is a challenging task to accurately and faithfully attribute stall cycles to the corresponding cause.

At every cycle, the warp scheduler looks for a ready warp by iterating over the active warps in the instruction issue stage at the GPU pipeline as illustrated in Al-

Algorithm 1 The warp scheduler in GPUs.

```
1:  function WarpScheduler
2:      while NoReadyWarp AND NotVisitedAllWarps do
3:          ▷ visiting the next warp based on the scheduling order
4:          w ← nextWarp()
5:          probe(w)
6:          if w is ready then
7:              ▷ executing the instruction from w issue(w)
8:          end if
9:      end while
10: end function
```

gorithm 1. For each of the visited warps, whether they are ready or not, I record the execution status of the warp by instrumenting the baseline warp scheduler's implementation with a monitoring function, called *probe()* (Line 5 in Algorithm 1). If the visited warp is not ready, the monitoring function will investigate the sources of stalls and update the corresponding latency counting logics; otherwise, the visited warp is ready for execution.

Next, I delve deeper into the monitoring function *probe()* and present the latency attribution algorithm. First, I calculate the time during which a warp does not execute any new instruction because of the scheduling policy (Line 3 – 5 in Algorithm 2). This is determined to be the difference between the time when a warp is checked and the time when this particular warp was last checked. The time difference signifies how long the selected warp needs to wait until it can potentially issue the next instruction, i.e., the scheduling delay.

After the scheduling latency is determined, the monitoring function next examines whether the selected warp is ready or not. If ready, the *w.Exec* counter is incremented

Algorithm 2 The warp latency attribution function.

```
1: function PROBE(w)                              ▷ w: context of the input warp
2:     ▷ calculating the scheduling delay
3:     t ← CurrTime − w.PrevTime
4:     w.Scheduling ← w.Scheduling + t
5:     w.PrevTime ← CurrTime
6:     if waitingatasynchronizationbarrier then
7:         w.Sync ← w.Sync + 1
8:     else if instructionbufferisempty then
9:         w.Fetch ← w.Fetch + 1
10:    else if branchtaken then
11:        w.CtrlHazard ← w.CtrlHazard + 1
12:    else if datadependencydetected then
13:        if causedbyadatacachemiss then
14:            if w.CurrPendingAddr ≠ w.PrevPendingAddr then
15:                w.DataHazard ← w.DataHazard + 1
16:                w.PrevPendingAddr ← w.CurrPendingAddr
17:            else
18:                w.DataCacheMiss ← w.DataCacheMiss + 1
19:            end if
20:        else
21:            w.DataHazard ← w.DataHazard + 1
22:        end if
```

```
23:      else if functionalunitunabailable then
24:          w.StrlHazard ← w.StrlHazard + 1
25:      else
26:          w.Exec ← w.Exec + 1
27:      end if
28: end function
```

(Line 26 in Algorithm 2). Otherwise, the monitoring function investigates the stall factors sequentially – synchronization primitives, no available instructions in the instruction buffer or instruction cache misses, control hazard, data hazard, data cache misses, and structural hazard. Because a warp can be stalled due to multiple stall factors at the same time (the latency hiding feature by the massive multithreading GPUs), I want to be cautious about not double-counting latencies overlapped by several stall factors. For example, if a warp stalls due to data dependency on an older, load instruction that misses in the data cache, I attribute only the first stall cycle to the data hazard factor and any additional, subsequent stall cycle(s) to the data cache miss factor. This is because the underlying reason for the warp stall is the data cache miss encountered by the previous cache miss. If there is no following dependent instruction on a previous load cache miss (while the cache miss is being serviced), this latency attribution algorithm does not attribute any stall cycle for this data cache miss. Therefore, it is important to note that the latency shown under the data cache miss category represents only the latency penalties that stall pipeline execution.

I also want to highlight that since GPUs rely on the fast context-switching between the massive number of available warps to improve pipeline utilization, even if a warp encounters no delay from pipeline hazards, memory accesses, or synchronization overhead, it still suffers from delays caused by the scheduling policy. For

Table 3.1: GPGPU-sim configurations for latency characterization.

Architecture	*NVIDIA Fermi GTX480*
Num. of SMs	*15*
Max. # of Warps per SM	*48*
Max. # of Blocks per SM	*8*
Num. of Schedulers per SM	*2*
Num. of Registers per SM	*32768*
Shared Memory	*48KB*
L1 Data Cache	*16KB per SM (32-sets/4-ways)*
L1 Inst Cache	*2KB per SM (4-sets/4-ways)*
L2 Cache	*768KB (64-sets/16-ways/6-banks)*
Min. L2 Access Latency	*120 cycles*
Min. DRAM access Latency	*220 cycles*
Warp Size (SIMD Width)	*32 threads*

example, if a fair RR scheduling policy is applied for a GPU application that has 48 warps running on an SM, each warp will spend 47 cycles waiting until its next ready instruction can be executed. These 47 scheduling cycles will be used to overlap with other latencies caused by e.g., structural hazards, for a particular warp. These stall cycles are attributed to the scheduling latency instead of structural hazards, since the additional latencies are hidden by the warp scheduler which overlaps the scheduling latency with the execution of other ready warps.

3.2.3 Characterization Methodology

In order to investigate the warp criticality problem, I use GPGPU-sim simulator version 3.2.0 [11] to profile the behavior of GPGPU applications. GPGPU-sim is a

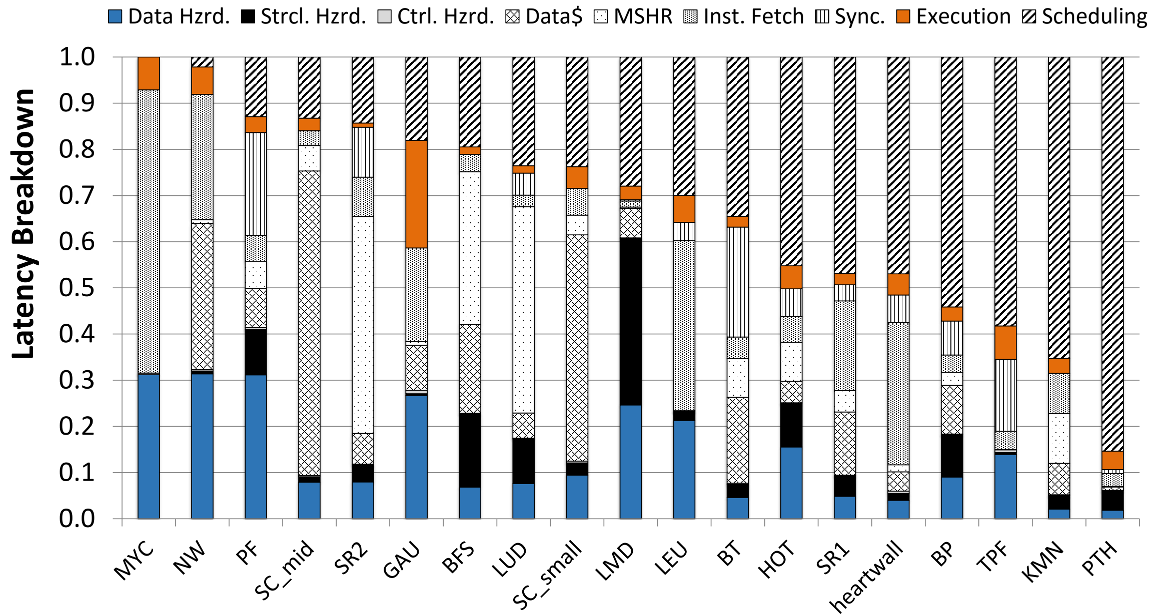

Figure 3.2: Latency breakdown for GPGPU applications. Applications are sorted by the latency hiding ability (*Scheduling*).

cycle-level performance simulator that models a general-purpose GPU architecture supporting NVIDIA CUDA [91] and its PTX ISA [88]. I run GPGPU-sim with the default configuration representing NVIDIA Fermi GTX480 architecture. Table 3.1 describes the simulation configuration and parameters in detail.

In addition to GPGPU-sim simulator, I choose 18 GPGPU applications from the Rodinia [19] and Parboil [107] benchmark suites to characterize the latency-hiding feature in modern GPUs. Table 3.2 lists the details of these 18 benchmarks along with the input data set used for my characterization study, and the computation/memory-intensive characteristics.

3.2.4 Latency Characterization

Next, I apply the latency attribution algorithm to investigate GPU's latency hiding ability and analyze the source of warp stall. Figure 3.2 shows the latency characterization results for GPGPU applications. The x-axis presents the latency breakdown of the GPGPU applications sorted by the degree of the scheduler's latency-hiding ability while the y-axis shows the latency stalls attributed by the various factors. The latency results are the average number across all warps in each of the applications. The bars illustrate the stall cycles contributed by each stalling factor. Applications toward the right are the ones that benefit from the baseline RR scheduling policy whereas the applications toward the left are the ones whose latency stall cycles cannot be hidden by the scheduler.

First, I investigate the effectiveness of latency hiding ability of the baseline RR scheduling policy by focusing on the *Scheduling* bars. The RR scheduling policy is able to hide the majority of the warp stall cycles in applications, such as `BP`, `TPF`, `KMN`, and `PTH`. These applications are considered as *well-behaving* GPGPU applications. On the other hand, the latency-hiding ability of the RR scheduling policy is poor for applications such as `MYC` and `NW`. This is because there lacks warp-level parallelism in these two applications. In other words, there are not sufficient active warps in the warp pool.

The next two factors dominating stall cycles are from *Inst. Fetch* and *Data$* – stalls caused by waiting for ready instructions in the instruction buffer and by waiting for data to be served from the cache memory. Applications such as `MYC` and `NW`, suffer from instruction fetch stalls significantly. This is again due to the lack of warp-level parallelism. When there are enough active warps in the pool, warps will *prefetch* instructions for each other. However, since the number of active warps in these two

applications is small, instruction fetch often results in a cache miss. Furthermore, the scheduler cannot overlap the instruction fetch latency with other concurrent warps. As a result, the applications spend a significant amount of time waiting for available instructions for execution.

Applications such as `SC_small` and `SC_mid` suffer from long-latency data cache misses. The significant amount of stall cycles caused by data cache misses indicates that these two applications heavily accesses the memory hierarchy and suffer significantly from its low degree of memory-level parallelism.

I next look at the amount of latency stalls contributed by the three classic types of pipeline hazards, i.e., data, structural, and control hazards. Data hazard stalls are caused by the particular instruction ordering within an application. I observe that applications, e.g., `MYC`, `NW`, and `PF`, suffer from data hazard stalls more than other GPGPU applications. Structural hazard stalls are caused by the competition for pipeline resources, in this case, the functional units in the pipeline. Applications such as `LMD`, spend a significant amount of time waiting for the structural hazard to be resolved. This is because the warps in `LMD` heavily compete for the load/store unit in the pipeline. Over all GPGPU applications, I do not see much stall penalty caused by control hazards. Compared with CPU applications, GPGPU applications contain less branch instructions [59]. Furthermore, since the branch resolution latency is relatively small, it can be easily hidden by the scheduler.

In addition to structural hazard caused by the unavailability of functional units, contention in the miss status holding registers (MSHRs) can cause additional penalty. `LUD`, in particular, experiences a significant amount of delay by the unavailability of MSHRs. This is because warps in `LUD`, a memory-bandwidth intensive program [19, 55], often request data from the memory in a burst manner. As a result,

the performance of `LUD` is significantly degraded by MSHR contention. Such contention happens to `SR2` and `BFS` as well.

Finally, the *Sync.* component in Figure 3.2 indicates the amount of time warps wait at synchronization barriers (e.g., the ___*syncthread()* API calls) in the kernel. `BT`, `PF`, and `TPF` are three applications whose warps spend a large amount of time synchronizing on barriers. This significant synchronization time is caused by workload imbalance between the warps.

3.3 Warp Criticality

By the latency attribution result, I find that, for some applications, warps often wait at an explicit or implicit synchronization barrier, and the wait time cannot be hidden well. This is caused by a problem—*warp criticality*.

In order to illustrate the warp criticality problem, I take `BFS` from the Rodinia benchmark suite [19] as a motivating example. The CUDA version of `BFS` contains two kernels that are called repeatedly in a loop. Both kernels have a thread-block size of 512 threads that are grouped into 16 warps. These warps are then mapped to the available SMs on the GPU. The first kernel expands the search frontier from the current node to the next node level that contains multiple child nodes. The second kernel performs the actual visit and then sets up the conditions for the next iteration of the first kernel. All threads are synchronized at the end of each kernel. Algorithm 3 illustrates the pseudo code of `BFS`. At the end of two kernels, all warps are synchronized implicitly, without an explicit use of barriers, before all warps can proceed.

To present the concept of warp criticality, I take a particular thread-block in `BFS` as an example. Figure 3.3 shows the amount of idle time the non-critical warps spend to wait for the critical warp to arrive at the end of the first kernel in Thread-Block

Algorithm 3 The GPGPU application—`BFS`.

```
1: function KERNEL_1(node)
2:     ▷ tid: the unique thread ID
3:     if node[tid].mask == True then
4:         node[tid].mask ← False
5:         i ← 0
6:         while i < node[tid].no_of_neighbors do
7:             id ← node[tid].child[i]
8:             if node[tid].visited == False then
9:                 node[id].update ← True
10:            end if
11:        end while
12:    end if
13:    ▷ an implicit barrier here
14: end function
15: function KERNEL_2(node)
16:    if node[tid].update == True then
17:        node[tid].update ← False
18:        node[tid].mask ← True
19:        node[tid].visited ← True
20:    end if
21:    ▷ an implicit barrier here
22: end function
```

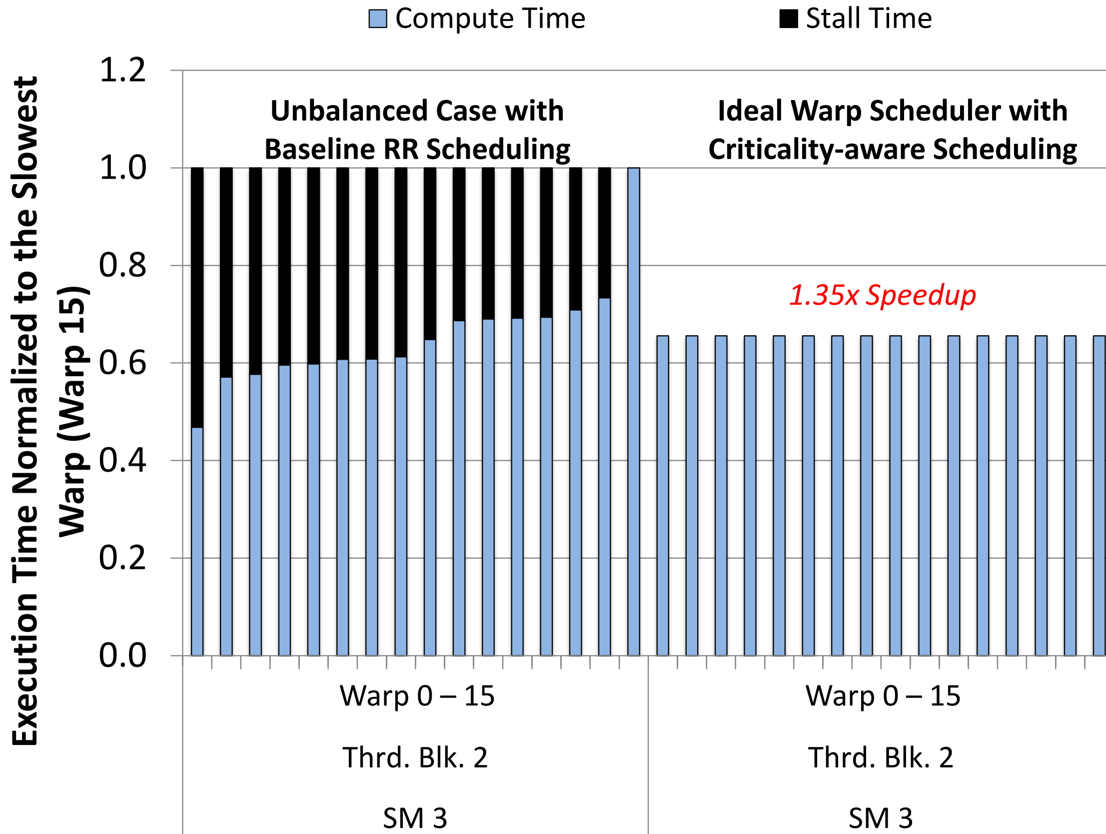

Figure 3.3: An example of warp criticality from the GPGPU application BFS.

2 SM 3 of BFS. The bars plot the compute and idle time normalized to the critical warp, *Warp* 15. These show large warp stall times, with a worst-case stall time of 53% for Warp 0. In contrast, the right set of bars depict an ideal scenario, in which the warp scheduler preferentially selects and executes warps based on their degrees of criticality to equalize the execution time, resulting in a 1.35x speedup.

I further investigate the latency breakdown for all warps in this thread-block that is executed in lockstep. Figure 3.4 compares the latency breakdown of each warp in the thread block. I observe that the critical warp (*Warp 15*) in this particular thread block (Thread-Block 2 SM3) suffers from longer *Scheduling*, *MSHR*, *Data$*, and *Strcl. Hzrd.* latency delays than other warps. As we will see later in this chapter

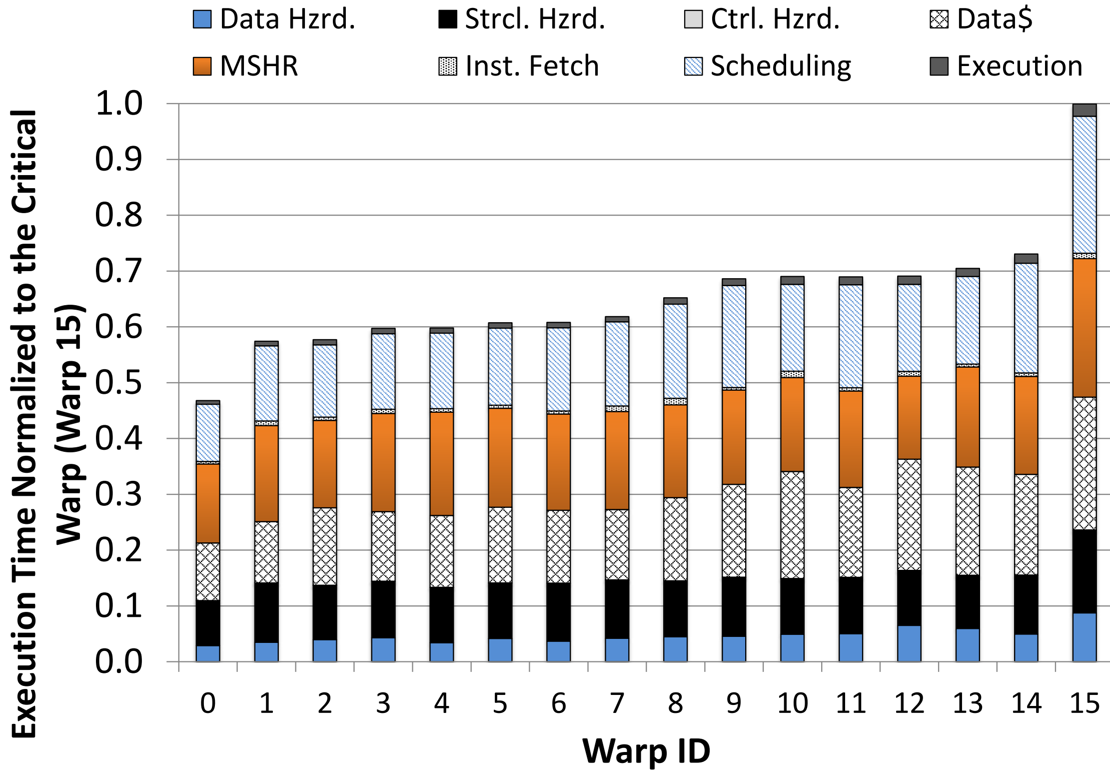

Figure 3.4: Latency breakdown for the BFS application. Warps are sorted by the execution time.

(Chapter 3.4 and 3.5), my proposed criticality-aware scheduling policies can reduce these latencies for the critical warp and result in faster thread block execution time.

In BFS, based on the input dataset, each node has a different number of child nodes as illustrated in Algorithm 3. A warp has to traverse through all neighbors (connected nodes) in the graph. Because the number of child nodes varies in each node level, the amount of work distributed to the warps in the same thread-block varies as well. This causes an imbalanced workload distribution among the warps. The warp which needs to traverse through more child nodes (Line 6 in Algorithm 3) finishes more slowly and becomes the critical warp in the thread-block. In fact, the warp execution time is proportional to the number of iterations of the algorithm. Therefore, I believe that

the number of iterations specified in `BFS` could be a good indicator for predicting the degree of warp criticality for this application (Chapter 4.1).

3.4 Warp Scheduler Design Exploration for Critical Warp Acceleration

3.4.1 CAWS Algorithms

To reduce warp execution time disparity, I explore a family of **C**riticality-**A**ware **W**arp **S**cheduling (**CAWS**) policies and seek to bridge the execution time gap between the parallel warps.

1. **Round robin (RR) scheduling.** As the name suggests, the RR schedules warps in an iterative manner. All warps in the warp pool are treated *equally* regardless of their degree of criticality and are given the same amount of time resource in the baseline GPU.

2. **Thread block based CAWS (CAWS-blk) scheduling.** This scheduling policy aims to improve the execution time of a thread-lock (limited by the execution time of the longest running, critical warp in the thread block) by giving more time resource to the most critical warp in the thread block. By giving higher priority to the most critical warp at the thread block granularity, CAWS-blk allows thread-blocks to finish faster, such that hardware resources become available to other thread-blocks earlier. Consequently, the resource contention is alleviated.

3. **Streaming multiprocessor based CAWS (CAWS-SM) scheduling.** Instead of improving the execution time of the critical warp local to a particular thread-block, I design a global, the CAWS-SM policy that aims to improve the execution time of the critical warp(s) within the same SM but across different

thread blocks. The idea behind this policy is to equalize the execution time of all warps on an SM. CAWS-SM selects the top N critical warps from the warp pool to accelerate, where N is equal to the number of thread blocks on an SM in the design. The SM-based policy is particularly helpful when some of the thread blocks mapped to the same SM do not contain any critical warp and other thread-blocks contain one or more critical warps.

4. **Average based CAWS (CAWS-avg) scheduling.** Instead of giving more time resource to the top critical warp in a particular thread-block (e.g., CAWS-blk) or within a particular SM (e.g., CAWS-SM), I also design a CAWS-avg scheduling policy that identifies a number of critical warps (ranging from none to m) within either a thread block or a SM which require more time resource, where m is determined by the average execution time of all warps scaled by a factor. CAWS-avg evaluates warp criticality by the average execution time of all warps. When the disparity of execution time for warps in an SM is insignificant, giving the slowest warp a higher priority may cause criticality inversion, which occasionally happens in CAWS-SM and CAWS-blk. The advantage of CAWS-avg is to avoid the occurrence of criticality inversion.

3.4.2 CAWS Implementation

To implement CAWS, I design a per-warp priority counter that indicates the degree of warp criticality. The counter value determines when a warp is going to be selected by the scheduler for execution. For example, given there are 48 active warps in the pool in the baseline RR scheduling policy, all warps would initially have counter values from 0 to 47. The warp with a zero counter value is to be scheduled and its counter is reset based on its priority. In this case, the counter for the selected warp is always reset to be 47 since all warps are treated equally and have the same

priority in the baseline policy. In addition to the selected warp, counters for other warp are decremented by one.

Since the value of the priority counter determines how often a warp is scheduled for execution (a smaller value indicates a warp is more important and needs to be scheduled more often), I have to carefully select the counter value for the critical warp(s). For example, to give critical warp(s) 20% more time resource, the scheduler will have to schedule the critical warp(s) 20% more frequently compared to other warps in the pool. In other words, in a pool with 48 warps, the counter value of the critical warps should be reset to 0.8 * 48 and so on.

In the example application, `BFS`, I recognize that the execution disparity between the fastest and the slowest warps is approximately 20% on average. To compare the effectiveness of the CAWS policies explored in this thesis work, I experiment with a counter configuration that guarantees 20% more time resource to the high-priority, critical warp(s) identified in each policy by resetting the value of the priority counters to 40 for critical warps and to 48 for the remaining warps

For the CAWS-avg policy in particular, I divide all warps in the pool into three priority levels. Warps with execution time more than 20% of the average are determined to be critical warps and are given 20% more time resource. This implies that the counter value for these warps are reset to 40 for a pool of 48 warps. Warps with execution time less than 20% of the average are determined to be fast warps and are given 20% less time resource by setting the counter value accordingly. Finally, the remaining warps in the pool will adopt the default counter value as the baseline RR policy.

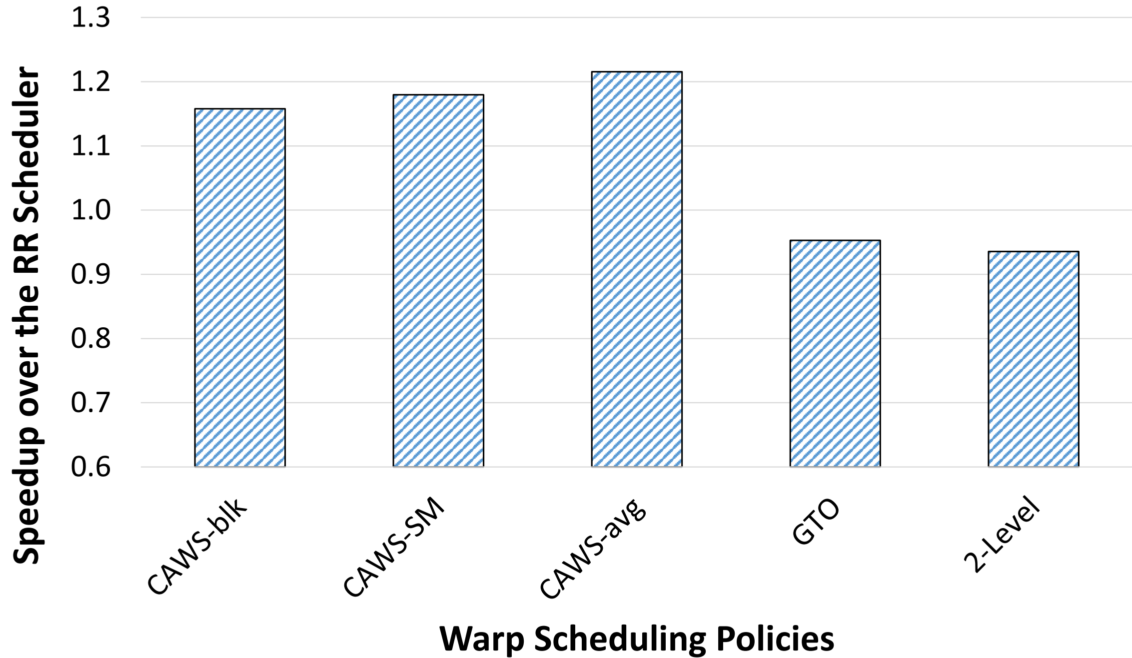

Figure 3.5: The speedup comparison for different warp scheduling policies on `BFS`.

3.5 Evaluation and Analysis

To guide CAWS schedulers to accelerate the critical warp execution, I encode the oracle warp criticality information based on the warp execution time with the baseline RR policy. Furthermore, in order to keep the simulation region consistent and to create a fair comparison metric, in this chapter, I only use the first round of kernel execution to present and analyze the performance of different warp scheduling policies.

Figure 3.5 compares the performance of various CAWS implementations with the baseline RR and the other state-of-the-art warp schedulers, GTO [101] and 2-Level [87]. It reveals that when the oracle knowledge of the critical warps is available, CAWS is able to achieve a 1.21x speedup for `BFS` with the CAWS-avg algorithm,

which outperforms all other warp scheduling algorithms. Moreover, CAWS-blk and CAWS-SM schemes also improves `BFS` performance by 16% and 18% respectively.

The reason CAWS-avg obtains better performance improvement than CAWS-blk and CAWS-SM is because of the criticality inversion. Namely, the critical warp receives more time slices for execution than needed. As a result, a fast running warp become a new critical warp and worsen the performance. Table 3.3 compares the frequency of criticality inversion in each of the three CAWS schemes with respect to the execution time improvement. It clearly indicates that with a higher frequency of criticality inversion occurrence, the amount of performance improvement is less. Therefore, a scheduling policy that does not introduce criticality inversion (CAWS-avg) can gain the performance improvement most.

Next, to illustrate how the CAWS policies can effectively reduce the execution time of the critical warp(s), I compare the execution time of all warps in the particular thread block (Thread-Block 2, SM 3) used previously in Figure 3.4. Figure 3.6 shows the execution time and the latency breakdown for all the warps in the thread block under the oracle, CAWS-avg scheduling policy. The execution time of the critical warp (*Warp 15*) is reduced by 28.4% when compared to using the baseline RR scheduling policy. This execution time improvement comes primarily from the reduction in the scheduling delay (*Scheduling*), the contention in the MSHR entries (*MSHR*), and the contention in the functional units (*Strcl. Hzrd*). These latency components are exactly the ones that the CAWS policy intends to improve. When comparing to the warp latency breakdown under the baseline RR scheduling policy in Figure 3.4, the *Data$* latency component is increased under the oracle, CAWS-avg policy. This is because the *Data$* stall cycles of the critical warp are better hidden by the RR scheduling policy. Overall, by applying the oracle, CAWS-avg policy, the execution time of this particular thread block is improved by 27%.

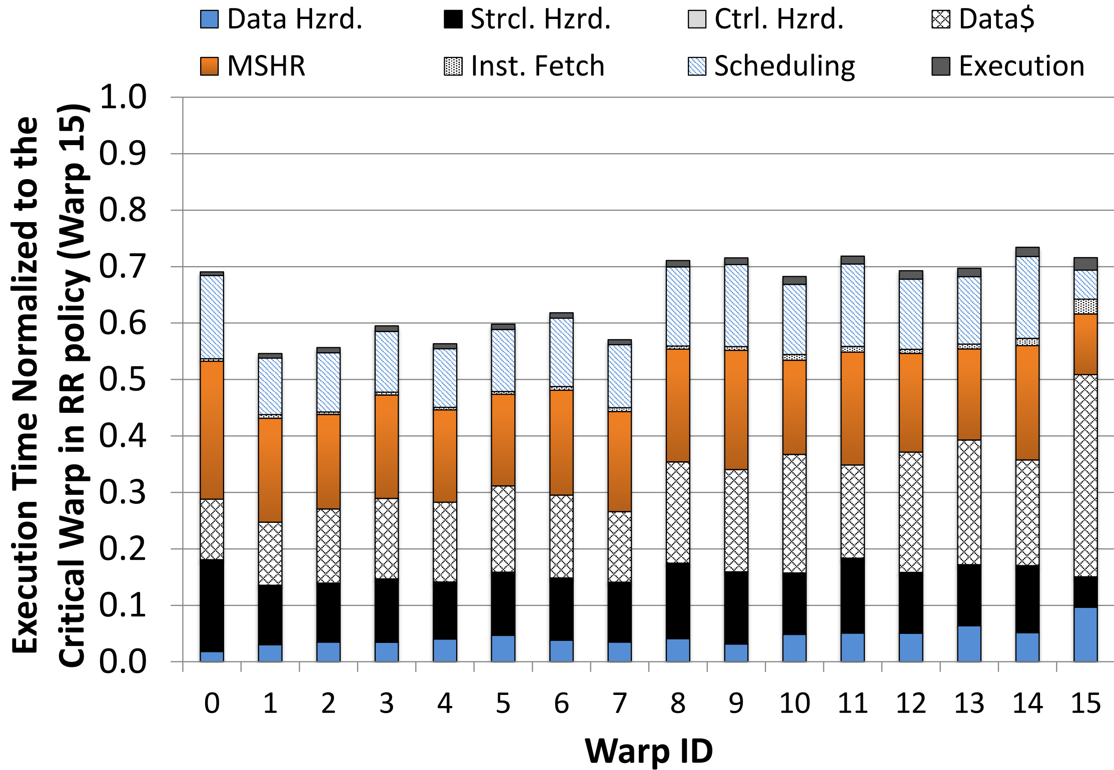

Figure 3.6: Latency breakdown for the BFS application under the oracle CAWS-avg scheduling policy. Warps are sorted by the execution time under RR policy.

3.6 Chapter Summary

This chapter presents a detailed characterization and evaluation for the latency-hiding capability of modern GPU architectures, highlighting the different factors that comprise the execution latency in the GPU pipeline, across a wide range of GPGPU applications. I find that the fast context-switching and massive multithreading architecture can effectively hide much of the latency [69]. However, for certain GPGPU applications such as BFS, the overall performance is limited by the critical warps. To address such a performance issue, I design a family of criticality-ware warp schedul-

ing policies that aim to equalize the execution of all warps to maximize the hardware resource utilization and minimize the application execution time [68].

Table 3.2: Benchmarks for GPGPU latency hiding ability characterization. *M* and *C* stand for memory- and computation-intensive respectively.

Abbrev.	Application	Dataset	Category [20, 55]
`BT`	B+Tree [19]	M nodes	*M*
`BP`	Back Propgation [19]	65536 nodes	*M*
`BFS`	Breadth First Search [19]	65536 nodes	*M*
`GAU`	Gaussian [19]	1024x1024 matrix	*C*
`HTW`	Heartwall [20]	656x744 gray scale AVI	*C*
`HOT`	Hotspot	512x512 nodes [19]	*C*
`KMN`	K-Means [19]	494020 nodes	*M*
`LMD`	LavaMD [19]	10 nodes	*C*
`LEU`	Leukocyte [19]	640x480 gray scale AVI	*C*
`LUD`	LUD Decomposition [20]	2048x2048 matrix	*C*
`MYC`	Myocyte [19]	100 nodes	*C*
`NW`	Needleman-Wunsh [19]	1024x1024 nodes	*M*
`PF`	Particle Filter [19]	128x128x10 nodes	*C*
`PTH`	Pathfinder [19]	100000 nodes	*C*
`SR1`	SRAD1 [19]	502x458 nodes	*C*
`SR2`	SRAD2 [19]	2048x2048 nodes	*M*
`SC_small`	Streamcluster (small) [19]	32x4096 nodes	*M*
`SC_mid`	Streamcluster (mid) [19]	64x8192 nodes	*M*
`TPF`	Two-Point Angular [107]	487x100 nodes	*M*

Table 3.3: The speedup and frequency of criticality inversion within a thread-block for `BFS`.

Policy	Speedup	Criticality Inversion
CAWS-blk	1.16	8.89%
CAWS-SM	1.18	2.22%
CAWS-avg	1.21	0%

Chapter 4

COORDINATED CRITICALITY-AWARE WARP ACCELERATION

Chapter 3 showed that GPU's performance is constrained by the warp criticality problem. I demonstrated a criticality-aware warp scheduling design, CAWS, which is able to improve GPU's performance by accelerating the critical warp execution and by reducing the warp execution time disparity. However, CAWS heavily relies on the oracle criticality knowledge to guide the critical warp acceleration. In reality, it is difficult to obtain the oracle criticality information in advance because, for many applications, the workload distribution is dependent on the input data. Thus, it cannot be statically known.

To design a more effective and practical solution for accelerating critical warp execution, in this chapter, I first identify the root-causes of the warp execution time disparity. I then propose a coordinated solution, **C**riticality-**A**ware **W**arp **A**cceleration (**CAWA**) which accurately predicts the critical warps and efficiently manages computation and memory resources to accelerate the critical warp execution dynamically.

4.1 Source of Execution Time Disparity

I observe that the significant execution time disparity between parallel warps can be caused by four major factors:

1. workload imbalance,
2. diverging branch behavior,
3. contention in the memory subsystem, and

Algorithm 4 `BFS` searching algorithm.

```
1:  function BFS(w)                                  ▷ w: node (thread/warp)
2:      while notYetVisitedAllNeighbors do
3:          n ← nextNode
4:          if n.hasNotBeenVisited then
5:              ▷ This is a child node
6:              n.Cost ← w.Cost
7:              n.hasNotBeenVisited ← False
8:              w.nChild ← w.nChild + 1
9:          else
10:             ▷ This is a non-Child node
11:             w.nNonChild ← w.nNonChild + 1
12:         end if
13:     end while
14:     ▷ kernel exit point/implicit barrier
15: end function
```

4. warp scheduling order.

I use `BFS` from the Rodinia benchmark suite [19] as an example application to illustrate the degree of warp criticality contributed by each factor.

4.1.1 Workload Imbalance

In a GPGPU kernel function, tasks are not always uniformly distributed to each thread. Therefore, some threads have heavier workloads than others, leading to the scenario where some warps have heavier workloads than other warps. Warps with

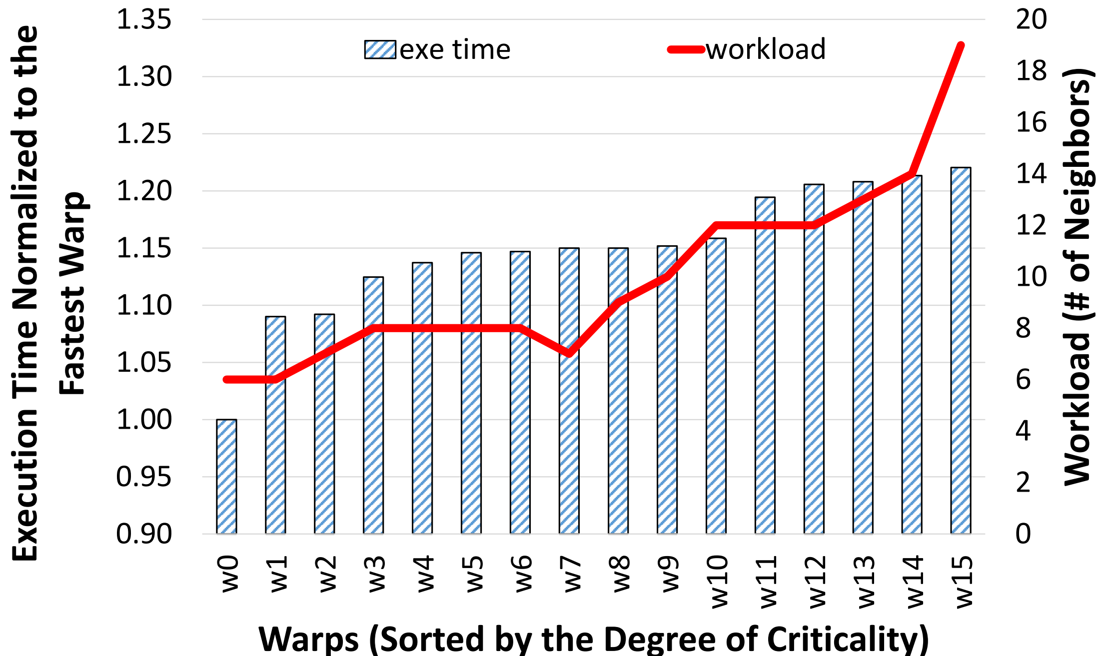

Figure 4.1: Warp execution time disparity caused by workload imbalance (imbalance number of neighbors) for BFS.

heavier workloads require longer time to process their tasks. Consequently, warps with heavier workloads often become slower running warps or the *critical warps*.

In BFS, workload imbalance comes from building and traversing an unbalanced tree-like data structure. Each node has to traverse all of its neighboring nodes to build a tree (Line 2 in Algorithm 4). Depending on the data inputs, the number of child nodes of a particular node mapped to a warp could vary, resulting in different per-warp workloads. Figure 4.1 shows the per-warp execution time for all warps in a particular thread-block (Thread-Block 2 on SM 3) in BFS. The warps are sorted based on the warp execution time and the per-warp workload (number of neighboring nodes). The execution time disparity between the fastest and the slowest running warps is approximately 20% of the fastest warp's execution time, leading to a significant waiting time for the fastest running warp.

4.1.2 Diverging Branch Behavior

In addition to the unbalanced workload scenario for parallel warps in a thread-block, diverging branch behavior could also cause varying execution time for warps.

At runtime, warps can undergo different control paths leading to different number of dynamic instructions across different warps. This problem could be worsened if threads in a warp also take diverging control paths, i.e., the branch divergence problem, leading to a larger instruction execution gap between warps. Prior studies [32, 33, 84, 99] showed that the branch divergence problem can significantly degrade the performance of GPGPU applications.

To remove the workload imbalance effect and focus on the impact of the diverging branch behavior, I modify the data input provided to `BFS` to represent a balanced tree. Figure 4.2 shows the warp execution time for the same thread-block with a balanced workload. Although the computation workload is equally distributed across warps, I can still observe varying warp execution time. The execution time difference between the fastest and the slowest warps is significant 40%. This is because while a node traverses through its neighbors in the input graph, only the data represented child nodes (nodes that have not yet been visited) need to be processed. Visiting child and non-child nodes (nodes that have been visited before) fall onto different if-else blocks (Line 4–12 in Algorithm 4). This introduces a varying number of per-warp dynamic instruction counts, since some warps will execute the taken path while others execute the not-taken path.

In the worst-case scenario, when thread-level branch divergence occurs [84, 99], instructions in both the taken and not-taken paths need to be executed in some warps, resulting in a higher number of dynamic instruction executed, while other warps only have to execute one of the two paths. The dynamic instruction count

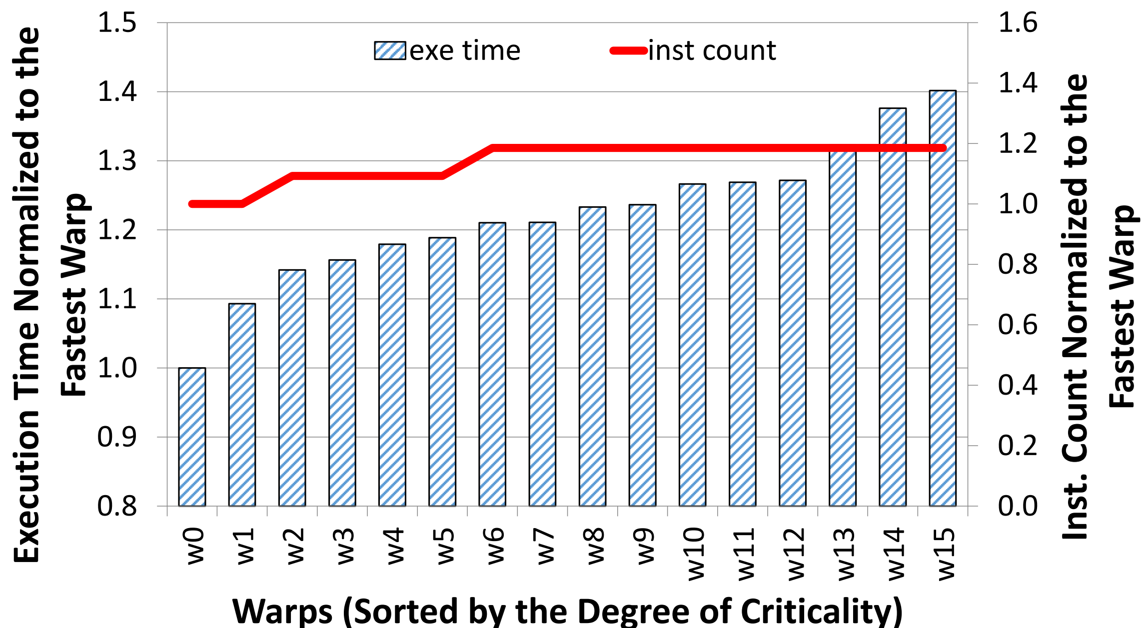

Figure 4.2: Warp execution time disparity caused by diverging branch behavior for `BFS`.

disparity between warps could be as high as 20% (number of instructions) in `BFS` as illustrated by the red curve in Figure 4.2.

4.1.3 Contention in the Memory Subsystem

Hardware resource contention, particularly in the memory subsystem, can exacerbate the warp criticality problem. Jog et al. [54, 55] observed that the memory subsystem has a significant impact on GPGPU applications. Poor data alignment and warp scheduling design can introduce extra stall cycles which significantly reduce the performance of GPUs. Jia et al. [51, 52] also pointed out that interference in the L1 data cache as well as in the interconnect between the L1 data caches and the L2 cache are the major factors that limit GPU performance. This is because the data

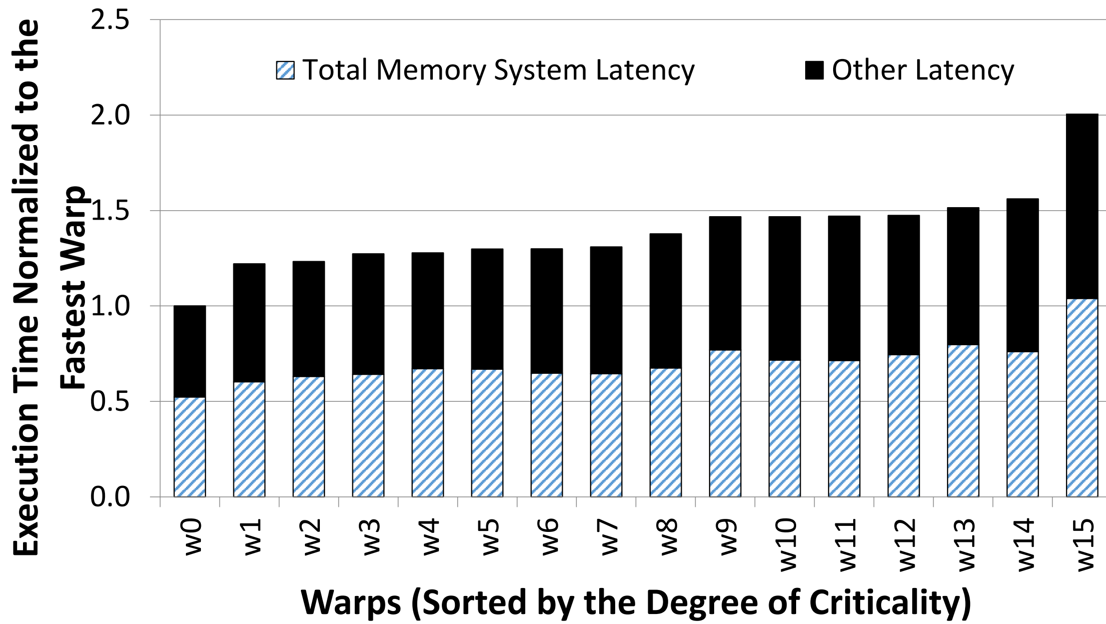

Figure 4.3: Warp execution time disparity caused by memory subsystem delay for `BFS`.

cache capacity and the memory bandwidth are scarce resources servicing the memory demand of the massively parallel GPUs.

The cache interference, particularly in the L1 data caches, could worsen the warp criticality problem. Figure 4.3 shows the portion of a warp's execution time caused by delays in the memory subsystem. I observe that the slower-running warps often experience higher memory access latencies. Figure 4.4 shows the reuse distance analysis of critical warp cache lines [1] . More than 60% of the cache blocks that could be reused by the slower-running, critical warps are evicted before the re-references by the critical warps. This is caused by the interference between critical and non-critical cache blocks in the L1 data cache.

[1]Data is based on an L1 data cache of 16KB, 4-way set-associative, 128B cache block.

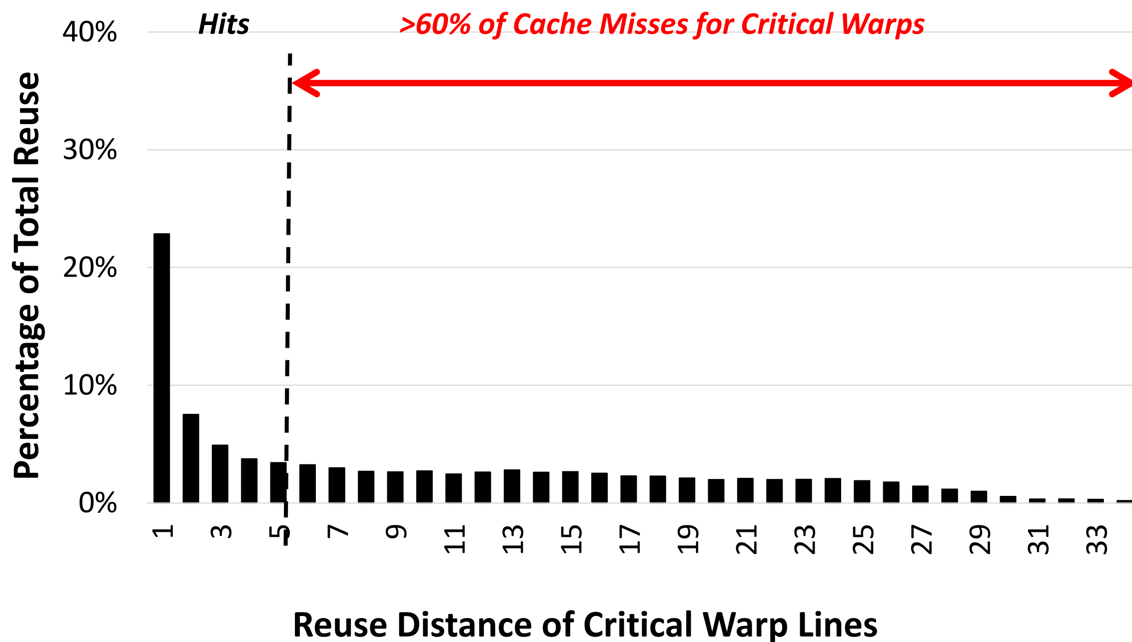

Figure 4.4: L1 data cache reuse distance for the critical warps in BFS.

4.1.4 Latency Introduced by the Warp Scheduler

The execution of warps can experience additional delay in the warp scheduler. Because of the particular warp execution order determined by the scheduler, when a warp becomes ready for execution, it can experience up to N cycles of scheduling delay, where N represents the number of warps. State-of-the-art warp scheduling policies are criticality-oblivious and introduce additional delay that could further degrade the performance of critical warps. As Figure 4.5 shows, scheduling policies such as the baseline round robin (RR) scheduler, can contribute as much as 52.4% additional wait time for the critical warp.

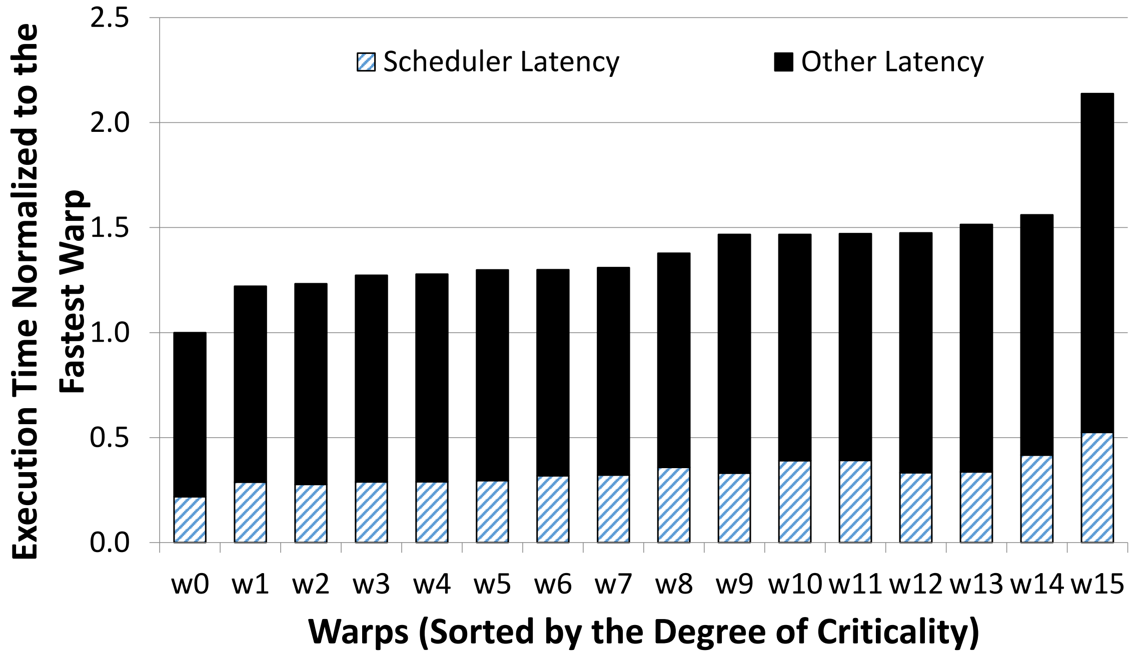

Figure 4.5: Warp execution time disparity caused by warp scheduling delay for `BFS`.

4.2 Coordinated Criticality-Aware Warp Acceleration Design

In order to mitigate the execution time disparity between warps, I design a coordinated warp scheduling and cache prioritization scheme, **C**riticality-**A**ware **W**arp **A**cceleration (**CAWA**), based on the observations made in Chapter 4.1. CAWA consists of three major components:

1. a dynamic warp criticality prediction logic (CPL),
2. a greedy criticality-aware warp scheduler (gCAWS), and
3. a criticality-aware cache prioritization technique (CACP)

Figure 4.6 illustrates a modern GPU pipeline with the newly-proposed components in CAWA highlighted in the orange box.

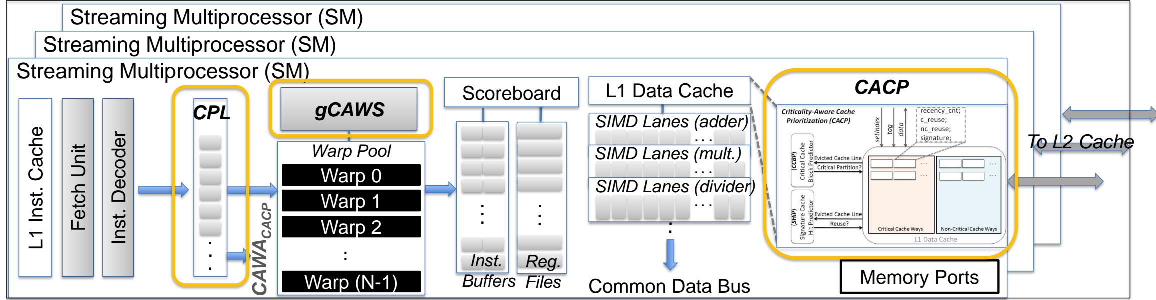

Figure 4.6: The CAWA architecture.

To accelerate the critical warp execution, CPL is designed to identify slower-running warps that have a high likelihood to become critical warps at runtime. gCAWS then allocates more computation resources to the predicted-to-be-critical warps with higher scheduling priorities. This alleviates the dynamic workload imbalance-caused warp criticality as well as the additional scheduling delay imposed onto the critical warps.

In addition to reducing critical warp execution time by allocating more computation resources, CACP also proactively reserves a certain amount of cache capacity for data that is useful to critical warps (CACP). By doing so, CAWA ensures a certain degree of performance guarantee for the critical warps by reducing the amount of long latency cache misses experienced by the critical warps. Next, I present the three major components in CAWA in detail.

4.2.1 *Critical Warp Identification with Criticality Prediction Logic*

To identify critical warps at runtime, I develop a **C**riticality **P**rediction **L**ogic (**CPL**) to monitor the execution progress of individual warps in the scheduler's pool by implementing a criticality counter per warp. The per-warp criticality counter represents the execution progress of each warp and is updated based on (1) the degree of instruction count disparity caused by diverging branch behavior, and (2) stall

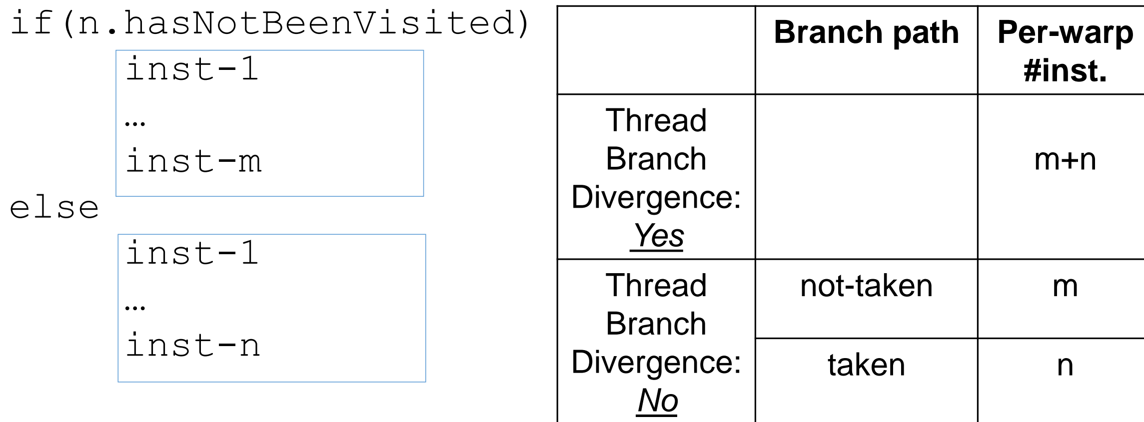

Figure 4.7: The instruction count disparity caused by branches.

latencies caused by shared resource contention. The per-warp criticality counters are used by the proposed gCAWS scheduler and CACP cache prioritization scheme for critical warp acceleration.

To consider the influence of workload imbalance and diverging branch behavior on warp criticality, CPL is designed to update per-warp criticality counters based on the number of instructions in the executed branch path. Figure 4.7 shows an example to highlight the possibility of diverging dynamic instruction counts per warp based on the branch path behavior. Warps that experience thread-level branch divergence will have to execute m+n number of instructions while other warps will execute either m or n instructions based on the branch outcome. Depending on the values of m and n, even without branch divergence, warps could face a significantly different amount of instructions for execution. This could translate to diverging warp execution time.

Based on the branch outcome, CPL updates the per-warp criticality counter accordingly with the inferred size of the basic block determined with the current branch instruction pointer (*currPC*) and the target instruction pointer (*nextPC*). By doing so, CPL would increment or decrement the per-warp criticality counter. In addition

Algorithm 5 An example of the instruction-based CPL.

1: $[PC_1]$ $\$L0:$ $@p0\ bra\ \$L2$ ▷ jump to PC_4 if p0 is true (ΔInst = PC_4 - PC_1 + 1)

2: $[PC_2]$ $\$L1:$ $add.u64\ \%r1,\ \%r1,\ \%1$ ▷ jump to PC_5 (ΔInst = PC_5 - PC_1 + 1)

3: $[PC_3]$ $bra\ \$L3$

4: $[PC_4]$ $\$L2:$ $sub.u64\ \%r1,\ \%r1,\ \%1$

5: $[PC_5]$ $\$L3:$ $mov.u64\ \%r2,\ \%r1,\ \%r2$

to the branch outcomes, CPL also decrements the criticality counter whenever an instruction is committed in order to balance the execution progress.

Algorithm 5 demonstrates an example of how the instruction-based CPL works. If branch divergence occurs at PC_1 for a particular warp, the warp has to run through all three instructions (PC_2, PC_3, and PC_4). On the other hand, there is no branch divergence and depending on the branch outcome, the warp will execute either one (PC_4) or two (PC_2 and PC_3) instructions. After executing PC_1, the target address become available and is used to calculate the additional dynamic instructions per-warp by CPL.

In addition to updating the per-warp criticality counters based on dynamic execution progress, CPL also records additional stall latencies experienced due to shared resource contention as well as scheduler delays, while updating the criticality counter. CPL monitors the stall cycles between the current and the next instruction execution for each warp and increment the criticality counter accordingly for all warps. Algorithm 6 presents the criticality counter update mechanism based on stall cycles in CPL, where the *stallCycle* represents the total stall cycles between executing two consecutive instructions.

Algorithm 6 The criticality counter with stall cycles.

```
1: function WarpScheduler                     ▷ select a ready warp to execute
2:     while NotVisitedAllWarps do
3:         ▷ select warps in the order based on the scheduling policy
4:         w ← findNextWarp()
5:         if w.isRready() then
6:             ▷ stallCycles = total stall time between two consecutive instructions
7:             w.nStall ← w.nStall + stallCycles
8:             InstructionExecute(w)
9:             break
10:        end if
11:    end while
12: end function
```

Overall, the per-warp criticality counter is updated as follows:

$$nCriticality = nInst \times w.CPI_{avg} + nStall; \tag{4.1}$$

where *nCriticality* represents the value of the per-warp criticality counter, *nInst* represents the relative instruction count disparity between the parallel warps, $w.CPI_{avg}$ represents the per-warp average CPI, and *nStall* is the stall cycles incurred by shared resource contention and the scheduler.

4.2.2 greedy Criticality-Aware Warp Scheduler

With the critical warp identified by CPL, the **g**reedy **C**riticality-**A**ware **W**arp **S**cheduler (**gCAWS**) is designed to give more computation resources to critical warps by prioritizing the execution of critical warps over other warps and by providing a larger time slice to warps in a greedy manner. gCAWS incorporates the strengths

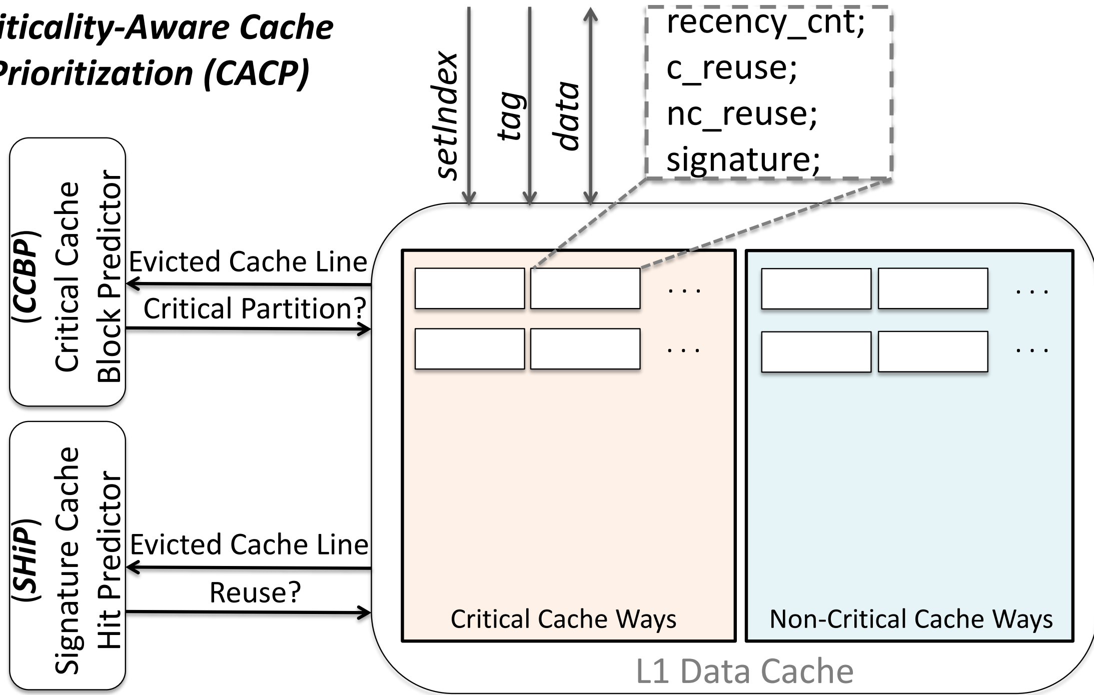

Figure 4.8: The criticality-aware cache prioritization scheme.

of the Greedy-Then-Oldest (GTO) [101] and CAWS warp schedulers (Chapter 3). At each cycle, gCAWS selects a ready warp for execution based on the degree of warp criticality determined by the per-warp criticality counter in CPL. If there are multiple warps having the same criticality, the warp scheduler will select the oldest one based on the GTO algorithm. Then gCAWS greedily executes instructions from the selected critical warp until this particular warp has no further available instructions. Consequently, the critical warp not only receives a higher scheduling priority but also benefits from a larger time slice.

4.2.3 Criticality-Aware Cache Prioritization

In addition to allowing the critical warps to access pipeline resources more often and for a longer time duration, **C**riticality-**A**ware **C**ache **P**rioritization (**CACP**) is designed to allocate a certain fixed amount of the L1 data cache capacity to data that

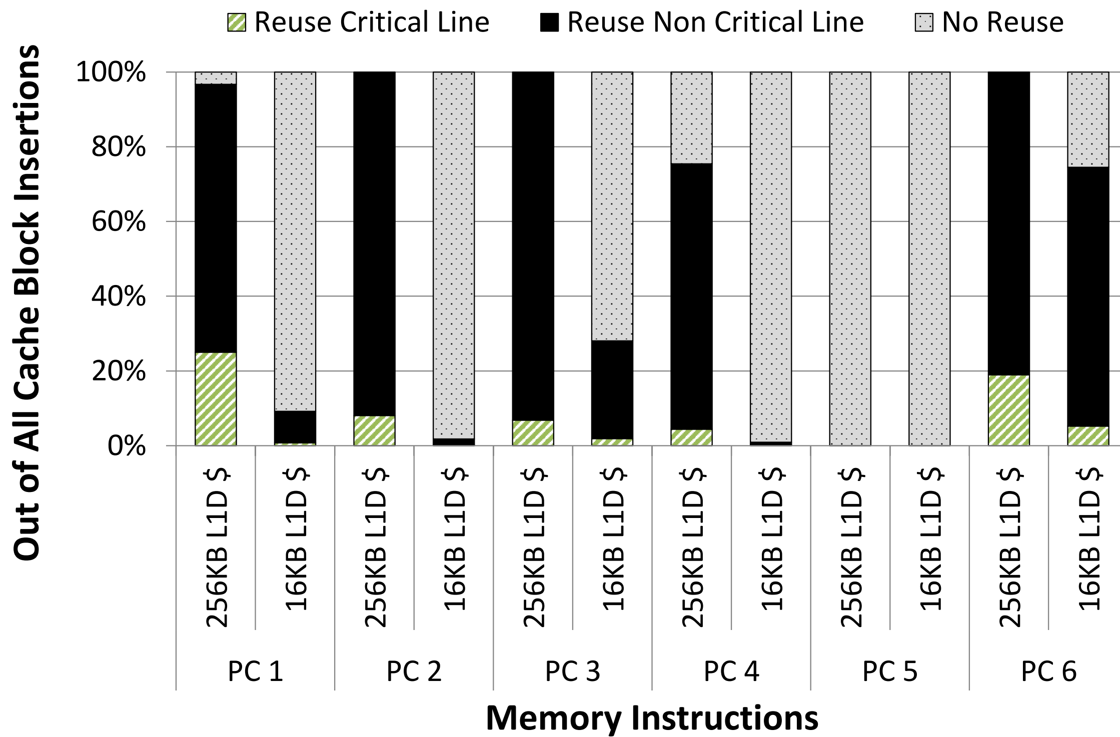

Figure 4.9: Reuse behavior of different PCs for `BFS`.

will be used by the critical warps for performance guarantee. The insight that CACP built upon is that not all cache lines have equal importance—Data that will be used by critical warps is latency-critical and should be treated with higher priority at the L1 data cache. To do so, among all incoming cache lines, CACP first predicts critical cache lines (cache lines that will be used by critical warps) with the critical cache block predictor, and retains these critical cache lines in the cache partition reserved for critical warps. Figure 4.8 shows the proposed CACP.

CACP partitions the L1 data cache into two parts in the granularity of ways: critical cache ways and non-critical cache ways. The number of ways dedicated to critical versus non-critical cache blocks are determined through experimental analysis

of various benchmarks. Through sensitivity analysis, CACP achieves the best overall performance when 8 out of 16 ways are dedicated to critical cache blocks.

Based on the unique characteristics of GPGPU—relatively small instruction footprint but diverse reuse behavior—I design a **C**ritical **C**ache **B**lock **P**redictor (**CCBP**) to differentiate critical cache lines from non-critical cache lines such that an incoming cache block will either be inserted in the cache partition reserved for critical or non-critical cache lines. Figure 4.9 shows that there are six memory instructions in the application `BFS`, which all warps execute. The left bar within each memory instruction represents the reuse pattern of critical cache blocks in a 256KB data cache whereas the right bar represents the reuse pattern of critical cache blocks in the baseline 16KB data cache. First, when the cache is large enough, cached blocks have a high likelihood of reuse by both critical or non-critical warps. However, the L1 data cache in GPUs are often too small to accommodate the active working set of the entire application. The second observation is that the reuse patterns for the various memory instructions are different. For instance, the majority of the cache blocks brought by *PC-5* never receive any reuse before being evicted from the cache. With the two key observations, CCBP is designed to learn the reuse patterns for cache blocks based on the insertion instructions and predict which cache blocks will be reused by critical warps at runtime.

CCBP is built upon the idea of the signature-based cache hit predictor (SHiP) that was originally proposed for the last-level CMP cache [119]. CCBP learns whether an incoming cache line will be used by critical warps or not based on a signature. I design the signature for CCBP to be a combination of instruction program counters (PCs) and memory address regions as the two pieces of information have been shown to be useful in learning and correlating cache block reuse patterns [58, 61, 63, 119]. A signature is formed by XOR-ing the lower 8 bits of an instruction PC and the

Algorithm 7 The pseudo code of the cache miss pperation for CACP.

```
1:  function atMiss(req)
2:      ▷ select partition and insert cache line
3:      line.signature ← (req.pc XOR req.addr)
4:      if CCBP[line.signature] > Threshold then
5:          ▷ predict to be critical line
6:          L1D.CriticalPartition.insert(line)
7:          L1D.CriticalPartition.setPosition(line)      ▷ set SHiP insertion position
8:      else
9:          ▷ predict to be non-critical line
10:         L1D.NonCriticalPartition.insert(line)
11:         L1D.NonCriticalPartition.setPosition(line)      ▷ set SHiP insertion
    position
12:     end if
13: end function
```

lower 8 bits of the memory address, and is used to index into the CCBP which is a simple array of 2-bit saturating counters. The operations of CCBP is outlined in Algorithm 7, 8, and 9 in detail.

In addition to CCBP, CACP includes an additional signature-based cache hit predictor (SHiP) that learns and predicts the reuse pattern of any incoming cache blocks based on the same signature used for CCBP. The outcome of SHiP is used to guide the insertion position of the cache block. For example, if a 2-bit re-reference interval prediction (RRIP) replacement policy is used [49], the outcome of SHiP will guide a cache block insertion position to be in the *long* (re-reference prediction value = 2) versus in the *distance* (re-reference prediction value = 3) re-reference prediction.

Algorithm 8 The pseudo code of the cache hit operation for CACP.

```
1:  function ATHIT(req)
2:      ▷ set RRIP promotion position
3:      if InCriticalPartition(line) then
4:          L1D.CriticalPartition.setPromotion(line)
5:      else
6:          L1D.NonCriticalPartition.setPromotion(line)
7:      end if
8:      if IsCriticalWarp(req.WarpID) then
9:          ▷ correct prediction
10:         line.c_reuse ← TRUE
11:         CCBP[line.signature] + +
12:         SHiP[line.signature] + +
13:     else
14:         ▷ hit is from non-critical warp
15:         line.nc_reuse ← TRUE
16:         SHiP[line.signature] + +
17:     end if
18: end function
```

CCBP plays the role of identifying cache lines that will be reused by critical warps and inserts these cache lines in the critical partition of the cache. To maximize hits, SHiP is used so that only the cache lines that receive reuse are retained in the cache. With the help of CCBP and SHiP, CACP is able to capture both cache lines that have intra-warp and inter-warp localities. Furthermore, CACP complements the gCAWS warp scheduler. While gCAWS prioritizes the critical warp execution, CACP reserves larger cache space to the critical warps to further reduce the execution latency.

Algorithm 9 The pseudo code of the cache line eviction operation for CACP.

```
1: function atEvict(line)
2:     if (line.c_reuse == FALSE) AND (line.nc_reuse == TRUE) AND (partition == CriticalPartition) then
3:         ▷ incorrect prediction
4:         CCBP[line.signature] − −
5:     else if (line.c_reuse == FALSE) AND (line.nc_reuse == FALSE) then
6:         ▷ no reuse
7:         SHiP[line.signature] − −
8:     end if
9: end function
```

Thus, CAWA is able to take the coordinated approach of criticality prediction, warp scheduling, as well as cache prioritization to provide the best performance speedup.

4.3 Evaluation and Analysis

4.3.1 Experimental Environment and Methodology

To evaluate the CAWA design, I use GPGPU-sim simulator version 3.2.0 [11] to explore the behavior of GPGPU applications. I run GPGPU-sim with the default configuration mimicking the NVIDIA Fermi GTX480 architecture and configure the per-SM L1 data cache as 16-way set-associative. Table 4.1 describes the simulation configuration and parameters used for the design evaluation for CAWA.

I select GPGPU applications from the Rodinia [19, 20] and Parboil [107] benchmark suites to evaluate the performance improvement of the coordinated CAWA design in GPUs. Table 4.2 lists the details of the benchmarks and their datasets used to evaluate the CAWA design. Since CAWA mainly aims to improve the performance

Table 4.1: GPGPU-sim simulation configurations for CAWA.

Architecture	*NVIDIA Fermi GTX480*
Num. of SMs	*15*
Max. Num. of Warps per SM	*48*
Max. Num. of Blocks per SM	*8*
Num. of Schedulers per SM	*2*
Num. of Registers per SM	*32768*
Shared Memory	*48KB*
L1 Data Cache	*16KB per SM (8-sets/16-ways)*
L1 Instruction Cache	*2KB per SM (4-sets/4-ways)*
L2 Cache	*768KB shared cache (64-sets/16-ways/6-banks)*
Min. L2 Access Latency	*120 cycles*
Min. DRAM Access Latency	*220 cycles*
Warp Size (SIMD Width)	*32 threads*

of those applications with irregular execution behavior as well as cache utilization, I categorize these benchmarks into two groups based on their execution time disparity and sensitivity to L1 data cache performance as sensitive (*Sens*) or non-sensitive (*Non-sens*).

4.3.2 Performance Overview

Overall, CAWA improves GPGPU performance by an average of 23% compared to the baseline RR warp scheduler for *Sens* applications and by an average of 9.2% over all applications, as Figure 4.10 shows. In particular, CAWA speeds up the performance of `KMN`, which suffers from severe cache thrashing, the most, by a significant

Table 4.2: Benchmarks for CAWA evaluation.

Abbrev.	Application	Dataset	Category
BFS	Breadth First Search [19]	65536 nodes	Sensitive (Sens)
BT	B+Tree [19]	1 million nodes	
HTW	Heartwall [20]	656x744 gray scale AVI	
KMN	K-Means [19]	494020 nodes	
NW	Needleman-Wunsh [19]	1024x1024 nodes	
SR1	SRAD_1 [19]	502x458 nodes	
SC_small	Streamcluster (small) [19]	32x4096 nodes	
BP	Back Propagation [19]	65536 nodes	Non-sensitive (Non-sens)
PF	Particle Filter [19]	128x128x10 nodes	
PTH	Pathfinder [19]	100000 nodes	
SC_mid	Streamcluster (mid) [19]	64x8192 nodes	
TPF	Two-Point Angular [107]	487x100 nodes	

3.13x over the baseline. I also compare the performance of CAWA with two other state-of-the-art warp schedulers, i.e., 2-Level [87] and GTO. Across the seven *Sens* applications, CAWA improves the performance the most by an average of 23% while 2-Level and GTO improves the performance by -2% and 16% respectively.

For memory intensive GPGPU applications such as `KMN`, performance is mainly restricted by the efficiency of the data caches. Because of the large amount of data which is streamed over the relatively small L1 data caches, the cached data is evicted from the L1 caches before it receives any additional reuses. GTO alleviates the cache thrashing problem by limiting the number of warps that could be active such that the active working set is kept small and the intra-warp data locality can be better captured in the L1 data cache. The significant performance improvement from CAWA

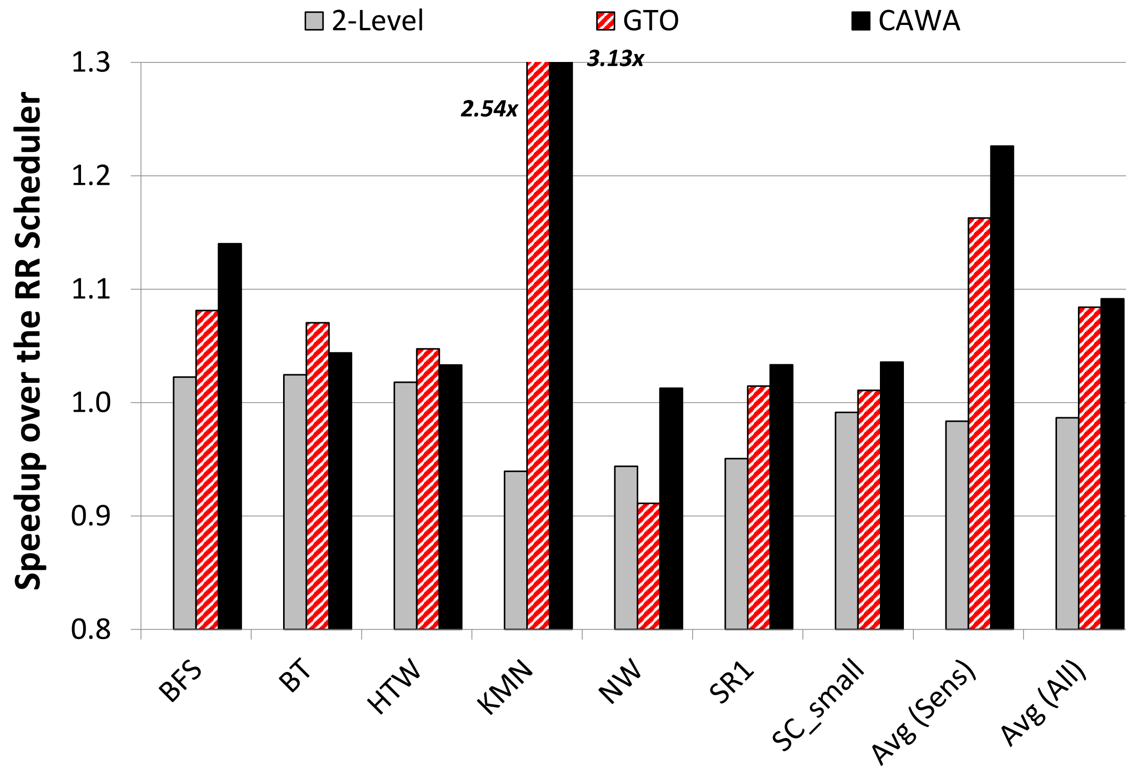

Figure 4.10: Performance improvement of CAWA.

is achieved for a similar reason. gCAWS is able to limit the number of active warps such that the aggregate working set can well fit into the L1 data cache. Furthermore, CACP reserves a fixed cache capacity for data useful to the critical warps thereby reducing the degree of interference between critical and non-critical cache blocks.

Figure 4.11 shows the number of misses per kilo instructions (MPKI) reduction for the GPGPU workloads under 2-Level, GTO, and CAWA respectively. Overall, CAWA reduces the L1 cache MPKI the most when compared to the other two schemes. For memory intensive applications such as `KMN`, CAWA significantly reduces the cache miss rate by 26.2%. For other applications such as `HTW` and `SC_small`, MPKI is instead increased under CAWA. This is because CACP prioritizes critical cache blocks over non-critical cache blocks over the baseline cache replacement policy that is designed

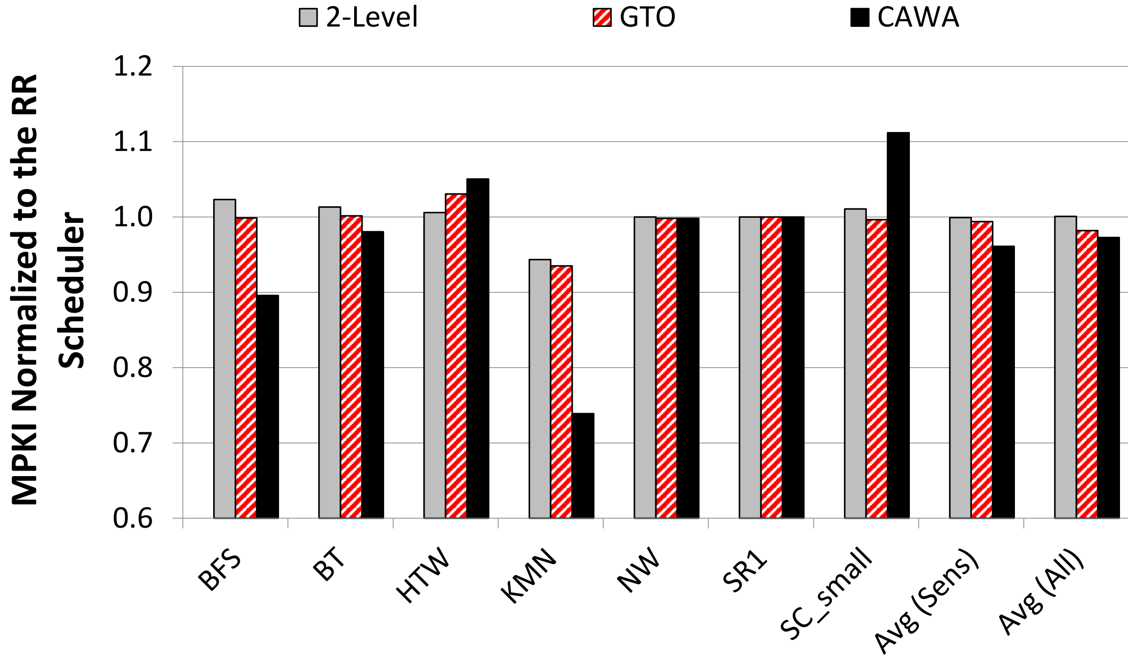

Figure 4.11: L1 data cache MPKI reduction of CAWA.

to minimize cache misses. Although MPKI is increased for `HTW` and `SC_small`, the corresponding speedup is improved by 3.3% and 3.6% respectively. This is because CAWA trades off cache blocks that may be used more often with cache blocks that are critical.

4.3.3 Performance Analysis for CPL

The critical warp prediction mechanism is a vital component in CAWA and is used to guide compute and memory resource prioritization. To evaluate the accuracy of CPL, I compare the periodic prediction outcomes with the slowest, critical warp based on its total execution time. To calculate the prediction accuracy, I define that if a warp's criticality is larger than 50% of warps in a thread-block, this warp is a slow warp. Since warp criticality could change at runtime which is not captured by the static warp execution time analysis, it is difficult to calculate the prediction

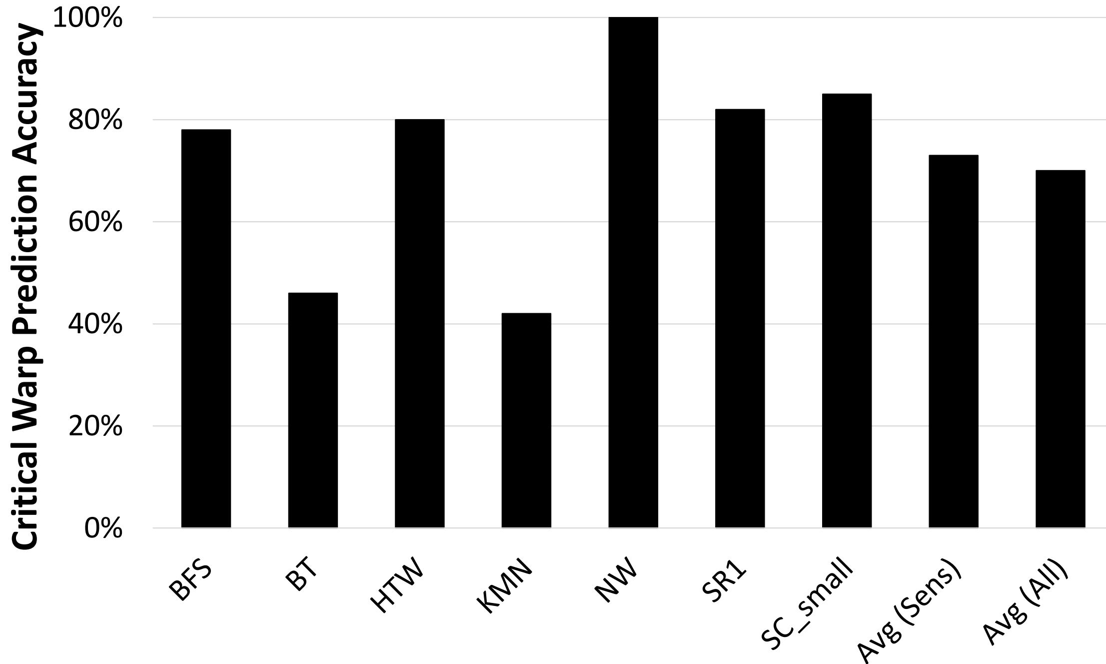

Figure 4.12: The prediction accuracy of CPL.

accuracy. Thus, I count the prediction accuracy as the frequency that the critical warp is identified as a slow warp. Figure 4.12 shows the warp criticality prediction accuracy comparison. On average, CPL can accurately identify critical warps as a slow warp with a prediction accuracy of 73% [2] . Since CPL learns the sources and degree of delay dynamically and reflects the delay and execution progress in the per-warp criticality counters, CPL is able to adjust its warp criticality prediction outcomes at runtime.

4.3.4 Performance Analysis for gCAWS

To understand how gCAWS help CAWA improve GPU performance, I compare the performance improvement of gCAWS with CAWS. Figure 4.13 shows the perfor-

[2] CPL results in an 100% prediction accuracy for `NW` because it is an application which lacks warp-level parallelism, i.e., a thread-block has only one or two warps

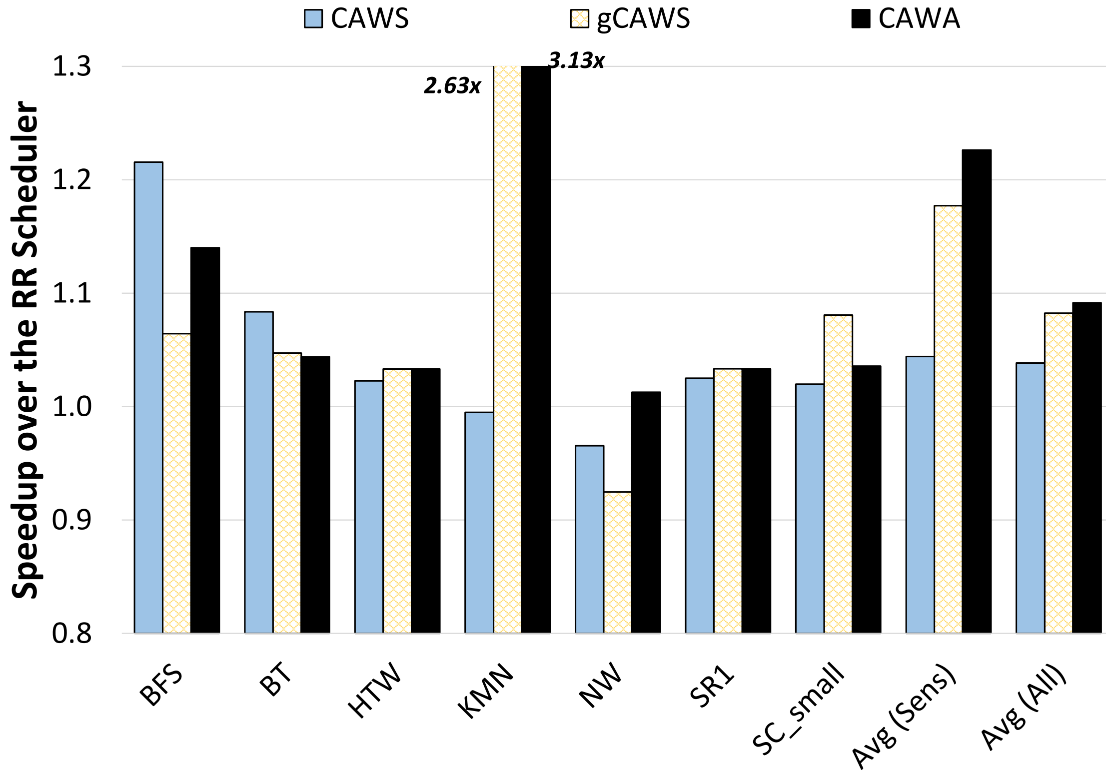

Figure 4.13: The performance improvement of gCAWS.

mance improvement comparison for CAWS with the oracle warp criticality knowledge, gCAWS, and CAWA. With the oracle warp criticality information obtained off-line, CAWS performs the best on small GPU kernels such as `BFS`, `BT` and `NW`. This is because for these small kernels, the prediction and training overhead of CPL is relatively high. Although CPL has a good prediction accuracy, gCAWS and CAWA are not able to achieve the potential speedup under using the oracle knowledge. On the other hand, for large kernel code such as `HTW` and `SR1`, my proposed gCAWS and CPL can further improve performance compared with CAWS.

I also notice that gCAWS and CAWA achieve a greater performance improvement on `KMN` than CAWS. This is because `KMN` has heavy memory contention and prefers to run with fewer number of warps. gCAWS adopts a greedy scheme to temporarily limit

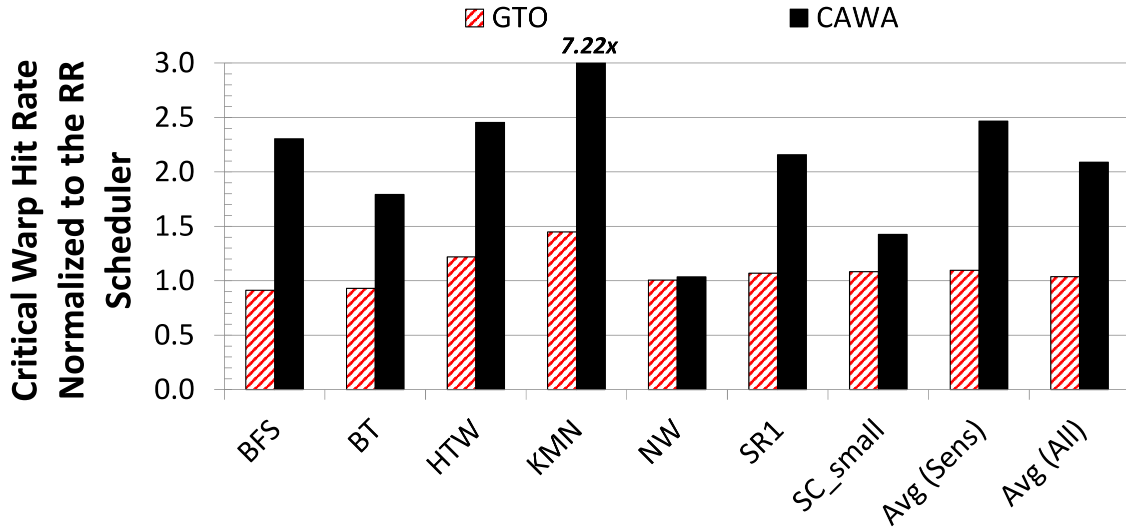

Figure 4.14: L1 data cache critical warp hit rate of CAWA.

the number of active warps while minimizing warp criticality and resource contention. Because CAWS does not limit the number of active warps to mitigate the memory contention, gCAWS and CAWA outperforms CAWS on `KMN`.

Overall CAWA can obtain an additional 5% speedup on *Sens* benchmarks compared with gCAWS. This additional performance improvement is due to the cache prioritization with CACP. However, `BT` and `SC_small` have a slight performance degradation under CAWA. This is because these two particular applications have high degree of inter-warp data reference and spatial locality. While memory requests from the critical warps have higher priority to be allocated, CACP does not take the inter-warp reference pattern into account. Therefore, warps may encounter longer memory access latency with CACP.

4.3.5 Performance Analysis for CACP

Next, I perform analysis for the cache prioritization scheme. CAWA relies on the CACP to accelerate the execution of critical warps, thereby reducing latencies coming

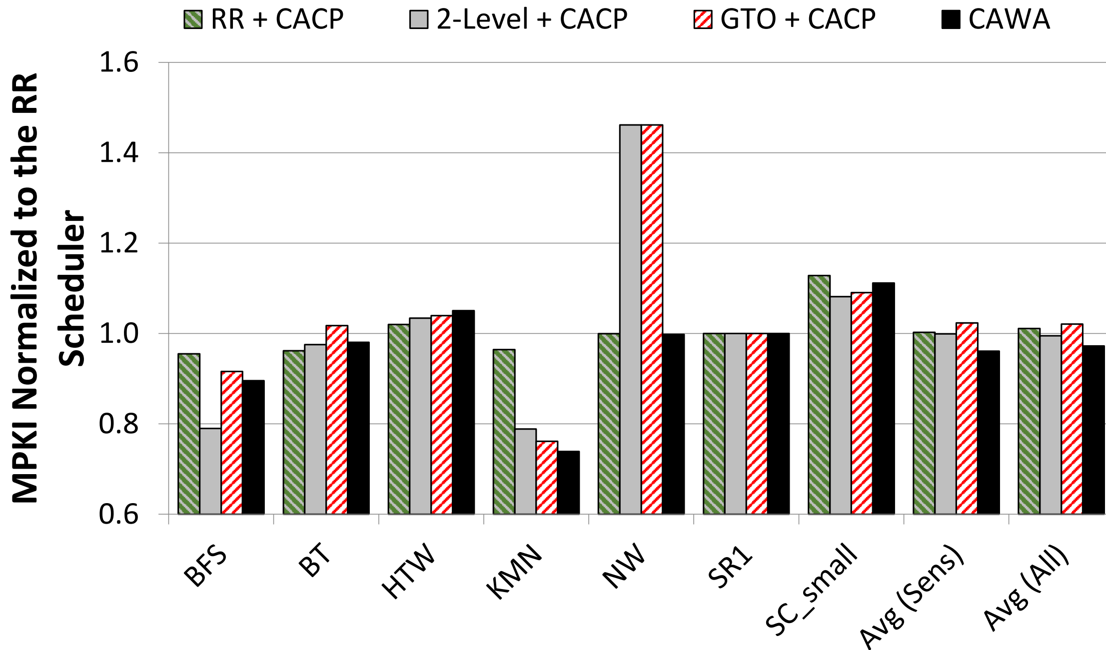

Figure 4.15: L1 data cache MPKI reduction of CACP with different warp scheduling policies.

from the memory subsystem. CACP separates the data cache to the critical and the non-critical partitions explicitly. Based on the prediction outcome from the CCBP and from the SHiP, cache blocks are inserted into either the critical or the non-critical cache partitions with the appropriate insertion positions. Figure 4.14 shows the normalized cache hit rate received by critical warp memory requests under CAWA when compared to the baseline. The explicit cache partitioning with CCBP significantly improves the hit rate for critical warps, by an average of 2.46x and by as much as 7.22x for `KMN`, which outperforms other state-of-the-art warp schedulers. Schedulers, e.g., GTO, that are criticality-oblivious, will not specifically improve the memory performance specifically for critical warps; therefore, the cache hit rate performance is less consistent across the applications.

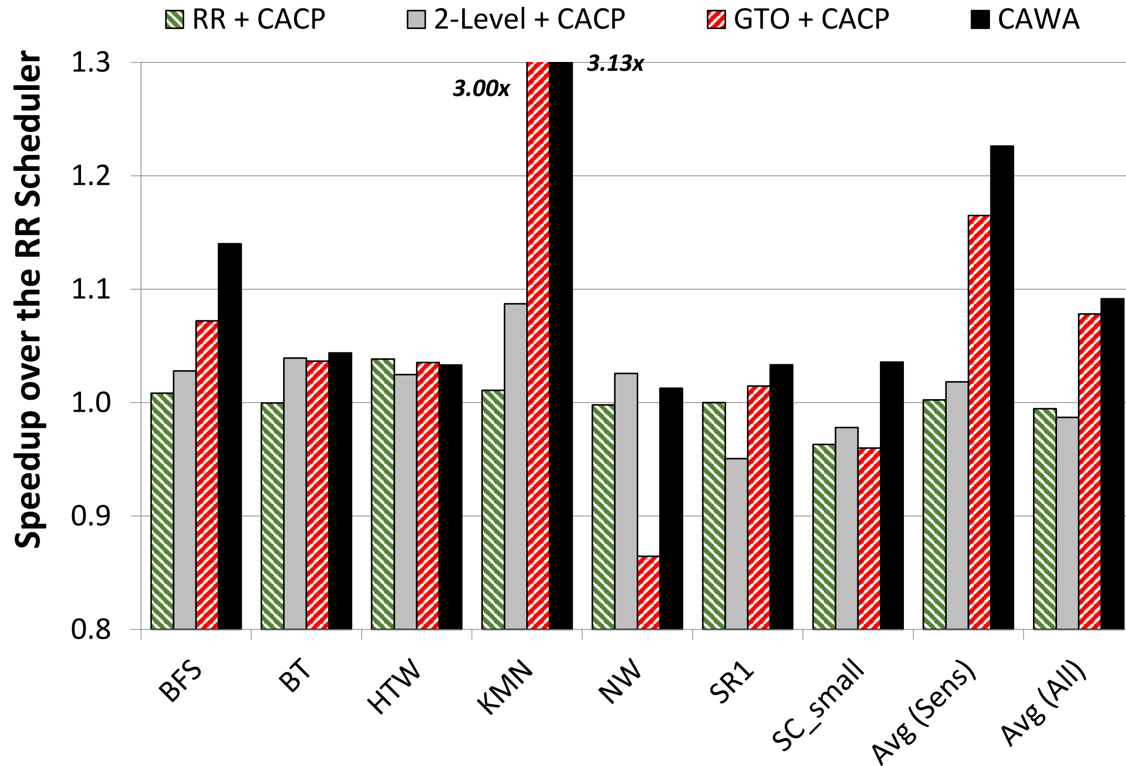

Figure 4.16: L1 data cache performance improvement of CACP with different warp scheduling policies.

To understand how well CACP performs in isolation from the gCAWS scheduler and in the presence of other state-of-the-art warp scheduling algorithms, I apply CACP to the baseline RR, GTO, and 2-Level schedulers. Independent from the warp schedulers, CACP employs the warp criticality prediction from CPL for cache prioritization. Figure 4.15 shows the MPKI reduction for the different warp schedulers under the influence of CACP and Figure 4.16 shows the corresponding performance improvement. When CACP is used in conjunction with the various state-of-the-art schedulers, additional performance improvement is achieved from 2% to 16.5% while the proposed coordinated management still performs the best.

4.4 Chapter Summary

This chapter introduces a new coordinated computation and memory resource prioritization design for GPGPU critical warp acceleration. Built upon the insights observed from the warp criticality characterization results, I design CAWA to dynamically predict critical warps and accelerate the execution of the critical warps with higher scheduling priorities and with larger scheduling time slices. Furthermore, CAWA reserves a partition of the L1 data cache for data predicted-to-be-useful for the critical warps. Therefore, the interference between critical and non-critical cache blocks is minimized. The simulation results show that the proposed design improves performance by an average of 23% for GPGPU workloads which have high warp execution time disparity and are cache-sensitive [72].

Chapter 5

INSTRUCTION-AWARE CONTROL LOOP BASED ADAPTIVE CACHE BYPASSING

In addition to the synchronization overhead and warp criticality, another significant factor that contributes to GPGPU application performance degradation is the memory subsystem (Chapter 3.2). Because GPUs execute programs in a massive multithreading manner, thousands of threads run simultaneously and compete for hardware resources such as the L1 data caches, L2 caches, and the interconnect bandwidth. This makes cache capacity and interconnect bandwidth critical resources for GPUs.

5.1 GPU Cache Access Behavior Characterization

GPU cache capacity sensitivity. To understand how caches improve GPU's performance, I first investigate the performance sensitivity to the data cache capacity. Figure 5.1 shows the performance sensitivity of GPGPU applications to the L1 data cache capacity. The x-axis represents the wide range of GPGPU applications studied in this paper (more methodology detail is given in Section 5.3) whereas the y-axis represents the speedup normalized to the baseline 16kB L1 data cache configuration. I observe that a large number of GPGPU applications—the cache sensitive (CS) workloads—gain a significant speedup with the increase in the L1 data cache capacity. When the L1 data cache size is quadrupled from the baseline 16kB configuration to 64kB, an average of 2.29x performance speedup is gained for the CS GPGPU applications. This indicates that a significant room of potential performance improvement can be gained if the L1 data cache capacity is increased or is managed more efficiently,

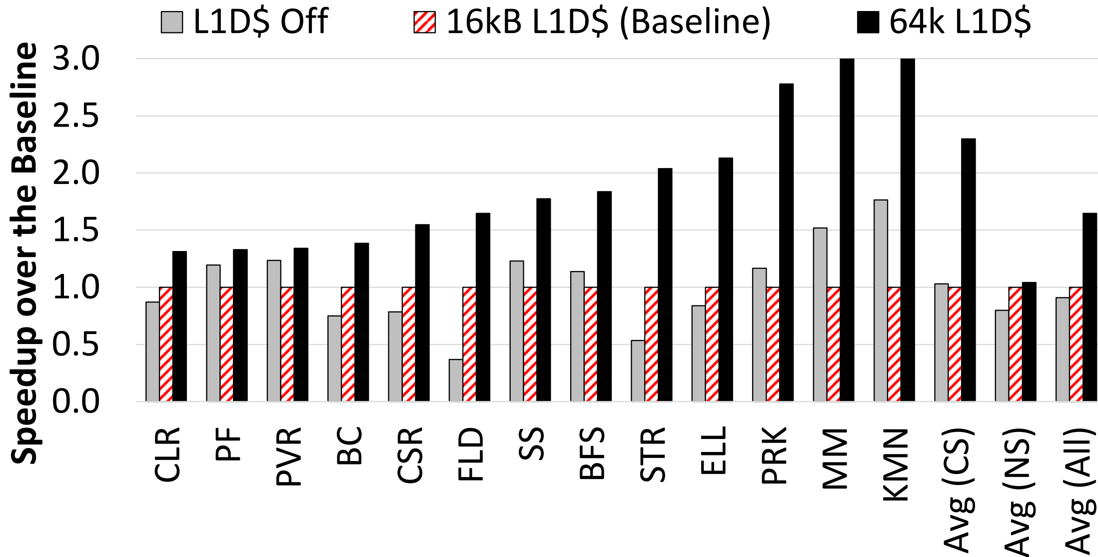

Figure 5.1: Speedup of different L1 data cache configurations over the baseline 16kB L1 data cache.

such that the large working sets common in GPGPU workloads can be accommodated more effectively.

Moreover, from Figure 5.1, I also notice that there are a number of GPGPU applications benefiting from turning off the L1 data caches completely, e.g., `MM`, `PRK` and `KMN`. This is because in these GPGPU applications, a large amount of data elements adjacent to the demanded data in a same cache line are brought into the cache but are not reused during its cache lifetime before being evicted from the cache (early eviction). Thus, spatial locality is not exploited efficiently, resulting in poor cache line utilization that wastes interconnect bandwidth and introduces additional queuing latency [51, 100]. Since simply increasing cache capacities and interconnect bandwidth to speed up the execution of GPGPU applications costs tremendous storage overhead, it is an impractical solution. A more sophisticated cache management approach is needed to improve the resource utilization efficiency of the memory subsystem in GPUs.

Algorithm 10 An example of GPGPU kernel source code from `KMN`.

```
1: function KERNEL(out[], in[], m)
2:     for i in [0 : m] do
3:         ▷ tid = the unique thread id from 0 to (n - 1)
4:         v ← in[tid * m + i]
5:         performing computations on v
6:         out[tid * m + i] ← v
7:     end for
8: end function
```

GPU data cache reuse behavior. In general, cache thrashing occurs when the data working set is larger than the cache capacity. Caches repeatedly swap in and out a cache line without receiving any hit. To illustrate the cache thrashing behavior commonly observed in GPUs, I take the cache access pattern of `KMN`, a cache sensitive application from the Rodinia benchmark suite [19], as an example. Algorithm 10 shows the pseudo code of `KMN` kernel. `KMN` iteratively reads input data from the *in* array and performs computations on each array element with n concurrent threads, where each thread works on m array elements. At each iteration, array elements ($in[i]$, $in[m+i]$, $in[2*m+i]$, ..., $in[n*m+i]$) are accessed sequentially by different threads and these array elements are mapped to multiple cache lines as shown in Figure 5.2. Since n is usually very large for GPGPU workloads, an access pattern of $(a_0, a_1, ..., a_k)^N$ in a single cache set would be observed, where a_i represents a unique access to the cache set and N represents the number of repeats. When k (or equivalently the reuse distance) is greater than the cache set associativity (S), cache thrashing occurs. Cache lines are evicted before receiving any re-reference and all memory accesses result in cache misses. Because GPUs process thousands of threads in parallel with large working sets, a similar cache access pattern with a large k can

Access Sequence: *in[0], in[N], in[2N], in[1], in[N+1], in[2N+1], in[3], …, in[kN+m]*

	Way 0				Way 1			
	Offset 0	Offset 1	…	Offset N-1	Offset 0	Offset 1	…	Offset N-1
Access 1: ***Miss***	in[0]	in[1]	…	in[N-1]			…	
Access 2: ***Miss***	in[0]	in[1]	…	in[N-1]	in[N]	in[N+1]	…	in[2N-1]
Access 3: ***Miss***	in[3N]	in[3N+1]	…	in[3N-1]	in[N]	in[N+1]	…	in[2N-1]
Access 4: ***Miss***	in[3N]	in[3N+1]	…	in[3N-1]	in[0]	in[1]	…	in[N-1]
Access 5: ***Miss***	in[N]	in[N+1]	…	in[2N-1]	in[0]	in[1]	…	in[N-1]
Access 6: ***Miss***	in[N]	in[N+1]	…	in[2N-1]	in[3N]	in[3N+1]	…	in[3N-1]

Figure 5.2: An example of thrashing in GPU caches. The data structure *in* is accessed sequentially by different threads and each access reads a word of a cache line. A cache line will never receive a hit since the working set is larger than the cache capacity. In this example, the references are mapped to a particular cache set that can accommodate two cache lines in a 2-way set-associative cache.

be commonly seen in GPGPU workloads. Cache thrashing is one of the primary bottlenecks limiting the performance of GPGPU workloads.

For such thrashing access behavior, it has been proved that inserting exactly S cache lines and bypassing all other memory requests can achieve an optimal cache hit rate [12, 97]. However, it is virtually impossible to identify k in advance with the diverse behavior of GPGPU workloads. Each GPGPU application has its own preference of the data cache configuration. Figure 5.3 shows the distribution of L1 data cache reuse distances (k) for GPGPU applications. The stacked bars show the distribution of reuse distance whereas the black curve indicates the median reuse distance. This figure provides three important insights for cache access behavior of GPGPU workloads.

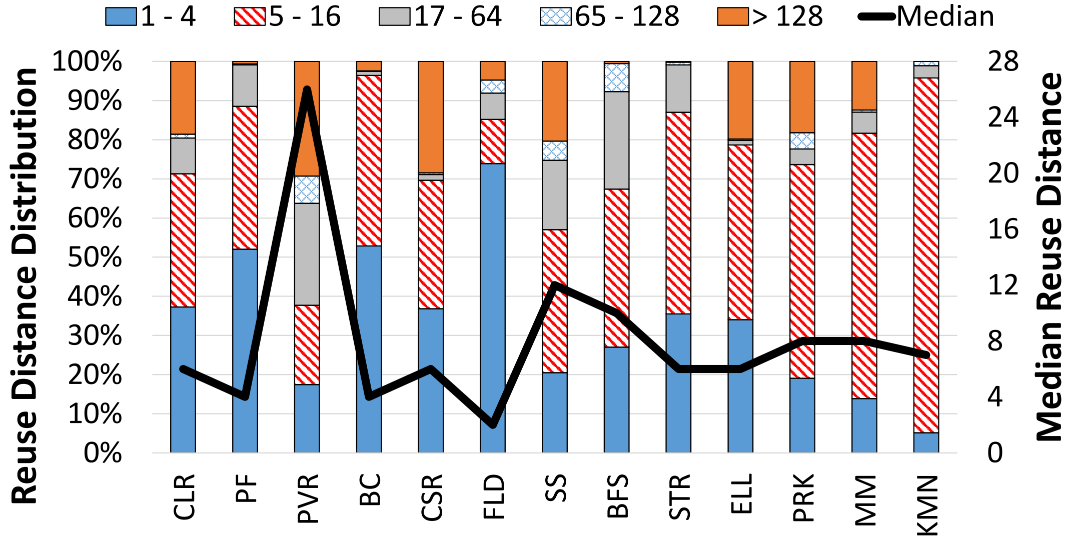

Figure 5.3: The distribution of L1 data cache reuse distance. The stacked bars show the distribution whereas the black curve indicates the median value.

1. The reuse distances (k) of these GPGPU applications often exceed 8. Since a common configuration for the GPU L1 data cache is 4 or 8-way set associative, a huge portion of cache lines will never be reused during its lifetime.

2. GPGPU applications have a diverse reuse behavior. For example, `PVR` has a long median reuse pattern whereas `FLD` has a short median reuse pattern.

3. The reuse distance can be extremely long and the distribution is dispersed within an application. For instance, in `PVR`, 29% of cache lines have reuse distances longer than 128 while 17% of cache lines have reuse distances less than 4, which are expected to receive cache hits.

According to these observations, it is difficult to design a static cache configuration or cache management policy that can achieve the optimal hit rate for all GPGPU applications.

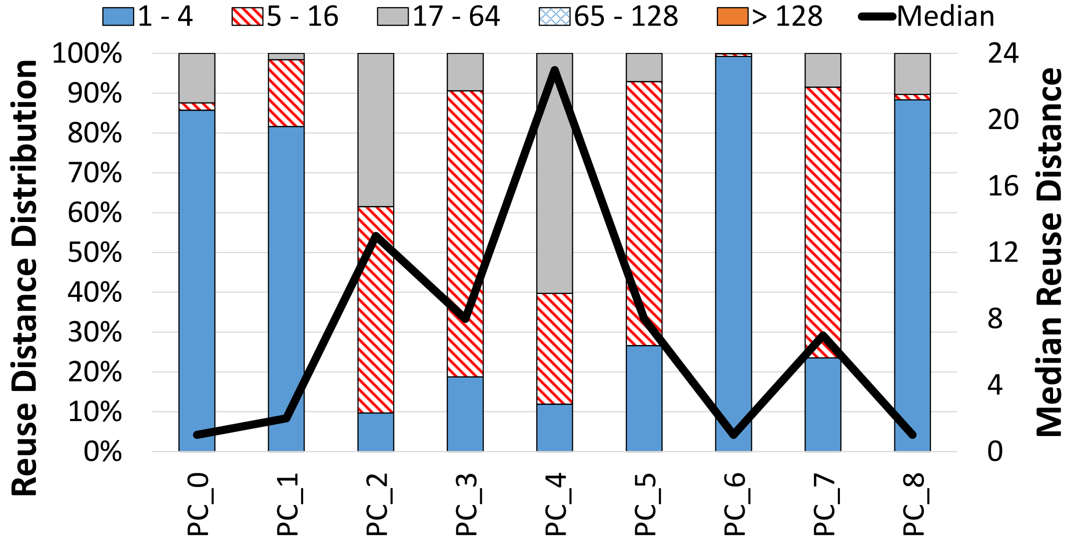

Figure 5.4: The distribution of L1 data cache reuse distance per insertion PC of BFS. The stacked bars show the distribution whereas the black curve indicates the median value.

To identify cache lines having long reuse distances within an application, I find that the distribution of reuse distances is highly correlated to the memory load/store instructions that insert a cache line. Figure 5.4 shows an example of the distribution of L1 data cache reuse distances with different memory instructions from BFS. It reveals that cache lines inserted by the same load instructions often have a similar reuse pattern. For instance, most cache lines inserted by PC_0, PC_1, PC_6, and PC_8 in BFS have short reuse distances. On the other hand, the cache lines from the other memory instructions have long reuse distances that are likely to incur cache misses. This implies that the unique program counter (PC) values of memory instructions can be a signature to predict and identify the behavior of cache lines.

GPU cache bypassing. Cache bypassing is a widely used technique to mitigate cache thrashing. Nevertheless, rather than bypassing all requests, bypassing only a selective portion of memory requests can achieve a better cache hit rate and execution

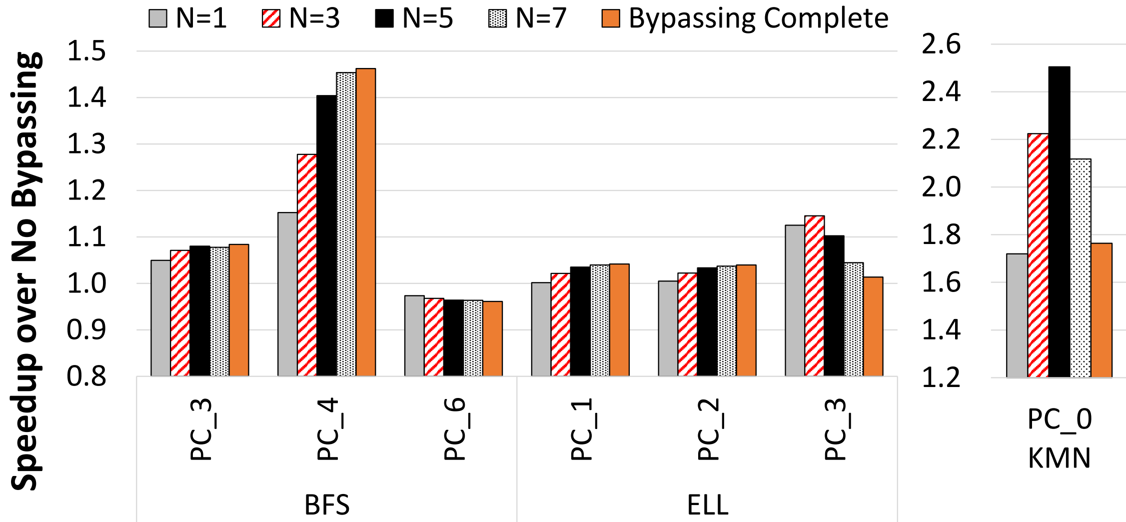

Figure 5.5: Speedup with varying an instruction's insertion and bypassing ratio. The bars represent different insertion/bypassing ratios. N indicates the insertion probability of memory requests from an instruction. A memory request has $\frac{1}{2^N}$ probability to be inserted into the L1 data cache.

time speedup [12, 97]. I have identified that different memory instructions have different cache reuse patterns. I also find that, instead of bypassing all memory references from an instruction with *long* reuse behavior, bypassing only a selective portion of the memory references is able to offer additional performance gain.

Figure 5.5 shows the application execution time speedup with different per-instruction bypassing probabilities. The x-axis represents memory instructions from different GPGPU applications and the y-axis represents the speedup normalized to no cache bypassing. Each bar represents a bypassing probability—$(1\text{-}\frac{1}{2^N})$—imposed on a memory instruction. With a greater N, a larger portion of memory requests are bypassed stochastically whereas, with a smaller N, a larger portion of memory requests are inserted into the cache. It is obvious that not all memory instructions benefit from the same degree of cache bypassing. The optimal bypassing probability varies from

instruction to instruction, and from application to application. For example, the optimal N for `ELL`'s PC_3 is 3, for `BFS`'s PC_4 is 7 (or bypassing all), and for `KMN`'s PC_0 is 5. Moreover, not all instructions benefit from bypassing, e.g., `BFS`'s PC_6. To capture the significant performance improvement potential with bypassing, a design must be able to intelligently learn the reuse patterns and dynamically adopt a variable bypassing probability on the basis of instructions.

5.2 Control-Loop Based Adaptive Cache Bypassing

5.2.1 Design Overview of Ctrl-C

In order to tackle the cache inefficiency problem in GPUs, I propose a low circuit implementation overhead design—**C**on**tr**o**l** Loop Based Adaptive **C**ache **B**ypassing (**Ctrl-C**)—to accurately predict the per-instruction cache reuse behavior and dynamically bypass memory requests to prevent cache thrashing.

Ctrl-C dynamically learns the per-instruction cache line reuse pattern in GPGPU applications, and then bypasses memory requests from the L1 data caches for instructions that generate requests with a low possibility of reuse. To do so, Ctrl-C employs feedback control loops to train the entries of an instruction reuse prediction table (iReuse Table) with the reuse history of evicted cache lines. When the fraction of zero-reuse cache lines inserted by a particular instruction is higher than a threshold, Ctrl-C starts bypassing memory requests from this instruction and increases its bypassing aggressiveness until a stable state is reached; namely, cache lines inserted with the instruction start receiving hits. Hence, a portion of the data working set is retained in the cache, leading to increased cache utilization and efficiency.

5.2.2 Cache Line Reuse Prediction and iReuse Table

In order to learn and predict the per-instruction cache line reuse pattern, I first design an instruction reuse prediction table (**iReuse Table**). iReuse Table records and predicts whether a cache line inserted by an instruction will receive reuses. The intuition is that if a cache line inserted by a particular instruction receives hits, the other cache lines inserted by the same instruction will also be likely to receive hits. To explicitly correlate the reuse patterns to the insertion instructions, I implement the iReuse Table with a simple hash table that is indexed by the lower bits of an instruction's PC value.

iReuse Table is an array of 1-bit counters with 128 entries and is indexed by the lower 7-bit of the instruction PCs. Since a GPGPU workload typically contains only tens of memory load/store instruction, a 128-entry hash table is sufficient to cover all distinct instructions without aliasing. The value of each iReuse Table entry indicates the reuse prediction for each instruction. Specifically, **0** means memory requests generated by this instruction are predicted to have *long* reuse distances, and should be bypassed from the cache because the likelihood of reuse is low. On the other hand, **1** indicates memory references predicted to have *short* reuse distances, and should be retained in the caches.

iReuse Table is trained with the reuse histories of cache lines in the L1 data caches. To track the reuse histories, Ctrl-C augments each cache line by two additional fields:

1. a 1-bit **reuse** which is used to indicate if the cache line has ever received a re-reference, and

2. a 7-bit **insertion instruction** which records the lower 7-bit of the instruction PC that brings the cache line in.

When a cache miss occurs, an eviction candidate is selected based on the underlying cache replacement policy. The reuse history of the evicted cache line, (*insertion instruction*, *reuse*), is then used to train the corresponding iReuse Table entry with a feedback control loop.

5.2.3 Feedback Control Loop

The goal of the per-instruction feedback control loops in Ctrl-C is to predict the reuse distances and to modulate the aggressiveness of per-instruction bypassing until a portion of the instruction's inserting data is retained in the cache. While iReuse Table separates instructions that generate memory requests with high likelihood of reuses from instructions with low likelihood of reuses to minimize the interference between the two types of memory accesses, for this instruction-based reuse learning and prediction mechanism to perform well, a design must handle bypassing appropriately. First, memory requests from an instruction, if bypassed, must have a chance to be inserted into the cache such that its reuse pattern can be captured by iReuse Table. Second, for instructions with *long* reuse distances, if the corresponding reuse distance can be predicted well, a portion of the instruction's active working set can be retained in the cache to start receiving cache reuses. Ctrl-C achieves this by inserting cache lines stochastically into the cache and by modulating the aggressiveness of bypassing for each individual instruction with a feedback control loop.

The feedback controller applies four additional counters to track the cache utilization behavior:

1. a 3-bit **AGG** which represents the aggressiveness of bypassing and controls the probability of cache line insertion,

2. a 7-bit **BYP** which records the total number of bypassing memory requests,

Algorithm 11 The operations of Ctrl-C at a cache miss—determining bypassing or insertion.

```
1:  function atMiss(memRequest)
2:      ▷ determining whether bypassing or not
3:      ctrl ← iReuse[request.PC]
4:      if ctrl.BYP < ((1 ≪ ctrl.AGG) − 1) then
5:          ▷ bypassing the request
6:          bypass(memRequest)
7:          ctrl.BYP ← ctrl.BYP + 1
8:      else
9:          ▷ inserting a new cache line
10:         evictedLine ← insert(memRequest)
11:         atEviction(evictedLine)
12:         ctrl.BYP ← 0
13:     end if
14: end function
```

3. a 10-bit **ZERO** which counts the total number of zero-reuse cache lines, and

4. a 10-bit **INSERT** which tracks the total number of inserting cache lines.

The *AGG* and *BYP* counters are used to regulate the aggressiveness of bypassing whereas the *ZERO* and *INSERT* counters are used to keep track of the cache utilization. Specifically, cache lines of an instruction have a probability of $\frac{1}{2^{AGG}}$ to be inserted into the cache.

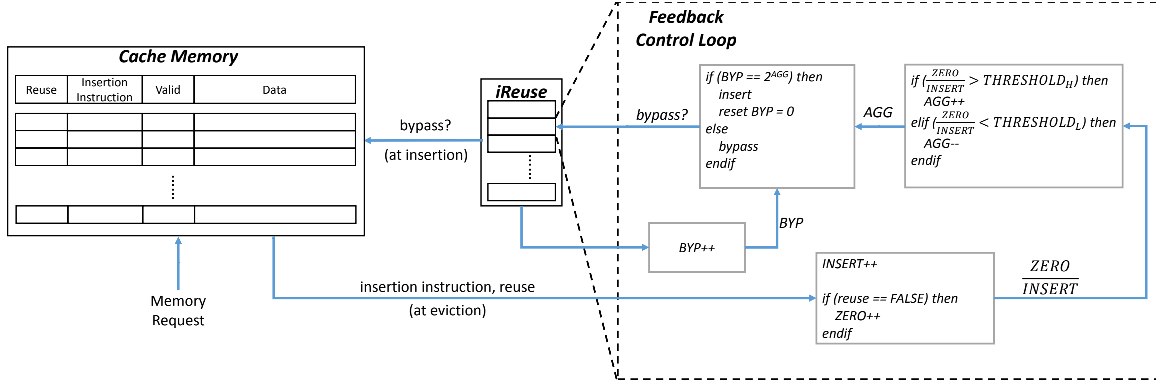

Figure 5.6: The system diagram of Ctrl-C design.

5.2.4 *The Ctrl-C Cache Bypassing Algorithm*

Next, I present the operations of Ctrl-C in detail. Figure 5.6 illustrates the architecture and bypassing algorithm of Ctrl-C. When a cache miss occurs, the corresponding iReuse Table entry is accessed for a decision on either inserting or bypassing the cache line. If a cache line is to be bypassed, the instruction's controller increments BYP and bypasses the cache line. If a cache line is to be inserted into the cache, an eviction candidate is selected and its reuse history information (*insertion instruction, reuse*) is input to the feedback controller to train the iReuse Table. The controller increments *INSERT* to keep track of the number of insertions that have occurred thus far. If the evicted cache line's *reuse* bit is 0, indicating that it has never received a reuse during its cache line lifetime, the controller also increments *ZERO* to keep track of the number of zero-reuse lines. Then, the feedback controller for the corresponding instruction updates BYP with a bypassing decision. If a cache line gets evicted in the meantime, iReuse also increments *INSERT* and updates *ZERO* based on whether the evicted cache line has received a reuse or not during the line's lifetime in the cache.

The feedback controller learns the optimal bypassing aggressiveness by periodically examining the current cache utilization, namely, the number of zero-reuse cache

lines out of the total inserted cache lines ($\frac{ZERO}{INSERT}$). Specifically, the controller aims to keep $\frac{ZERO}{INSERT}$ within a target range (*THRESHOLD_H* and *THRESHOLD_L*) in order to improve the cache efficiency. If $\frac{ZERO}{INSERT}$ shifts from the target range, the feedback controller tunes the bypassing aggressiveness by modulating *AGG*. Specifically, during a time period, if $\frac{ZERO}{INSERT}$ is greater than a certain target threshold (*THRESHOLD_H*), the number of zero-reuse cache lines is too high. This implies that the reuse distance of this instruction's cache lines is too long—more memory requests shall be bypassed from the cache to prevent data from early eviction. Thus, the controller increments *AGG* by 1 to increase the probability of bypassing for this instruction. On the other hand, if $\frac{ZERO}{INSERT}$ is lower than the low threshold (*THRESHOLD_L*), the controller decrements *AGG* by 1 to bypass memory requests less aggressively in order to recover from an incorrect prediction. When $\frac{ZERO}{INSERT}$ settles between *THRESHOLD_H* and *THRESHOLD_L*, the controller is in a stable state with the optimal degree of bypassing. Consequently, the cache retains a portion of the instruction's inserting data probabilistically and its utilization is more efficient. Algorithm 11 and 12 illustrate the operations of Ctrl-C at a cache miss and cache line eviction in detail.

5.3 Evaluation and Analysis

5.3.1 Experimental Environment and Methodology

I use GPGPU-sim version 3.2.2 [11] to evaluate the performance of the Ctrl-C. I build the Ctrl-C design on top of GPGPU-sim and run with the default configuration to simulate the NVIDIA Fermi GTX480 GPU [89]. As a comparison, I also implement a PC-based Adaptive Bypassing, which exploits per-instruction confidence counters to predict the reuse patterns [109, 119]. When detecting cache lines inserted by an instruction do not have any reuse, the PC-based Adaptive Bypassing bypasses all

Table 5.1: GPGPU-sim simulation configurations for Ctrl-C.

Architecture	*NVIDIA Fermi GTX480*
Num. of SMs	*15*
Max. Num. of Warps per SM	*48*
Max. Num. of Blocks per SM	*8*
Num. of Schedulers per SM	*2*
Num. of Registers per SM	*32768*
Shared Memory	*48kB*
L1 Data Cache	*16kB per SM (32-sets/4-ways)*
L1 Instruction Cache	*2kB per SM (4-sets/4-ways)*
L2 Cache	*768kB unified cache (64-sets/8-ways/12-banks)*
Min. L2 Access Latency	*120 cycles*
Min. DRAM Access Latency	*220 cycles*
Warp Size (SIMD Width)	*32 threads*

memory requests generated by this particular instruction. Table 5.1 and 5.2 show the configurations of the simulation infrastructure and the control loop configurations for Ctrl-C in detail.

I select a wide range of GPGPU applications from the Mars [39], NVIDIA SDK [90], Pannotia [21], and Rodinia [19, 20] benchmark suites to represent the diverse behavior of GPGPU workloads. Based on the performance sensitivity to the cache size quadruples from the baseline 16kB to 64kB, I classify these applications into two categories: (1) Cache-Sensitive (CS) applications which achieves a speedup greater than 1.2x with the 64kB L1 data caches and (2) Non-Cache-Sensitive (NS) applications

Table 5.2: Default configurations for the Ctrl-C control loop design.

Design	Configuration
Size of iReuse Table	*128 1-bit counters*
AGG	3-bit counter
BYP	7-bit counter
REF	10-bit counter
ZERO	10-bit counter
SAMPLE_PERIOD	*(1024 » AGG) cache evictions*
Target Threshold	*[0.1, 0.4]*

which has less than 1.2x speedup with the 64kB L1 data caches. Table 5.3 lists the details of the benchmarks and their input datasets.

5.3.2 Performance Improvement

My evaluation indicates Ctrl-C can achieve a significant speedup that is close to using a double sized (32kB) L1 data cache, and outperforms the PC-based Adaptive Bypassing. Figure 5.7 shows the overall performance improvement of the proposed Ctrl-C design. Compared to the baseline 16kB L1 data cache configuration, with Ctrl-C, all CS applications, except `FLD`, obtain more than 1.1x speedup and can be as high as 2.39x (`KMN`). I notice that `FLD` does not achieve a good speedup. This is because `FLD` does not have a high fraction of zero-reuse lines. Thus, Ctrl-C does not bypass any memory request and the performance is the same as the baseline configuration. Overall, Ctrl-C improves the performance of the CS applications by an average of 1.42x speedup.

In contrast, the PC-based Adaptive Bypassing improves the performance of the CS workloads by an average of 1.23x speedup, which is lower than Ctrl-C. This is

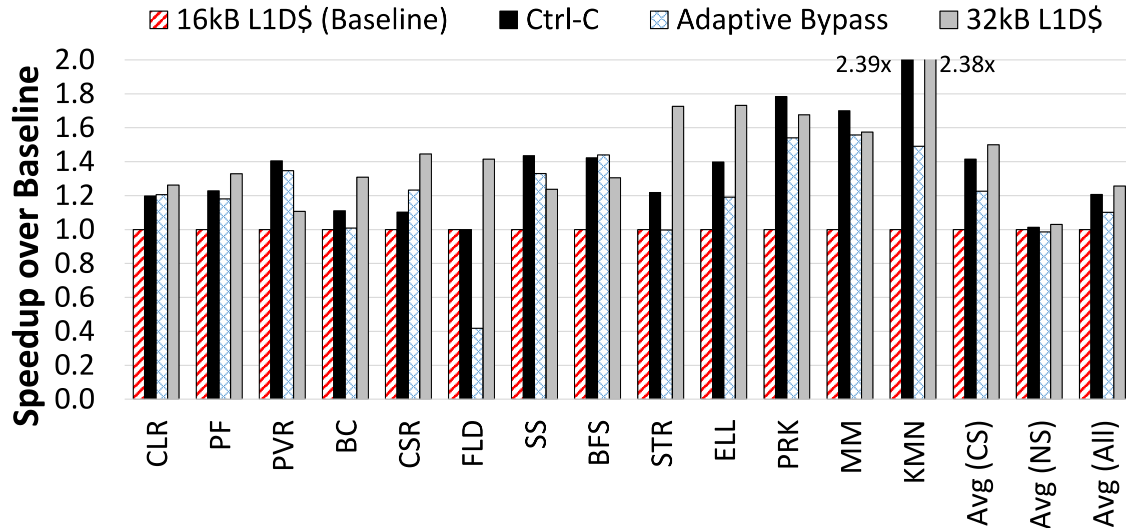

Figure 5.7: The performance improvement of Ctrl-C.

because, instead of bypassing cache lines of an instruction probabilistically, when the PC-based Adaptive Bypassing detects a portion of cache lines of an instruction do not receive reuse hits, it starts bypassing all memory references from this instruction. As a result, this design loses the opportunity to learn and capture the cache lines that can receive potential reuses once it has learned the per-instruction reuse history. Moreover, it is difficult to detect an incorrect prediction in such a design if it bypasses all requests. Therefore, with the PC-based Adaptive Bypassing, an application may instead experience performance degradation, e.g., `FLD`.

5.3.3 MPKI and Interconnect Traffic Reduction

Ctrl-C gains a great speedup by alleviating the data cache miss rate and the L1 to L2 caches interconnect traffic. Figure 5.8 and Figure 5.9 present the normalized MPKI and interconnect traffic with Ctrl-C for all CS applications.

For CS applications, the working set is typically much larger than the data cache capacity. A large amount of data is evicted from the data caches before receiving

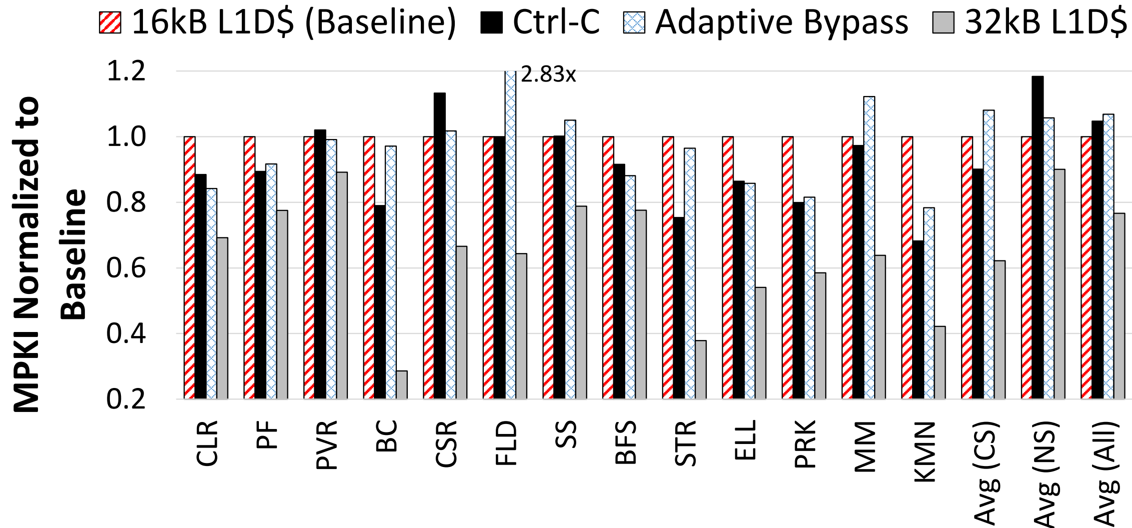

Figure 5.8: The L1 data cache MPKI reduction of Ctrl-C.

any reuse. Therefore, the performance of CS applications is mainly restricted by the cache thrashing problem. However, Ctrl-C adaptively protects cache lines from early eviction by bypassing a part of the memory requests and thereby a significant number of cache misses turn into hits. Additionally, Ctrl-C also effectively filters out the interconnect traffic since the data caches receive more hits and reduce the number of zero-reuse data elements in a cache line. Overall, with Ctrl-C, the MPKI of L1 data caches and the L1 to L2 caches interconnect traffic are reduced 9.9% and 43.7% respectively. The MPKI and interconnect traffic reductions are translated into 41.5% performance improvement.

5.3.4 Fraction of Zero-reuse Lines

Although CS applications have a significant portion of zero-reuse lines due to the severe cache thrashing problem with the baseline 16kB L1 data caches, Ctrl-C can effectively eliminate zero-reuse lines. Figure 5.10 shows the percentage of zero-reuse lines with Ctrl-C. Since the default target threshold for the feedback control loops is

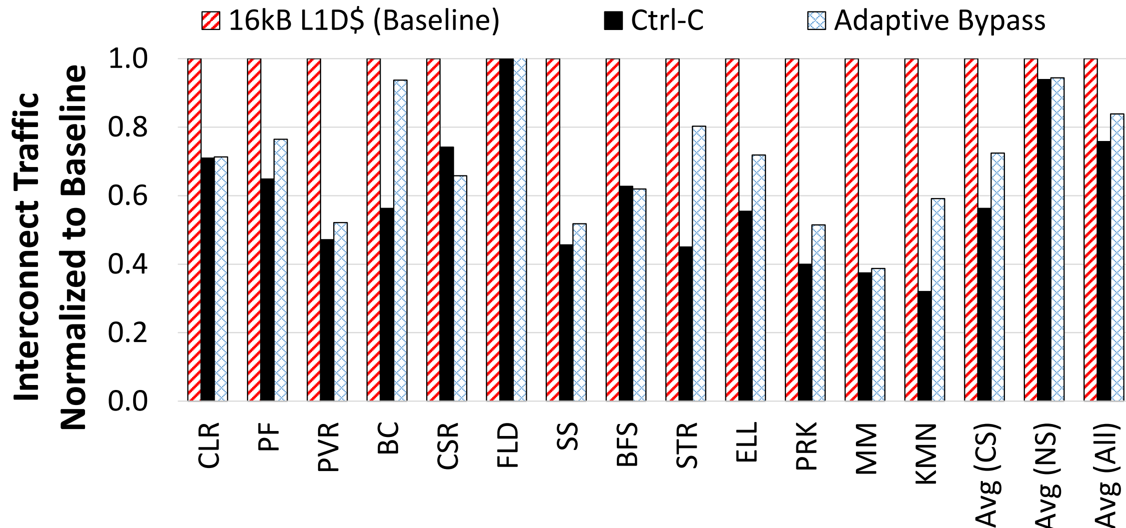

Figure 5.9: The L1 to L2 caches interconnect traffic reduction of Ctrl-C.

0.1 to 0.4 (Table 5.2), I observe that the average fraction of zero-reuse lines reduces from 79.5% to 30.8%, which meets the target threshold for CS applications with Ctrl-C.

5.3.5 Hardware Implementation Overhead

The proposed Ctrl-C is a low circuit implementation overhead design. Its implementation overhead includes iReuse table, the feedback controllers, and the two additional meta data fields per cache line. Overall, with the baseline 16kB data cache (32-set, 4-way set associative), Ctrl-C needs only 608 bytes additional storage. Compared to the baseline 16kB cache, this corresponds to approximately 3.5% storage overhead. This extra storage raises a significant 41.5% speedup for CS applications.

5.4 Chapter Summary

This chapter presents a dynamic scheme to perform cache bypassing specifically for GPUs without the need of off-line analysis. This work identifies GPU cache

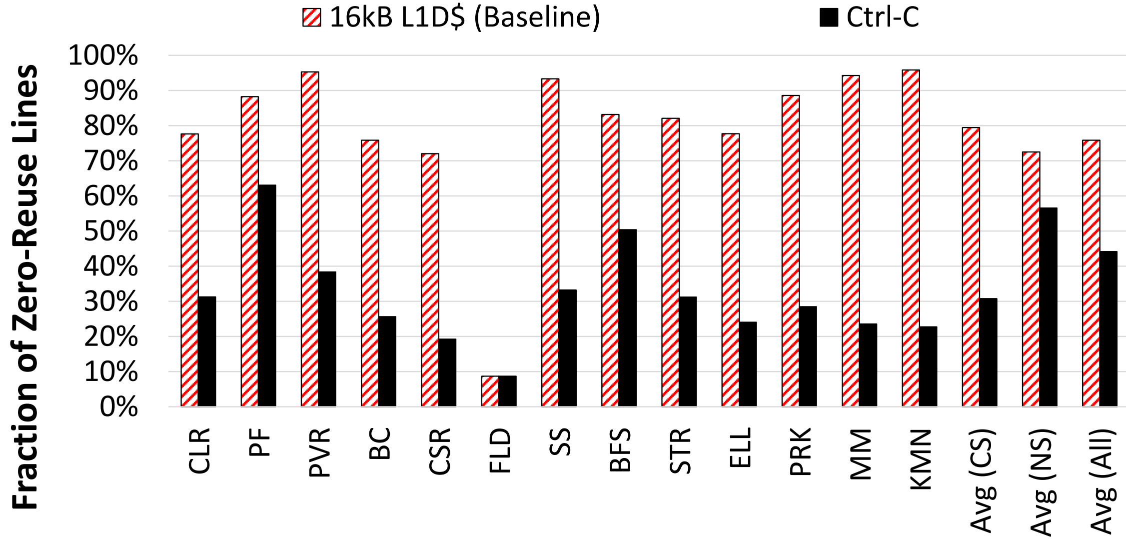

Figure 5.10: The fraction of zero-reuse cache lines with Ctrl-C.

line reuse patterns and the optimal bypassing settings vary across different memory instructions. Based on this key observation, I introduce a novel cache bypassing scheme, called Ctrl-C, to mitigate the data cache inefficiency problem in GPUs by learning and adjusting the optimal bypassing aggressiveness per memory instruction. The evaluation results shown here suggest that Ctrl-C is able to significantly reduce the MPKI and interconnect bandwidth demand. With the proposed Ctrl-C design, cache sensitive GPGPU applications can achieve an average of 1.42x speedup.

Algorithm 12 The operations of Ctrl-C at a cache line eviction—updating the bypassing aggressiveness.

```
1: function atEviction(evictedLline)
2:     ▷ updating iReuse table
3:     ctrl ← iReuse[evictedLine.insertPC]
4:     ctrl.INSERT ← ctrl.INSERT + 1
5:     if evictedLine.reuse == FALSE then
6:         ctrl.ZERO ← ctrl.ZERO + 1
7:     end if
8:     if ctrl.INSERT ≥ SAMPLE_PERIOD then
9:         fract ← ctrl.ZERO/ctrl.INSERT
10:        ▷ NOTE: ctrl.AGG is a saturating counter
11:        if fract > THRESHOLD_H then
12:            ▷ predicting long reuse distance
13:            ctrl.AGG ← ctrl.AGG + 1
14:        else if fract < THRESHOLD_L then
15:            ▷ predicting short reuse distance
16:            ctrl.AGG ← ctrl.AGG − 1
17:        end if
18:        ctrl.INSERT ← 0
19:        ctrl.ZERO ← 0
20:    end if
21: end function
```

Table 5.3: Benchmarks for Ctrl-C performance evaluation.

Abbrev.	Application	Dataset	Category
BO	Binomial Options [90]	512 Options	Non-Cache Sensitive (NS)
PTH	Pathfinder [19]	100k nodes	
HOT	Hotspot [19]	512x512 nodes	
BP	Back Propagation [19]	65536 nodes	
FWT	Fast Walsh Transform [90]	32k samples	
HTW	Heartwall [20]	656x744 AVI	
SR1	SRAD1 [19]	502x458 nodes	
NW	Needleman-Wunsh [19]	1024x1024 nodes	
SR2	SRAD2 [19]	2048x2048 nodes	
SC	Streamcluster [19]	32x4096 nodes	
BT	B+Tree [19]	1M nodes	
DCT	Discreet Cos Trans. [90]	10 blocks	
WC	Word Count [39]	86kB text file	
MIS	Maximal Ind. Set [21]	ecology	
CLR	Graph Coloring [21]	ecology	Cache Sensitive (CS)
PF	Particle Filter [19]	28x128x10 nodes	
PVR	Page View Rank [39]	1M data entries	
BC	Betweenness Central [21]	1K (V), 128K (E)	
CSR	Dijkstra-CSR [21]	USA road NY	
FLD	Floyd Warshall [21]	256(V), 16K (E)	
SS	Similarity Score [39]	1024x256 points	
BFS	Breadth First Search [19]	65536 nodes	
STR	String Match [39]	165k words	
ELL	Dijkstra-ELL [21]	USA road NY	
PRK	Pagerank-SPMV [21]	Co-Author DBLP	
MM	Matrix Mul [39]	1024x1024	
KMN	K-Means [19]	494020 objects	

Chapter 6

PERFORMANCE CHARACTERIZATION AND PREDICTION FOR HETEROGENEOUS COMPUTER SYSTEMS WITH GPUS

In Chapters 3 to 5, I focus on discussing GPU designs in the perspective of microarchitecture. Aside from the microarchitecture designs, communication with the host CMP is also a critical factor limiting the performance gain of GPUs, especially for discrete GPU cards. Commercial GPU cards are attached to the host machine with the peripheral component interconnect express (PCIe) [94] or the accelerated graphics port (AGP) [43] interface. When a GPU kernel is launched to a GPU card, the system has to synchronize the input data and computation results stored in the host main memory as well as the GPU's internal memory. With the limited bandwidth of the system bus and host main memory, the data transfer and synchronization operations are expensive in terms of execution time, becoming a significant performance bottleneck of GPU acceleration. Therefore, offloading computation onto a hardware accelerator, such as a GPU, does not have promising performance improvement. To deliver the optimal system throughput, there is a need to have a prediction mechanism which is able to accurately estimate the performance benefits for offloading versus not offloading the computation.

6.1 Heterogeneous Systems and the OpenCL Framework

Modern computer systems are accelerator-rich, integrating with many types of hardware accelerators in a single machine, e.g., GPUs. By coordinating different execution abilities provided by the accelerators, a system is able to obtain better computation throughput or lower energy dissipation. To efficiently exploit the vari-

eties of hardware accelerators, open computing language (OpenCL) is proposed as an industry standard that defines a unified programming interface for developing and running an application across different instruction set architectures (ISAs) [16]. Applications implemented in OpenCL can be dynamically compiled by an OpenCL just-in-time (JIT) compiler and adapted for running on the designated execution targets that support the standard (e.g., CMPs, GPUs, or FPGAs) without paying additional porting efforts. Hence, depending upon the requirements of an application (e.g., size of data transfer), the optimization goal, as well as the performance/power characteristics of the available devices, an intelligent OpenCL scheduler can schedule the application kernels onto different devices to be processed to improve the system efficiency as shown in Figure 6.1. For example, if the OpenCL scheduler finds out that the data movement overhead may dominate the overall execution time for a kernel, the scheduler will dispatch the kernel to run on the host CMP to minimize the execution time.

State-of-the-art OpenCL scheduling frameworks, such as [6, 114, 115], proposed to build predictive models to determine an optimal execution target among all available accelerators of different compute and power characteristics. Nevertheless, these prior works focus on scheduling a single OpenCL kernel only and do not consider realistic runtime effects such as memory interference stemmed from background processes, operating system activities, and co-located applications. For example, in an on-demand cloud computing environment, e.g., AWS [3], Google Cloud [36], and Azure [24], a compute node is able to concurrently service multiple user requests or run several copies of virtual machines with native CPU applications and OpenCL applications. In such an execution environment, co-located applications contend for shared resources in the memory subsystem and receive a varying degree of performance degradation from memory interference. Thus, existing OpenCL schedulers that only consider the

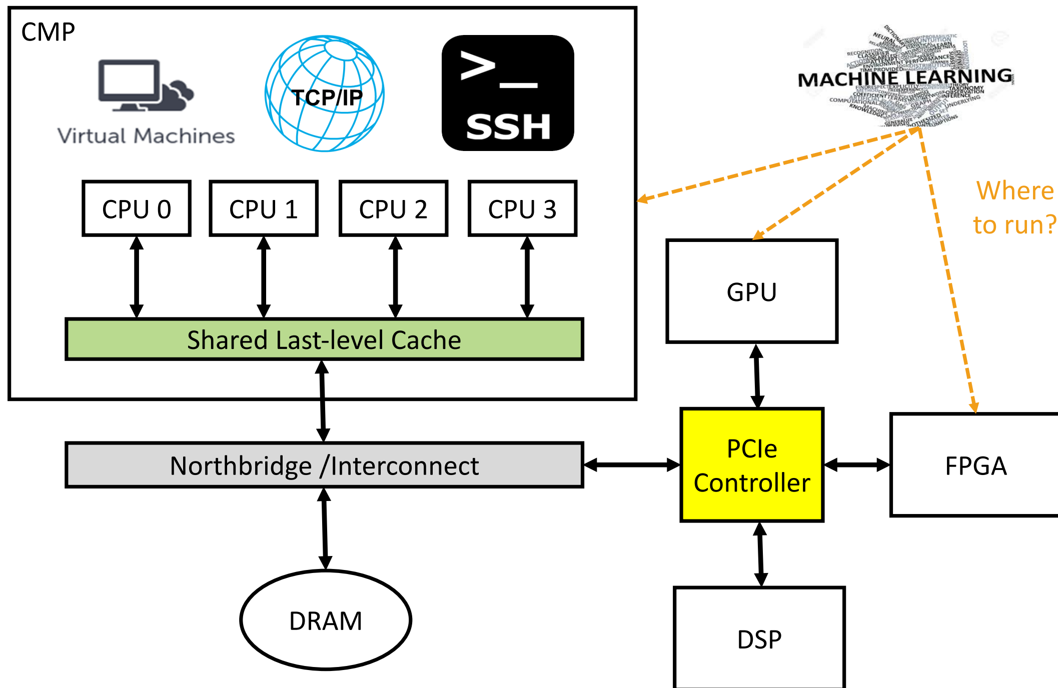

Figure 6.1: An example of a heterogeneous computer system with multiple OpenCL enabled devices. This diagram exhibits a machine equipped with a CMP and multiple hardware accelerators, including a GPU. All the CPU cores share the last-level cache, interconnect, PCIe controller, and main memory. An OpenCL application, e.g., machine learning, can be scheduled to run on the CMP or an accelerator according to the optimization goal.

characteristics of the application itself but do not take into account memory interference from co-located workloads are not robust and provide sub-optimal performance gain.

To understand the need for an intelligent scheduler that can make an accurate decision for *which optimal execution target* an application should be executed on *in the presence of memory interference*, I first perform detailed performance characterization studies for a diverse set of OpenCL applications *alone* and with *co-located*

applications in Section 6.3. Based on the observations, I then develop a light-weight and scalable performance prediction scheme, called HeteroPDP, to guide the OpenCL scheduler to accurately select the optimal execution target with the presence of memory interference in Section 6.4.

6.2 Methodology

This section introduces the experimental setup for the performance characterization studies and the design evaluation on a real heterogeneous computer system.

6.2.1 Experiment Infrastructure and Configurations

To explore the memory interference and performance degradation on a heterogeneous multiprogrammed environment, I build a system that comprises an Intel Core i7-3770 processor (a quad core CMP with an 8MB shared last-level cache) [45] and an AMD GCN2.0 Hawaii discrete GPU card [4] attached via a PCI-e 16x bus. On this system, the host processor and the GPU card share the same host DRAM controller and main memory modules. Both the CMP cores and GPU card are OpenCL-compatible and are able to execute OpenCL programs. The detailed experiment setup and system configurations are presented in Table 6.1.

To collect application-specific information for performance prediction, I instrument the OpenCL JIT compiler to generate the static information, e.g., the static instruction count (Section 6.4), as the input for the HeteroPDP predictors. To collect runtime system resource utilization information such as the last-level cache miss count, I integrate Intel's performance counter monitor toolkit (PCM) [116] into HeteroPDP to periodically collect system resource utilization information at runtime.

6.2.2 *Workload Construction*

I use a wide range of workloads exhibiting varying execution behavior for the performance characterization studies. I use 6 applications from the SPEC2006 [40] benchmarks suite with the reference dataset to represent the native CPU workloads. I classify the native CPU applications into two categories: computation or memory intensive benchmarks, based on the average miss per kilo instruction (MPKI) [78]. I take various applications from the AMD SDK [5], Intel SDK [46], Hetero-Mark [108], Pannotia [21], Rodinia [19, 20], SHOC [28], and XSBench [110] benchmark suites to evaluate the behavior of OpenCL applications. Due to the resolution of the performance counters used in HeteroPDP, I do not use OpenCL kernels that finish faster than 2 seconds and focus my studies on the longer-running 26 OpenCL application kernels as the representative benchmarks in this work. Table 6.2 and Table 6.3 list the native CPU and OpenCL benchmarks used respectively.

For the *co-located* execution scenario, I construct workload combinations by pairing one native CPU application and one OpenCL application, which results in 6*26 = 156 multiprogrammed workloads. To study the scalability of HeteroPDP, I increase the number of native CPU applications and synthesize an additional 38 multiprogrammed workloads, consisting of two SPEC applications and one OpenCL application from the listed benchmarks. To prevent the experimental machine from overheating and from thermal throttling, the 38 workloads are the combinations that complete within 5 minutes.

6.3 Motivation for an Intelligent Execution Target Scheduler

In this section, I present the performance characterization and analysis for the *alone* and *co-located* execution scenarios to motivate the need of an accurate perfor-

mance degradation predictor and execution target scheduler for heterogeneous systems. In the *alone* case, an OpenCL application is the sole application running on a heterogeneous system and is to be dispatched onto an execution target among all available processors or accelerators (discrete GPU cards in my thesis). On the other hand, in the *co-located* case, an OpenCL application is to be dispatched onto the heterogeneous system, which is servicing other applications, i.e., native CPU applications.

6.3.1 Performance Characterization

Offloading an OpenCL application onto a hardware accelerator does not always lead to performance improvement or energy reduction. This is mainly because of three reasons. First, to perform computations on an accelerator, it often requires moving a considerable amount of data between the host system and the accelerators to synchronize the execution, which is expensive in terms of execution time and energy consumption [13, 37, 81, 93, 108]. Second, to make the shared data accessible by the host CPU as well as the hardware accelerators, the device driver or operating system has to frequently modify the page tables and translation lookaside buffers (TLB) to remap the data into different memory spaces, which can introduce very long operation latencies [112]. Third, the OpenCL JIT compiler is not always able to transform and optimize the OpenCL kernel code well to fully utilize the dedicated target accelerator, making the performance sub-optimal [114]. Consequently, offloading computations onto an accelerator may instead degrade the application performance and incur higher energy dissipation.

Figure 6.2 shows the system performance for running an OpenCL application on the Intel CMP or the discrete GPU card *alone* and *co-located*, averaged across the 26 OpenCL applications. The horizontal axis indicates the execution target of the

OpenCL application whereas the y-axis represents the system performance: execution time speedup for *alone* and fairness for *co-located*. Note that, *fairness* is a commonly-used metric to evaluate the execution time slowdown for multiprogrammed execution environments [9, 31, 86]. It is defined as the ratio of the minimum and the maximum execution time slowdown among all concurrent applications as shown in Equation 6.1, where i represents any of the co-located applications and *slowdown* is the ratio of an application's execution time in *co-located* and that in *alone*.

$$Fairness = \frac{min(slowdown_i)}{max(slowdown_i)} \tag{6.1}$$

Figure 6.2(a) shows that, although offloading the OpenCL application to the GPU achieves an impressive speedup on average as compared with the CMP execution target, there is ample room for performance improvement. With the oracle execution target information, the application performance can be further improved by an average of 50%. Furthermore, Figure 6.2(b) shows a similar performance trend for *co-located* cases. Clearly, *to maximize system performance, an intelligent execution target scheduler is needed for both the alone and co-located execution scenarios.*

6.3.2 Optimal Execution Target in the Presence of Memory Interference

I delve deeper into a few workload combinations to illustrate that the optimal OpenCL execution target varies in the presence of memory interference from a memory-intensive co-located application. In this study, I use `mcf` as the memory-intensive application running on the CMP. When an OpenCL application is co-located with `mcf` on the CMP, shared last-level cache contention degrades application performance whereas when the OpenCL application is offloaded to the GPU, performance degradation comes from a different level of the memory hierarchy, i.e., the DRAM memory

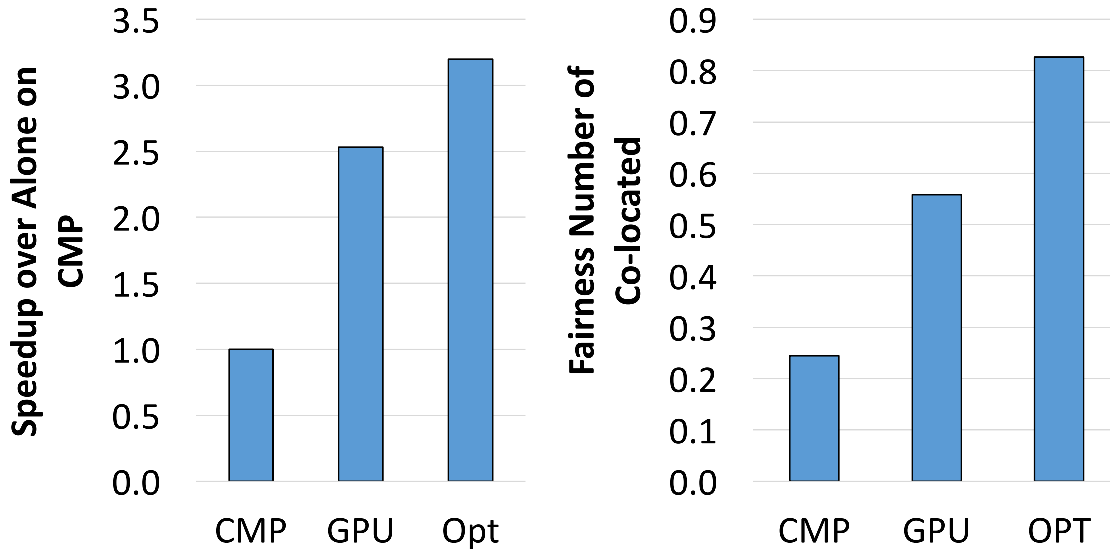

Figure 6.2: The average execution time speedup of running OpenCL Applications *alone* and the execution time slowdown fairness of *co-located* on a quad-core CPU, GPU, and the optimal execution target between the CPU and GPU devices.

bandwidth. The already expensive data transfer cost for OpenCL application offloading is exacerbated.

Figure 6.3(a) shows the execution time speedup of five different OpenCL applications *alone* on the CMP versus the GPU accelerator and the optimal, higher-performing execution target. Figure 6.3(b) shows the execution time speedup of the same OpenCL applications *co-located* with `mcf` and the optimal execution target. The optimal execution target for three out of the five OpenCL applications, i.e., `BIT`, `HIS`, and `XSB`, is changed. It is clear that the scheduling decision depends upon the memory intensities and interference between the co-located workloads. Hence, *simply considering the features of an OpenCL application is insufficient to maximize application and system performance—it is crucial for an intelligent execution target scheduler to take into account the characteristics of all co-located applications.*

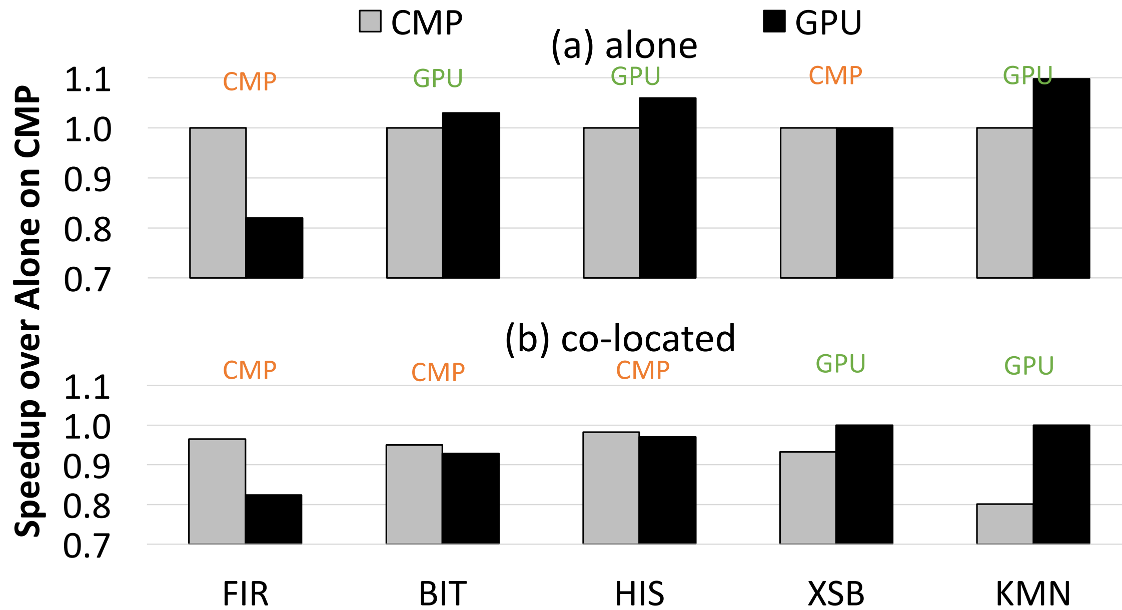

Figure 6.3: The execution time speedup of an OpenCL application for (a) it is running *alone* and (b) it is *co-located* with the native CPU application `mcf`. The labels on the top indicate that the optimal target device based on the execution time speedup.

6.3.3 Performance Degradation with Different Co-location Scenarios

To fairly evaluate the overall system performance, the *fairness* metric is commonly-used for co-located workloads in the multiprogramming execution as defined in Equation 6.1 [9, 31, 86]. The goal of using fairness as the optimization goal is to ensure a fine balance of the slowdown among all co-located applications. A fairness number of 1.0 represents a system with equal slowdown among all co-located workloads. That is, an ideal system design for a multiprogramming environment is to make the fairness closer to 1.0.

In order to identify the execution target preference in my experimental platform with optimal fairness, I also define the GPU and CPU fairness ratio as shown in Equation 6.2. That is, with a higher fairness ratio, it implies an OpenCL kernel

has stronger preference to run on the GPU and vice versa. Figure 6.4 shows the execution target preference for the OpenCL application in the *co-located* scenario for all 156 workload combinations in this study. The x-axis represents all workload combinations while the y-axis represents the fairness ratio of the OpenCL application running on the CMP versus on the GPU. The data points are sorted based on the fairness ratio in the increasing order. We can observe that, for a large number of workload combinations (toward either end of the curve), there is a clear OpenCL execution target preference. Besides, the fairness ratio varies significantly, from 0.001 to 100.

$$Fairness\ Ratio = \frac{Fairness_{CMP}}{Fairness_{GPU}} \tag{6.2}$$

6.3.4 Performance Degradation with Different Scheduling Priorities

Real-time constraint and scheduling priorities of processes can affect the scheduling decision as well. Many interrupt services, for example, must be handled by the host processor with a hard real-time deadline. To evaluate how scheduling priorities can influence the scheduling decision of an OpenCL application and affect the overall system performance, I adopt the metric of weighted slowdown [31] and use it to calculate fairness as defined in Equation 6.3 and 6.4, where $weight_i$ represents the scheduling weight given to the process i. Figure 6.5 presents the fairness ratio based on the weighted slowdown of each co-located native CPU application with the weight factor varying from 0.5 to 2.5. 0.5 means the co-located native CPU application is more latency tolerable than the OpenCL application, whereas 2.5 indicates the co-scheduled native CPU application is highly latency critical. The weights can also be representative of, for example, the operating system scheduling priority. We see that

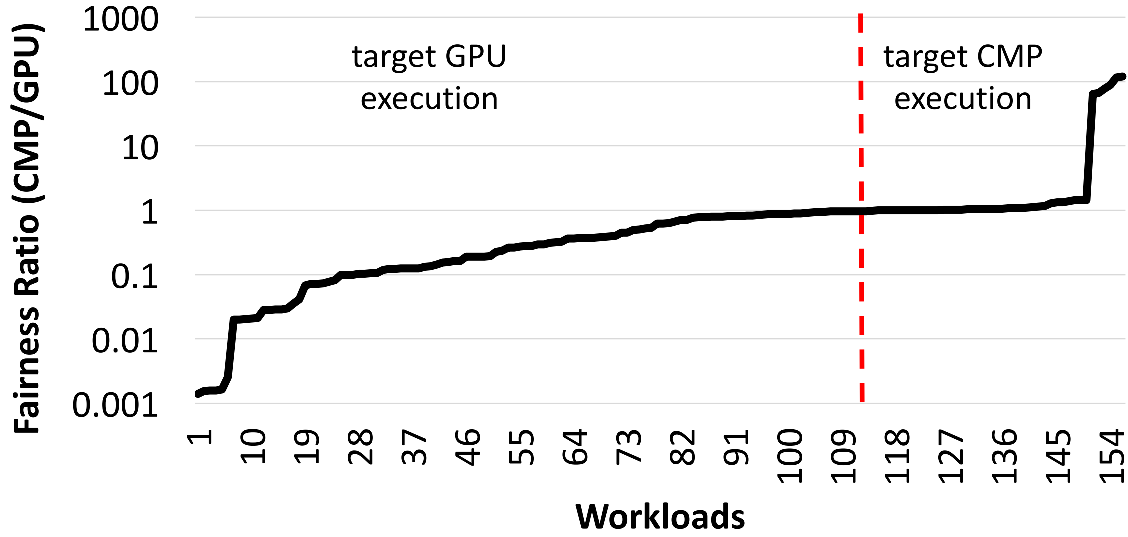

Figure 6.4: The fairness ratio between running an OpneCL Kernel on the CMP versus on the GPU for workloads comprising one OpneCL application and one native CPU application. Higher than 1.0 indicates running on CMP has higher fairness number and thereby preferring to run on the CMP.

when the scheduling priority of the co-located native CPU process increases, the fairness ratio shifts remarkably as well, favoring GPU as the OpenCL execution target as labeled with the blue boxes in Figure 6.5. Therefore, in order to meet the real-time deadline, an intelligent OpenCL execution target scheduling framework should also consider the process scheduling priorities to reach a correct target selection decision.

$$WeightedSlowdown_i = slowdown_i \times weight_i \tag{6.3}$$

$$WeightedFariness = \frac{min(WeightedSlowdown_i)}{max(WeightedSlowdown_i)} \tag{6.4}$$

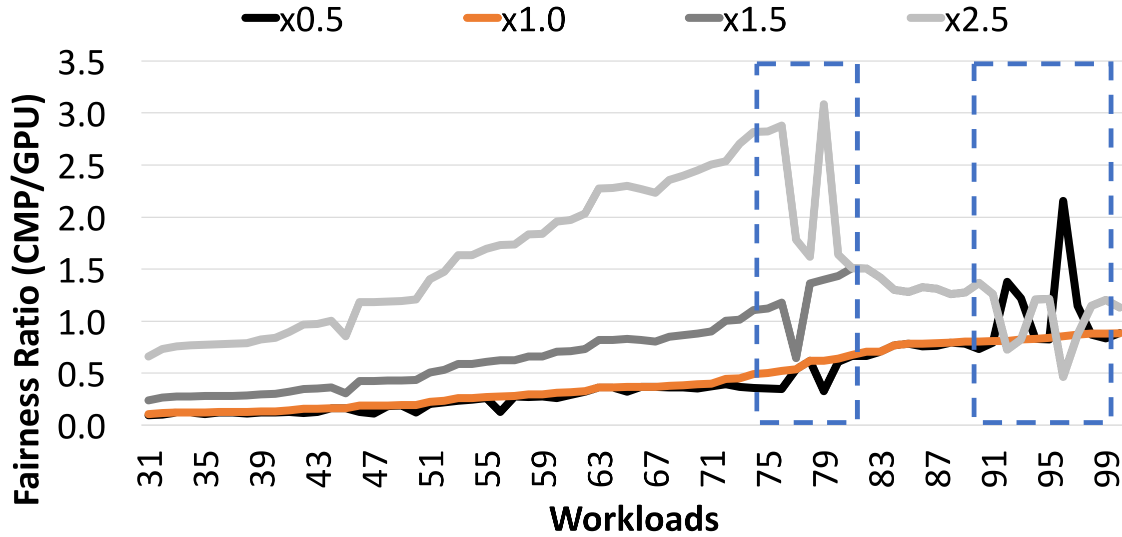

Figure 6.5: The fairness ratio of running an OpneCL Kernel on the CMP versus on the GPU when the co-located native CPU application is assigned to have different OS scheduling priorities/weights. The blue boxes point out workloads having varying target execution devices when the co-located application has different scheduling weights.

6.4 Performance Degradation Predictor for Heterogeneous Systems

Based on the performance characterization studies discussed in Section 6.3, I design a simple, light-weight performance prediction and optimization framework, called *HeteroPDP*. The goal of HeteroPDP is to estimate application slowdown for each co-located application and schedule the OpenCL application to an execution target in a heterogeneous system to maximize the fairness, system throughput, or weighted speedup.

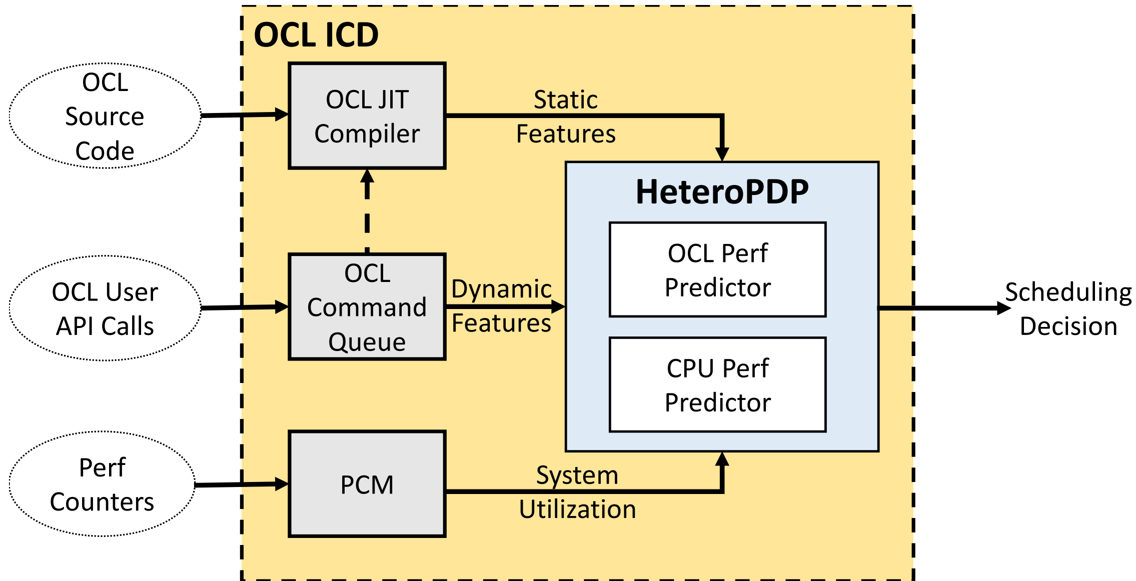

Figure 6.6: System diagram of the HeteroPDP prediction scheme.

6.4.1 The HeteroPDP Prediction Scheme Overview

HeteroPDP is implemented as a part of the OpenCL independent client driver (ICD). When an OpenCL API is invoked within an application, HeteroPDP collects application-specific information such as the size of data transfer between the host and device memories, available in the command queue of the ICD. Based on the application-specific features important for performance prediction, HeteroPDP estimates application execution time for both OpenCL application and native CPU applications, and selects an execution target for the OpenCL application. The proposed HeteroPDP framework is illustrated in Figure 6.6.

To predict the execution time and performance degradation for OpenCL kernels, I use a regression-based approach. While a full-fledge machine learning technique can also be used and may offer higher prediction accuracies, my evaluation result in the later section (Section 6.5) indicates a simple performance model works sufficiently well to facilitate the execution target selection for OpenCL kernels.

6.4.2 OpenCL Kernel Execution Time Prediction for *alone*

To establish the regression model for predicting the performance of an OpenCL application or kernel when it is running alone in a heterogeneous system, I first analyze and identify a set of important kernel characteristics, including both static and dynamic features. The static features of a kernel can be retrieved by the OpenCL JIT compiler at the compilation time, e.g., the number of static instructions. On the other hand, the dynamic features of a kernel include parameters such as input data sets and user commands specified at the kernel launch time (e.g., the total number of threads).

The kernel characteristics are extracted with the instrumented OpenCL JIT compiler and the device driver, and are used to train the regression-based performance prediction models: one for predicting the OpenCL application execution time of the host CMP execution target and the other for the GPU execution target.

I run an OpenCL kernel with a varying number of threads and different sizes of input data sets and collect its corresponding execution time by querying the *clGetEventProfilingInfo()* API. I construct the correlation between the features and the execution time. Overall, the regression model expresses the predicted execution time as a function of a number of important features, as shown in Equation 6.5, where c_i and f_i represent the *i-th* coefficient and feature, respectively. Table 6.4 summarizes the kernel-specific features used in the performance prediction models for the execution targets of the host CMP and the GPU. Note that, these parameters and features are chosen to form the regression models because they are identified to be highly correlated to kernel execution time.

$$Performance_{execution\ target} = \sum_i c_i \times f_i \tag{6.5}$$

6.4.3 OpenCL Kernel Execution Time Prediction for co-located

Similar to predicting the execution time for an OpenCL application *alone*, I build an additional regression model to predict the kernel execution time in the presence of co-located applications. In such an execution scenario, shared memory resource utilization, e.g., the last-level cache and the DRAM bandwidth on the host CMP, influences the OpenCL application performance. To consider the memory interference effects, I include two additional features into the regression performance prediction model for *co-located*: (1) the shared last-level cache miss counts on the host CMP and (2) the host DRAM bandwidth utilization incurred by the co-located native CPU applications.

In summary, when an OpenCL kernel is launched, I use the regression models to predict the OpenCL kernel execution time for (1) each of the two available execution targets, *alone* ($time_{alone}$ with Equation 6.5) and (2) each of the two available execution targets, *co-located* ($time_{co-located}$). Then, HeteroPDP estimates the slowdown factor of the OpenCL application for the two execution targets with Equation 6.6. Details of the parameters and features used for the regression model training and the kernel execution time prediction are summarized in Table 6.4.

$$Slowdown = \frac{time_{co-located}}{time_{alone}} \tag{6.6}$$

6.4.4 Performance Model Training for OpenCL Kernels

To build the regression models for OpenCL kernel execution time prediction in HeteroPDP, I take a large set of 63 distinct OpenCL kernels with varying input data set sizes as the training set. I apply the commonly-used K-fold cross validation algo-

rithm [95] with 32 test passes to eliminate overfitting and to maximize the coefficient of the determination value (R-square) by narrowing down the training set size from 63 to 45 kernels. Coefficients employed for the regression models evaluated in this research work are listed in Appendix A.

6.4.5 Performance Degradation Prediction for Native CPU Applications

To assess fairness or weighted speedup of multiple concurrent applications running on the heterogeneous system, HeteroPDP has to determine the performance of native CPU applications as well. It does so with an offline-trained table. A major advantage of using an offline-trained table is the ease of computation overhead. Therefore, instead of applying a prediction model to project the execution time slowdown of co-located native CPU applications, I adopt the previously proposed Bubble-up algorithm to measure and estimate the CPU application slowdown caused by the co-schedule OpenCL application after the compilation of the OpenCL application [82]. In Bubble-up, a simple hash table is accessed at the compilation time to predict the degree of performance degradation under different levels of shared memory contention caused by other co-located applications. The table is constructed for each native CPU application and is trained with a collection of microbenchmarks that generate a fixed level of contention for a specific shared memory resource, such as the last-level cache or the shared DRAM bandwidth. Note, Bubble-up was originally proposed for application slowdown estimation of CPU applications in a multiprogramming execution scenario. I revise the algorithm for the purpose of performance degradation prediction for native CPU applications in a heterogeneous system setup.

For HeteroPDP, if an OpenCL kernel is running on the host CMP, the main resource contention occurs at the shared last-level cache. To predict the pressure the OpenCL kernel imposes onto the shared cache, I use the maximum number of

concurrent threads that can run on the CMP's SIMD units and the total working set size to estimate its demand for the shared cache capacity. On the other hand, when the OpenCL kernel is offloaded onto the discrete GPU, the major resource interference occurs at the data movement operations for the shared main memory bandwidth. To predict the slowdown caused by the bandwidth contention, HeteroPDP uses the total size of data transfer required for launching the OpenCL kernel to evaluate the host DRAM bandwidth requirement.

6.5 Evaluation and Analysis

In this section, I present the evaluation results for the accuracy of the prediction model as well as the performance of the proposed HeteroPDP scheme in the *alone* and *co-located* execution scenarios.

6.5.1 Execution Time and Execution Target Prediction Accuracy

The ultimate goal of the HeteroPDP framework is to predict the optimal execution target for an OpenCL application in the *alone* and *co-located* execution scenarios. Since HeteroPDP depends its execution target prediction on the four execution time prediction models, I also evaluate the prediction accuracy for the four individual models. Figure 6.7 presents the execution target selection accuracy for *alone* and *co-located*. The different portions of the bar represent the different prediction outcomes [**predicted execution target**, **optimal execution target**]. For instance, [**CMP**, **GPU**] means that the predicted execution target for the OpenCL application is the CMP host processor and the optimal execution target is the GPU card, resulting in an incorrect prediction outcome. For the *alone* case, the execution target is selected such that the execution time of the OpenCL application is minimized. In contrast, for the *co-located* case, the execution target is selected such that fairness, as defined

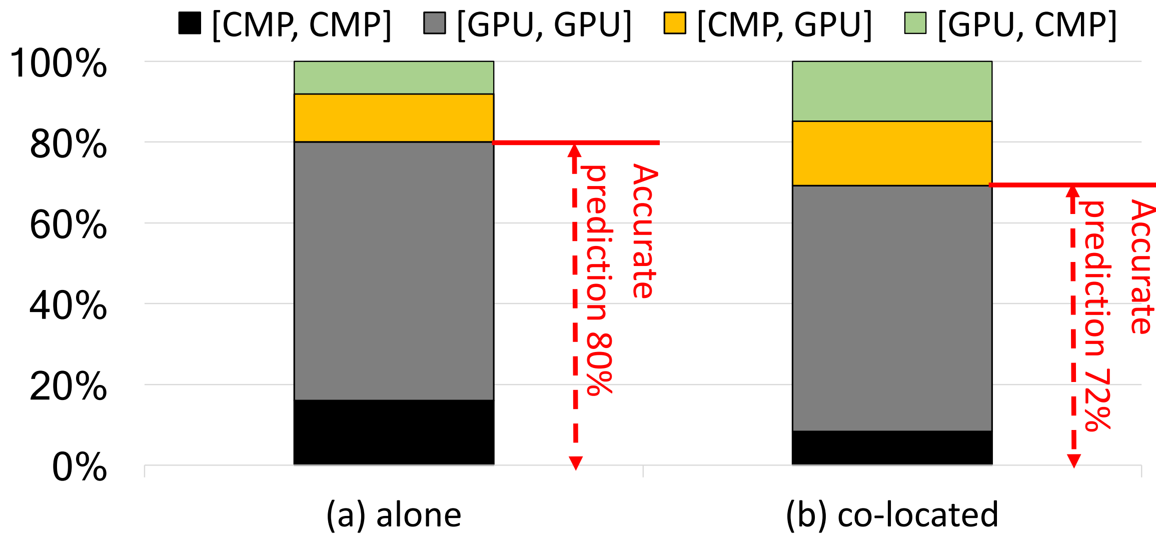

Figure 6.7: The prediction accuracy of selecting the optimal execution target device to run an OpneCL kernel when (a) running *alone*, and (b) *co-located* with a native CPU process.

in Equation 6.1, is maximized. Overall, HeteroPDP achieves 80% and 72% execution target prediction accuracy for the *alone* and *co-located* scenarios, respectively.

I investigate the prediction accuracy for the individual execution time models as well. Figure 6.8 shows the cumulative density function (CDF) for the execution time prediction accuracy for (1) the OpenCL application on the host CMP, *alone*, (2) the OpenCL application on the GPU, *alone*, (3) the OpenCL application on the host CMP, *co-located*, and (4) the OpenCL application on the GPU, *co-located*. We can observe that the execution time prediction error rate for the majority of applications or workload combinations is below 10%. For the four respective models, (1)–(4), 73%, 70%, 68%, and 72% of the workloads can meet the 20% error rate cutoff.

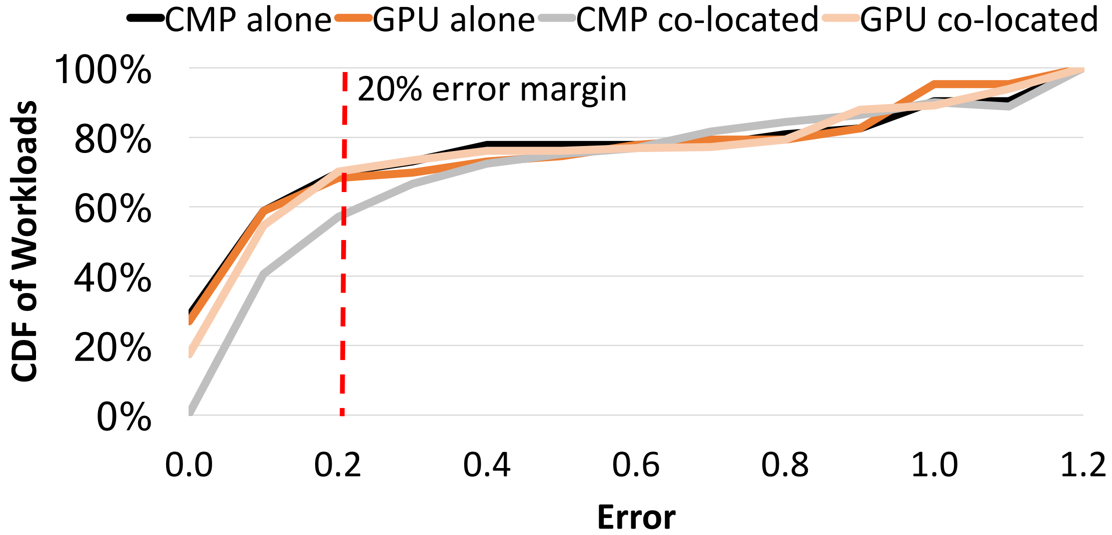

Figure 6.8: The CDF of prediction errors for predicting OpenCL kernel execution time. The red dash line indicates the 20% error margin.

6.5.2 Evaluation for System Performance

I next investigate the application and system performance impacts of HeteroPDP for *alone* and *co-located*. Figure 6.9 shows the performance speedup for an OpenCL application running *alone* on the target heterogeneous system. The bars represent the OpenCL application running on different execution target (CMP, and GPU), the execution target selected by HeteroPDP, and the optimal execution target (Opt), whereas the y-axis plots the speedup over the baseline execution target (CMP). We can observe that the *always offloading to GPU* choice improves the OpenCL application performance by 2.5x, while HeteroPDP improves the application performance by 3.0x. HeteroPDP bridges the performance gap between *always offloading to GPU* and *the optimal target selection* by 72%.

Figure 6.10 shows the respective performance speedup for the native CPU application and the OpenCL application of the *co-located* multiprogrammed workloads.

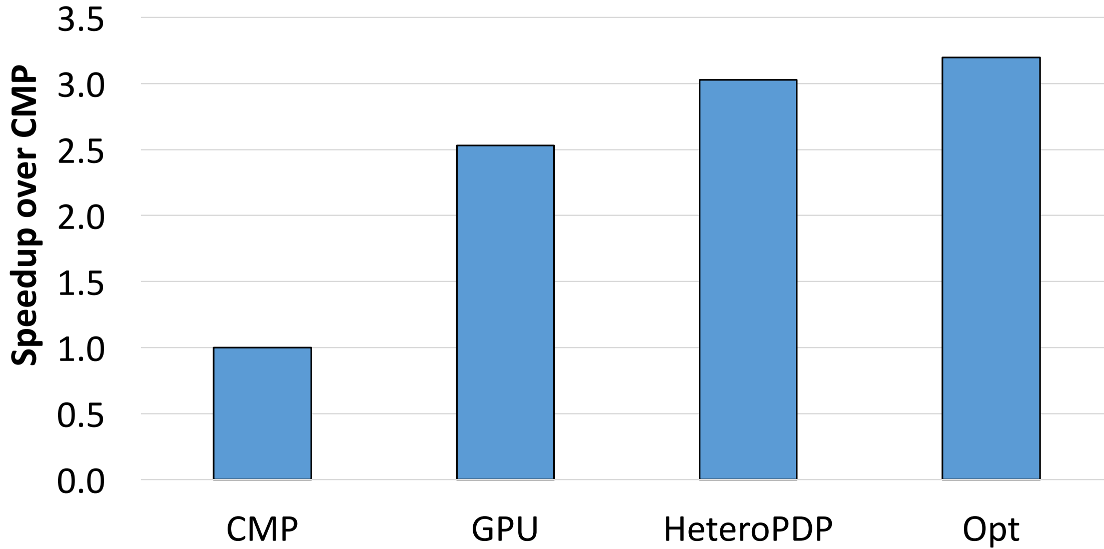

Figure 6.9: The system speedup of HeteroPDP when running an OpenCL application *alone*.

The x-axis again shows the execution target of the OpenCL application (the *Avg* bar indicates the average throughput across all co-located applications), the left y-axis shows the application performance speedup normalized to the baseline (where the OpenCL application runs on the host CMP), and the right y-axis plots the fairness evaluation. Similar to the *alone* execution scenario, the proposed HeteroPDP improves the weighted speedup over the *always offloading to GPU* choice and, at the same time, improves the fairness of the co-located applications.

6.5.3 HeteroPDP with Varying Scheduling Priorities

Assigning equal weights to the native CPU applications and the OpenCL application is not reflective of the scheduling priorities to be enforced in typical systems. As previously mentioned, HeteroPDP can be configured to consider the priorities of co-located applications when making a scheduling decision. Thus, I perform a characterization study by varying the weight ratio of the native CPU application and the

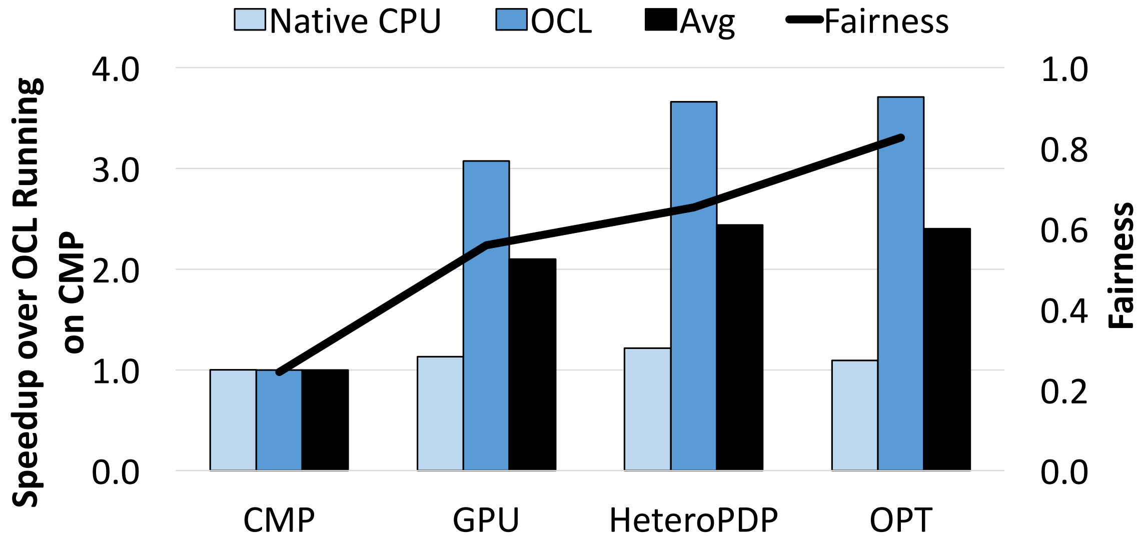

Figure 6.10: The speedup and fairness of HeteroPDP when an OpenCL application is *co-located* with a native CPU application. The label *Native CPU* represents native CPU workloads, *OCL* represents OpenCL workloads, and *Avg* is the average speedup across all co-located applications.

OpenCL application. This weight ratio is then taken into account when fairness of the system is calculated and thereby influencing the scheduling decision of the OpenCL application.

Figure 6.11 shows the execution target prediction accuracy evaluation for HeteroPDP with the weight ratio varying from 0.5 to 2.5. A weight ratio less than 1 indicates that the native CPU application has a lower priority than that of the OpenCL application, a weight ratio of 1.0 means all applications have an equal priority, and a weight ratio higher than 1.0 indicates that the native CPU application has a higher priority than that of the OpenCL application. As the importance of the native CPU application's speedup increases with a larger weight ratio, the optimal execution target for the OpenCL application increasingly switches to the GPU, as expected. HeteroPDP achieves a similarly good prediction accuracy of 75% for

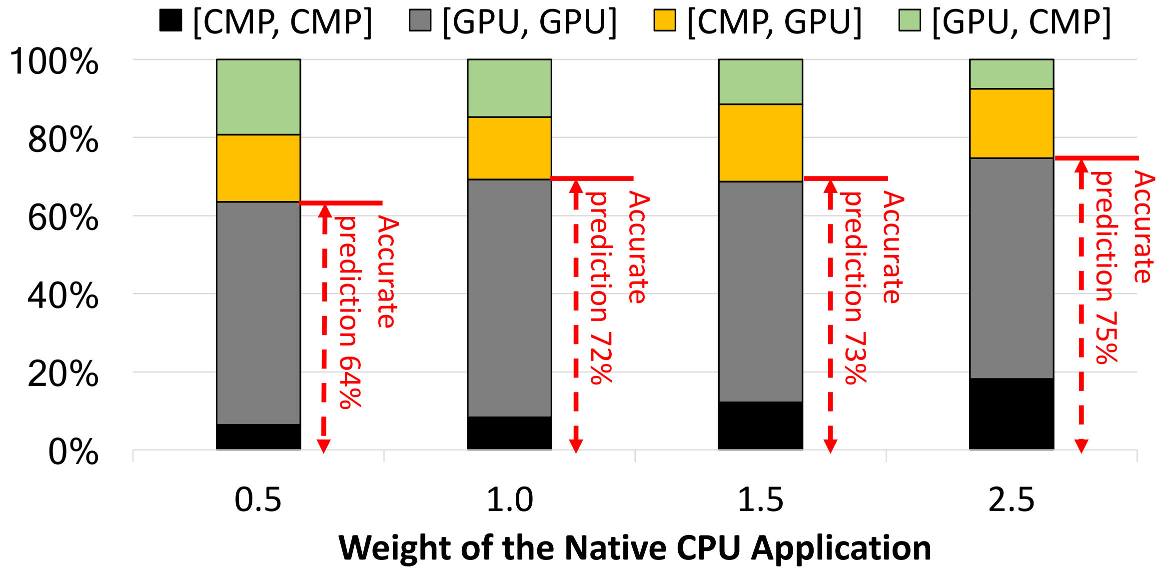

Figure 6.11: The prediction accuracy of selecting the optimal target to run an OpneCL kernel co-located with one native CPU application that has varying scheduling weights.

selecting the execution target. Figure 6.12 shows the corresponding system performance impact for HeteroPDP with varying scheduling priorities (weight ratios). As the native CPU application is given a heavier weight, its performance improvement becomes more important when maximizing the overall system throughput. We can observe that when the weight ratio is 0.5, the performance of OpenCL applications is lower than having equal weight (i.e., weight 1.0). This is because HeteroPDP's target prediction accuracy is slightly lower than with other weight ratios as shown in Figure 6.11. This also reflects upon the trend of weighted fairness improvement of the system.

6.5.4 HeteroPDP Scalability Analysis

Finally, I assess the scalability of the proposed design by increasing the number of native CPU applications on the four-core CMP. In this study, I co-locate two native

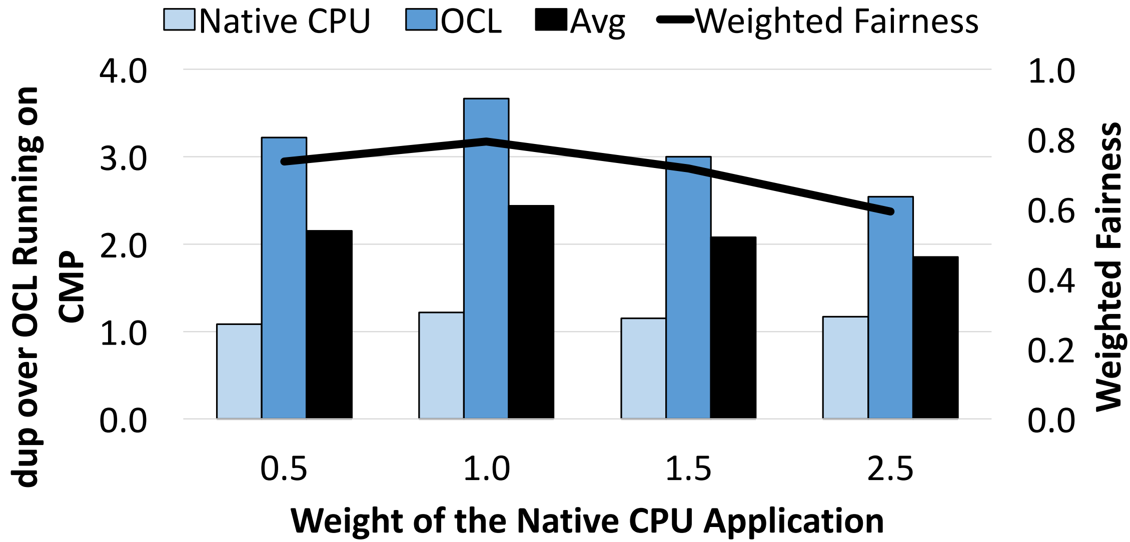

Figure 6.12: The speedup of HeteroPDP when running workloads consisting of one OpenCL application and one native CPU application with varying scheduling weights.

CPU application on the host CMP and evaluate the prediction trend of HeteroPDP for the OpenCL application. Figure 6.13 shows the prediction accuracy of the target device selection under such more resource-stressed execution environment. The evaluation result indicates that, although the number of co-located processes increases, HeteroPDP can still achieve a similarly good prediction accuracy of 70% as compared to the execution scenario with only one native CPU process (Figure 6.7). Similarly, the good execution target prediction accuracy translates into system throughput improvement for HeteroPDP. Figure 6.14 shows the respective speedup of the co-located applications as well as the system throughput and fairness results. HeteroPDP is able to continue its accurate execution target prediction without the need for prediction model revision and continues to mitigate the performance degradation in the *co-located* execution environment.

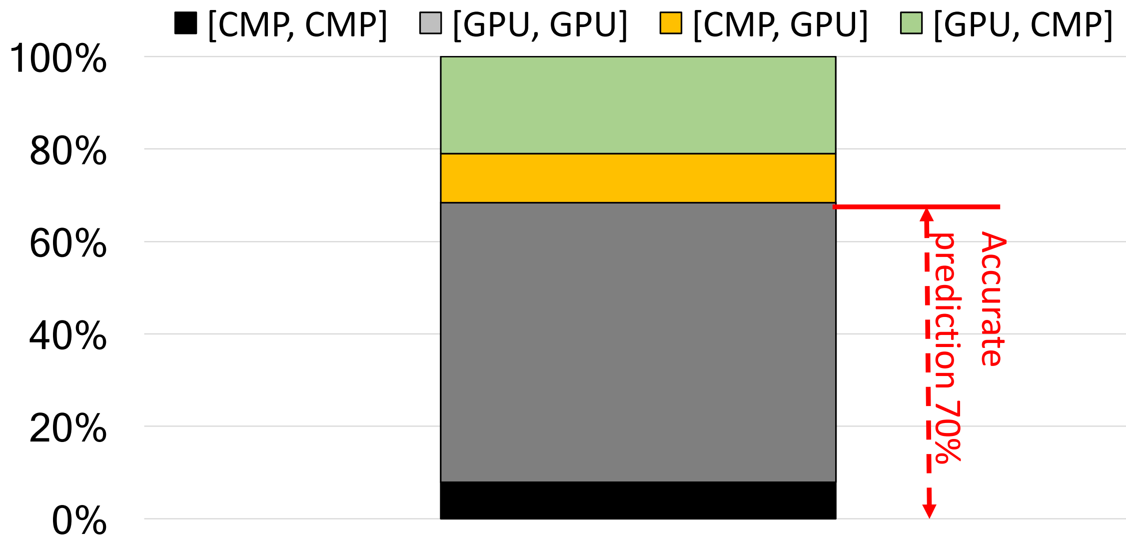

Figure 6.13: The prediction accuracy of selecting the optimal target device to run an OpneCL kernel *co-located* with two native CPU applications.

6.6 Chapter Summary

This chapter presents a detailed performance characterization study for the multi-programming heterogeneous computation environment. I show that the performance of an OpenCL application can be significantly affected by co-located native CPU applications and vice versa. Hence, a high-performing, robust OpenCL framework design should take the entire system utilization into account instead of only considering the characteristics of the OpenCL application.

In order to balance the performance degradation of a heterogeneous system, I develop a light-weight and scalable performance degradation predictor (HeteroPDP), based on simple regression models. HeteroPDP can accurately select the target execution device in a heterogeneous system to optimize and balance the performance degradation among all co-located workloads. HeteroPDP is designed and implemented within the existing OpenCL framework, and is evaluated on a real system

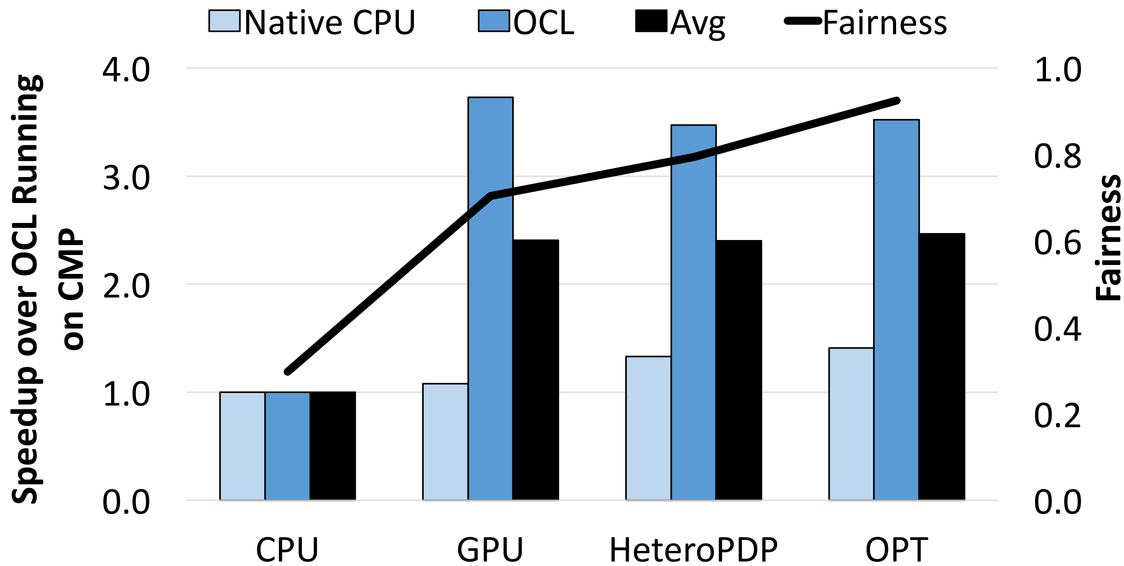

Figure 6.14: The speedup and fairness of HeteroPDP when running workloads consisting of two native CPU applications and one OpenCL application.

consisting of an Intel Core i7-3770 x86-64 CMP and an AMD FirePro GCN2.0 GPU. Overall, HeteroPDP improves the performance of OpenCL applications by 3x by intelligently selecting the execution target between the host CMP and the GPU while the *always offloading to GPU* decision produces 2.5x speedup. This bridges the performance gap between *always offloading to GPU* and the optimal target selection by 72%. This chapter shows that the simple regression model approach and the consideration of the multi-level memory interference in HeteroPDP can effectively improve the scheduling decision of OpenCL applications, leading to higher application performance and system throughput.

Table 6.1: Memory interference infrastructure setup and configurations.

Device	Configuration
Host CPU	Intel Core i7-3770 x86-64 CPU
	4 cores
	3.4GHz core frequency
	8MB shared last-level cache
	disabling turbo boost
	disabling hyperthreading
Host DRAM	DDR3-1600 24GB
	2 channels
	22GB/s max available bandwidth
Accelerator (GPU)	AMD FirePro S9150 GCN2.0 Hawaii GPU
	44 compute units (CUs)
	900 MHz core frequency
	PCIe 3.0 x16 8GT/s
GPU DRAM	GDDR5-1250 16GB with ECC
	512-bit width
	320GB/s max avaiable bandwidth
Software Runtime	Ubuntu 16.04
	Linux kernel v4.4.0
	clang/clang++ v3.8.0
	Intel PCM toolkit v2.11
	Intel OpenCL driver v1.2.0.18
	AMD OpenCL driver v2264.10

Table 6.2: CPU Workloads for the characterization studies and design evaluation. M and C stand for memory- and computation-intensive respectively.

Benchmark	Type	Suite
`bzip2`	C	SPEC2006 [40]
`calculix`	C	
`lbm`	M	
`mcf`	M	
`perlbench`	C	
`xalancbmk`	C	

Table 6.3: OpenCL workloads for the characterization studies and design evaluation. M and C stand for memory- and computation-intensive respectively.

Benchmark	Type	Suite	Benchmark	Type	Suite
`AC`	M	AMD SDK [5]	`BC`	M	Pannotia [21]
`BIN`	M		`CSE`	M	
`BS`	C		`ELL`	M	
`HIS`	M		`CFD`	M	Rodinia [19, 20]
`LUD`	C		`GAU`	M	
`MCA`	C		`HTW`	C	
`AES`	C	Hetero-Mark [108]	`KMN`	M	
`FIR`	C		`LEU`	C	
`KMN`	M		`PTH`	C	
`PR`	C		`SC`	M	
`BIT`	M	Intel SDK [46]	`S3D`	C	SHOC [28]
`GEMM`	M		`XSB`	M	XSBench [110]
`MF`	C				
`MC`	C				

Table 6.4: The OpenCL kernel features used for execution time prediction.

Feature	Category
# of scalar ALU instructions	Static features for predicting execution time on the CMP
# of scalar memory instructions	
# of vector ALU instructions	
# of vector memory instructions	
# of branch instructions	
# of atomic instructions	
# of memory instructions	Static features for predicting execution time on the GPU
# of integer instructions	
# of float-point instructions	
# of special math instructions	
# of branch instructions	
# of barrier instructions	
# of threads spawned	Dynamic features
size of memory buffer allocated	
last-level cache miss count	System utilization for predicting execution time of *co-located*
host DRAM bandwidth utilization	

Chapter 7

CONCLUSIONS

Employing a variety of hardware accelerators to improve system throughput and/or reduce energy consumption is the future trend of computer designs. In such kinds of heterogeneous computer systems, GPUs are a type of hardware accelerators that target at accelerating general-purpose parallel workloads by exploiting the massive multithreading computation paradigm. However, as presented in the prior chapters of this thesis, offloading parallel workloads onto a GPU does not always receive good performance benefits. To address the inefficiencies of GPU acceleration, my thesis delves into the microarchitecture as well as the system architecture of modern GPU designs to characterize the performance limits and propose practical solutions. Specifically, my thesis explores and solves three critical performance problems in modern GPU microarchitecture designs and heterogeneous systems with GPU accelerators.

In Chapter 3, I design a novel algorithm to characterize the latency hiding ability of the massive multithreading, throughput-oriented processors. With the latency breakdown algorithm, I show that the latency hiding ability is poor for many applications in GPUs. I then find out that warp criticality is one of the significant factors making the latency hiding ability sub-optimal. In Chapter 4, I further identify the root-causes of the warp criticality problem and propose an efficient solution, the *Criticality-Aware Warp Acceleration (CAWA)* mechanism, to dynamically mitigate the degree of warp criticality. By allocating larger execution time slices and reserving more cache storage for the critical warps, GPGPU workloads are able to receive higher performance gain.

While the design philosophy of GPU acceleration is to hide the operation latency by running a massive number of concurrent threads, it has got noticed that the large number of parallel threads can severely contend for the cache storage and interconnect bandwidth. In Chapter 5, I observe that the cache contention problem can be effectively alleviated by bypassing a portion of memory request and yielding the precious cache resource to the frequently referenced data. Based on this important observation, I designed a novel cache bypassing algorithm, the *Instruction-Aware Control Loop Based Cache Bypassing (Ctrl-C)* scheme, to maximize the GPU throughput by dynamically adjusting the bypassing aggressiveness per memory instruction. As a result, with a small circuit overhead, the performance of GPUs can be increased to a level similar to doubling the data cache capacity.

In addition to optimizing the GPU microarchitectures, in Chapter 6, I find that in a heterogeneous system, the GPU performance gain can be significantly affected by the memory interference at different levels of the memory hierarchy introduced by co-located applications. In order to maximize the system throughput and balance the execution time slowdown, I propose a light-weight and scalable *Performance Degradation Predictor for Heterogeneous Systems (HeteroPDP)* to dynamically evaluate the performance degradation of running an OpenCL kernel on different OpenCL enabled devices. With the high prediction accuracy of HeteroPDP for the execution target device, the overall system throughput and performance degradation of each co-located application can be optimized for heterogeneous computation environments.

Overall, my thesis breaks grounds as follows:

1. This thesis provides a new view to investigate the latency hiding ability and the inefficiencies of massive multithreading processors, specifically GPGPUs, by applying a novel latency breakdown algorithm.

2. This thesis presents a novel instruction granularity probabilistic cache bypassing design that can effectively mitigate the degree of L1 data cache contention in GPGPUs.

3. This thesis presents that the cache contention problem is a critical performance bottleneck for GPUs. The degree of cache contention, however, can be effectively mitigated by bypassing a portion of memory requests.

4. This thesis highlights the need of an intelligent OpenCL scheduling design which should consider the impacts of memory interference incurred by co-located applications on a multiprogramming heterogeneous computer system.

5. This thesis proposes architectural- and system-level solutions, i.e., CAWA [72], Ctrl-C [70], as well as HeteroPDP [71], to address the inefficiencies in modern GPU designs. With the proposed solutions, the performance gain of employing GPGPUs to accelerate computation can be further improved by an average of 1.23x (CAWA), 1.42x (Ctrl-C), and 3.0x (HeteroPDP).

In summary, my thesis gives new insights of future GPU designs in both microarchitectures and system architectures by offering detailed characterization and performance evaluation studies. With the essential studies of scheduling and memory management designs for GPUs, this thesis not only points towards a new direction for performance optimization, but also creates an ample space for future research in the computer architecture as well as the operating system domains.

REFERENCES

[1] Ashwin M. Aji, Antonio J. Peña, Pavan Balaji, and Wu-Chun Feng. Automatic command queue scheduling for task-parallel workloads in opencl. In *Proc. of the IEEE 2015 International Conference on Cluster Computing (CLUSTER'15)*, Chicago, IL, September 2015. doi: 10.1109/CLUSTER.2015.15.

[2] Ashwin M. Aji, Antonio J. Peña, Pavan Balaji, and Wu-Chun Feng. MultiCL. *Parallel Computing*, 58(C):37–55, October 2016. doi: 10.1016/j.parco.2016.05.006.

[3] Amazon. Overview of Amazon web services, December 2015. URL `https://d0.awsstatic.com/whitepapers/aws-overview.pdf`.

[4] AMD. AMD FirePro S9150 server GPU: The world's most powerful server GPU for high-performance computing, August 2014. URL `https://www.amd.com/Documents/firepro-s9150-datasheet.pdf`.

[5] AMD. AMD SDK—a complete development platform, March 2016.

[6] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proc. of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*, Waikiki, HI, December 2015. doi: 10.1145/2830772.2830780.

[7] ARM. big.LITTLE technology: the future of mobile, January 2013. URL `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Future_of_Mobile.pdf`.

[8] Akhil Arunkumar and Carole-Jean Wu. ReMAP: Reuse and memory access cost aware eviction policy for last level cache management. In *Proc. of the 32nd IEEE International Conference on Computer Design (ICCD'14)*, Seul, Korea, October 2014. doi: 10.1109/ICCD.2014.6974670.

[9] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proc. of the 39th IEEE/ACM International Symposium on Computer Architecture (ISCA'12)*, Portland, OR, June 2012. doi: 10.1145/2366231.2337207.

[10] Mihir Awatramani, Xian Zhu, Joseph Zambreno, and Diane Rover. Phase aware warp scheduling: Mitigating effects of phase behavior in GPGPU applications. In *in Proc. of the 2015 IEEE International Conference on Parallel Architecture and Compilation (PACT'15)*, San Francisco, CA, Octobor 2015. doi: 10.1109/PACT.2015.31.

[11] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE International Symposium on Analysis of Systems and Software (ISPASS'09)*, Boston, MA, April 2009. doi: 10.1109/ISPASS.2009.4919648.

[12] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, April 1966. doi: 10.1147/SJ.52.0078.

[13] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. CuMAS: Data transfer aware multi-application scheduling for shared GPUs. In *Proc. of the 2016 ACM International Conference on Supercomputing (ICS'16)*, Istanbul, Turkey, June 2016. doi: 10.1145/2925426.2926271.

[14] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proc. of the 36th IEEE/ACM International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, June 2009. doi: 10.1145/2485922.2485966.

[15] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proc. of the 40th IEEE/ACM International Symposium on Computer Architecture (ISCA'13)*, Tel-Aviv, Israel, June 2013.

[16] Alex Bourd. The OpenCL specification v2.2, May 2017.

[17] Qiong Cai, José Gonzàlez, Ryan Rakvic, Grigorios Magklis, pedro Chaparro, and Antonio Gonzàlez. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Proc. of the 17th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, Toronto, ON, Canada, October 2008. doi: 10.1145/1454115.1454149.

[18] Niladrish Chatterjee, Mike O'Connor, Donghyuk Lee, Daniel R. Johnson, Stephen W. Keckler, Minsoo Rhu, and William J. Dally. Architecting an energy-efficient dram system for gpus. In *Proc. of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, Austin, TX, February 2017. doi: 10.1109/HPCA.2017.58.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, Austin, TX, October 2009. doi: 10.1109/IISWC.2009.5306797.

[20] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proc. of the 2010 IEEE International Symposium on Workload Characterization (IISWC'10)*, Atlanta, GA, December 2010. doi: 10.1109/IISWC.2010.5650274.

[21] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *Proc. of the 2013 IEEE International Symposium on Workload Characterization (IISWC'13)*, Portland, OR, September 2013. doi: 10.1109/IISWC.2013.6704684.

[22] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. Adaptive cache management for energy-efficient GPU computing. In *Proc. of the 47th Annaul IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, United Kindom, December 2014. doi: 10.1109/MICRO.2014.11.

[23] Kyoshin Choo, William Panlener, and Byunghyun Jang. Understanding and optimizing GPU cache memory performance for compute workloads. In *Proc. of the 13th IEEE International Symposium on Parallel and Distributed Computing (ISPDC'14)*, Solan, India, June 2014. doi: 10.1109/ISPDC.2014.2.

[24] Micheal Collier and Robin Shahan. *Microsoft Azure Essential: Fundamentals of Azure*. Microsoft Press, 2 edition, September 2016.

[25] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *Proc. of the 51st IEEE/ACM Annaul Design Automation Conference (DAC'14)*, San Francisco, CA, June 2014. doi: 10.1145/2593069.2596667.

[26] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proc. of the 39th IEEE/ACM International Symposium on Computer Architecture (ISCA'12)*, Portland, OR, June 2012. doi: 10.1109/ISCA.2012.6237019.

[27] Hongwen Dai, Chao Li, Huiyang Zhou, Saurabh Gupta, Christos Kartsaklis, and Mike Mantor. A model-driven approach to warp/thread-block level GPU cache bypassing. In *Proc. of the 53nd IEEE/ACM Design Automation Conference (DAC'16)*, Austin, TX, June 2016. doi: 10.1145/2897937.2897966.

[28] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proc. of the GPGPU-3 General Purpose GPUs (GPGPU'10)*, Pittsburgh, PA, March 2010. doi: 10.1145/1735688.1735702.

[29] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarita, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proc. of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*, Vancouver, BC, Canada, December 2012. doi: 10.1109/MICRO.2012.43.

[30] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt. Parallel application memory scheduling. In *Proc. of the 44th Annual International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011. doi: 10.1145/2155620.2155663.

[31] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008. doi: 10.1109/MM.2008.44.

[32] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient SIMT control flow. In *Proc. of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, San Francisco, CA, February 2011. doi: 10.1109/HPCA.2011.5749714.

[33] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 37th IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010. doi: 10.1109/MICRO.2007.30.

[34] Vìctor García, Juan Gómez-Luna, Thomas Grass, Alejandro Rico, Eduard Ayguade, and Antonio J. Peña. Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. In *Proc. of the 2016 IEEE International Symposium on Workload Characterization (IISWC'16)*, Providence, RI, September 2016. doi: 10.1109/IISWC.2016.7581277.

[35] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindoholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proc. of the 38th IEEE/ACM International Symposium on Computer Architecture (ISCA'11)*, San Jose, CA, June 2011. doi: 10.1145/2000064.2000093.

[36] Google. Google cloud platform overview, January 2017. URL `https://cloud.google.com/docs/overview`.

[37] Cris Gregg and Kim Hazlewood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proc. of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'11)*, Austin, TX, April 2011. doi: 10.1109/ISPASS.2011.5762730.

[38] Tae Jun Ham, Juan L. Aragón, and Mararret Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proc. of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*, Waikiki, HI, December 2015. doi: 10.1145/2830772.2830800.

[39] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo, and Tuyong Wang. Mars: A MapReduce framework on graphics processors. In *Proc. of the 17th IEEE/ACM International Conference on Parallel Architectures and*

Compilation Techniques (PACT'08), Toronto, ON, Canada, October 2008. doi: 10.1145/1454115.1454152.

[40] John Henning. Spec cpu2006, August 2006. URL `http://www.spec.org/cpu2006/Docs/readme1st.html`.

[41] Joel Hestness, Stephen W. Keckler, and David A. Wood. A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior. In *Proc. of the 2014 IEEE International Symposium on Workload Characterization (IISWC'14)*, Raleigh, NC, October 2014. doi: 10.1109/IISWC.2014.6983054.

[42] Joel Hestness, Stephen W. Keckler, and David A. Wood. GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors. In *Proc. of the 2015 IEEE International Symposium on Workload Characterization (IISWC'15)*, Atlanta, GA, September 2015. doi: 10.1109/IISWC.2015.15.

[43] Intel. AGP Pro specification Rev. 1.1a, April 1999. URL `https://web.archive.org/web/20030621113346/http://www.agpforum.org:80/downloads/apro_r11a.pdf`.

[44] Intel. Intel Core i7-2600 processor, January 2011. URL `http://ark.intel.com/products/52213`.

[45] Intel. Intel Core i7-3770 processor, April 2012. URL `https://ark.intel.com/products/65719/Intel-Core-i7-3770-Processor-8M-Cache-up-to-3_90-GHz`.

[46] Intel. Intel sdk for opencl applications, June 2016. URL `https://software.intel.com/en-us/intel-opencl`.

[47] Ravi Iyer. Accelerator-rich architectures: Implications, opportunities and challenges. In *Proc. of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC'12)*, Sydney, Australia, February 2012. doi: 10.1109/ASPDAC.2012.6164927.

[48] Aamer Jaleel, William hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*, Toronto, ON, Canada, October 2008. doi: 10.1145/1454115.1454145.

[49] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. of the 37th IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010. doi: 10.1145/1815961.1815971.

[50] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely Jr., and Joel Emer. CRUISE: Cache replacement and utility-aware

scheduling. In *Proc. of the ACM 17th International Conference on Architectural Suport for Programming Languages and Operating Systems (ASPLOS'12)*, London, United Kindom, March 2012. doi: 10.1145/2189750.2151003.

[51] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proc. of the 26th ACM International Conference on Supercomputing (ICS'12)*, Venice, Italy, June 2012. doi: 10.1145/2304576.2304582.

[52] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proc. of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*, Orlando, FL, February 2014. doi: 10.1109/HPCA.2014.6835938.

[53] Daniel A. Jiménez. Insertion and promotion for tree-based pseudoLRU last-level caches. In *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, Davis, CA, December 2013. doi: 10.1145/2540708.2540733.

[54] Adwait Jog, Onur Kayıran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for GPGPUs. In *Proc. of the 40th IEEE/ACM International Symposium on Computer Architecture (ISCA'13)*, Tel-Aviv, Israel, June 2013. doi: 10.1145/2485922.2485951.

[55] Adwait Jog, Onur Kayıran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proc. of the 18th IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, Houston, TX, March 2013. doi: 10.1145/2451116.2451158.

[56] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proc. of the 44th IEEE/ACM International Symposium*

on Computer Architecture (ISCA'17), Toronto, ON, Canada, June 2017. doi: 10.1145/3079856.3080246.

[57] Onur Kayıran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. Managing GPU concurrency in heterogeneous architectures. In *Proc. of the 47th Annaul IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, United Kindom, December 2014. doi: 10.1109/MICRO.2014.62.

[58] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *Proc. of the 25th IEEE International Conference on Computer Design (ICCD'07)*, Lake Tahoe, CA, October 2007. doi: 10.1109/ICCD.2007.4601909.

[59] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *Proc. of the 2009 International Symposium on Workload Characterization (IISWC'09)*, Austin, TX, October 2009. doi: 10.1109/IISWC.2009.5306801.

[60] Mahmoud Khairy, Mohamed Zahran, and Amr G. Wassal. Efficient utilization of GPGPU cache hierarchy. In *Proc. of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15)*, San Francisco, CA, February 2015. doi: 10.1145/2716282.2716291.

[61] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *Proc. of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*, Atlanta, GA, December 2010. doi: 10.1109/MICRO.2010.24.

[62] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. Access pattern-aware cache management for improving data utilization in GPU. In *Proc. of the 44th IEEE/ACM International Symposium on Computer Architecture (ISCA'17)*, Toronto, ON, Canada, June 2017. doi: 10.1145/3079856.3080239.

[63] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. of the 28th IEEE/ACM International Symposium on Computer Architecture (ISCA'01)*, Göthenburg, Sweden, June 2001. doi: 10.1145/384285.379259.

[64] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proc. of the 18th IEEE/ACM International Symposium on High Performance Computer Architecture (HPCA'12)*, New Orleans, LA, February 2012. doi: 10.1109/HPCA.2012.6168947.

[65] Jaekyu Lee, Dong Hyuk Woo, Hyesoon Kim, and Mani Azimi. GREEN cache: Exploiting the disciplined memory model of OpenCL on GPUs. *IEEE Transactions on Computers*, 64(11):3167–3180, January 2015. doi: 10.1109/TC.2015.2395435.

[66] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. Vast: the illusion of a large memory space for GPUs. In *Proc. of the 23rd IEEE/ACM International Conference on Prallel Architectures and Compilation (PACT'14)*, Edmonton, AB, Canada, August 2014. doi: 10.1145/2628071.2628075.

[67] Minseok Lee, Gwangsun Kim, John Kim, Woong Seo, Yeongon Cho, and Soo-jung Ryu. iPAWS: Instruction-issue pattern-based adaptive warp scheduling for GPGPUs. In *Proc. of the 22nd IEEE/ACM International Symposium on High Performance Computer Architecture (HPCA'16)*, Barcelona, Spain, March 2016. doi: 10.1109/HPCA.2016.7446079.

[68] Shin-Ying Lee and Carole-Jean Wu. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proc. of the 23rd IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*, Edmonton, AB, Canada, August 2014. doi: 10.1145/2628071.2628107.

[69] Shin-Ying Lee and Carole-Jean Wu. Characterizing the latency hiding ability of GPUs. In *Proc. of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*, Monterey, CA, March 2014. doi: 10.1109/ISPASS.2014.6844477.

[70] Shin-Ying Lee and Carole-Jean Wu. Ctrl-C: Instruction-aware control loop based adaptive cache bypassing for GPUs. In *Proc. of the 34th IEEE International Conference on Computer Design (ICCD'16)*, Phoenix, AZ, October 2016. doi: 10.1109/ICCD.2016.7753271.

[71] Shin-Ying Lee and Carole-Jean Wu. Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference. In *Proc. of the 2017 IEEE International Symposium on Workload Characterization (IISWC'17)*, Seattle, WA, October 2017.

[72] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. In *Proc. of the 42nd IEEE/ACM International Symposium on Computer Architecture (ISCA'15)*, Portland, OR, June 2015. doi: 10.1145/2749469.2750418.

[73] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and transparent cache bypassing for GPUs. In *Prof. of the 2015 IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, Austin, TX, November 2015. doi: 10.1145/2807591.2807606.

[74] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. Locality-driven dynamic GPU cache bypassing. In *Proc. of the 29th ACM International Conference on Supercomputing (ICS'15)*, Irvine, CA, June 2015. doi: 10.1145/2751205.2751237.

[75] Jian LI, José F. Martínez, and Micheal C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proc. of the 10th*

IEEE International Symposium on High Performance Computer Architecture (HPCA'04), Madrid, Spain, February 2004. doi: 10.1109/HPCA.2004.10018.

[76] Yun Liang, Yu Wang, and Guangyu Sun. Coordinated static and dynamic cache bypassing for GPUs. In *Proc. of the 21st IEEE Symposium on High Performance Computer Architecture (HPCA'15)*, San Francisco, CA, February 2015. doi: 10.1109/HPCA.2015.7056023.

[77] Yun Liang, Xiaolong Xie, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1677–1690, April 2015. doi: 10.1109/TCAD.2015.2424962.

[78] Jun Min Lin, Yu Chen, Wenlong Li, Zhao Tang, and Aamer Jaleel. Memory characterization of SPEC CPU2006 benchmark suite. In *Workshop for the 11th Computer Architecture Evaluation of Commerical Workloads (CAECW'08)*, Salt Lake City, UT, February 2008.

[79] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2): 39–55, March 2008. doi: 10.1109/MM.2008.31.

[80] Chun Liu, Anand Sivasubramaniam, Mahmut Kandemir, and Mary J. Irwin. Exploiting barriers to optimize power consumption of CMPs. In *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, April 2005. doi: 10.1109/IPDPS.2005.211.

[81] Daniel Lustig and Margaret Martonosi. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *Proc. of the 19th IEEE/ACM International Symposium on High Performance Computer Architecture (HPCA'13)*, Shenzhen, China, February 2013. doi: 10.1109/HPCA.2013.6522332.

[82] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011. doi: 10.1145/2155620.2155650.

[83] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proc. of the 22nd IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*, Edinburgh, Scotland, October 2013. doi: 10.1109/PACT.2013.6618819.

[84] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proc. of the 37th IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010. doi: 10.1145/1815961.1815992.

[85] Sparsh Mittal. A survey of cache bypassing techniques. *MDPI Journal of Low Power Electronics and Applications*, 6(2):5, April 2016. doi: 10.3390/jlpea6020005.

[86] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of the 40th Annual IEEE/ACM International Smposium on Microarchitecture (MICRO'07)*, Chicago, IL, December 2007. doi: 10.1109/MICRO.2007.21.

[87] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *Proc. of the 44th Annual International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011. doi: 10.1145/2155620.2155656.

[88] NVIDIA. NVIDIA compute PTX: Parallel thread execution ISA v1.4, March 2009. URL `http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf`.

[89] NVIDIA. NVIDIA GeForce GTX 480/470/465 GPU datasheet, March 2010. URL `http://www.nvidia.co.uk/docs/IO/90201/GTX-480-470-Web-Datasheet-Final4.pdf`.

[90] NVIDIA. CUDA C/C++ SDK code samples v4.0, May 2011. URL `http://docs.nvidia.com/cuda/cuda-samples`.

[91] NVIDIA. NVIDIA CUDA C programming guide v4.2, April 2012. URL `http://developer.nvidia.com/nvidia-gpu-computing-documentation`.

[92] NVIDIA. NVIDIA GeForce GTX 750 Ti: Featuring first-generation Maxwell GPU technology, designed for extreme performance per watt, February 2014. URL `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf`.

[93] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proc. of the ACM 18th International Conference on Architectural Suport for Programming Languages and Operating Systems (ASPLOS'13)*, Houston, TX, March 2013. doi: 10.1145/2451116.2451160.

[94] PCI-SIG. PCI Express M.2 specification Rev. 1.1, December 2016. URL `http://pcisig.com/specifications/pciexpress/`.

[95] Richard R. Picard and R. Dennis Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, March 1984. doi: 10.1080/01621459.1984.10478083.

[96] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Orlando, FL, December 2006. doi: 10.1109/MICRO.2006.49.

[97] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. Adaptive insertion policies for high performance caching. In *Proc. of the 34th IEEE/ACM International Symposium on Computer Architecture (ISCA'07)*, San Diego, CA, June 2007. doi: 10.1145/1250662.1250709.

[98] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. Set-dueling-controlled adaptive insertion for high-performance caching. *IEEE Micro*, 28(1):91–98, January 2008. doi: 10.1109/MM.2008.14.

[99] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient GPU control flow. In *Proc. of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*, Shenzhen, China, February 2013. doi: 10.1109/HPCA.2013.6522352.

[100] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proc. of the 46th Annaul IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, Davis, CA, December 2013. doi: 10.1145/2540708.2540717.

[101] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *Proc. of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*, Vancouver, BC, Canada, December 2012. doi: 10.1109/MICRO.2012.16.

[102] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Divergence-aware warp scheduling. In *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*, Davis, CA, December 2013. doi: 10.1145/2540708.2540718.

[103] Dongyou Seo, Hyeonsang Eom, and Heon Y. Yeom. MLB: A memory-aware load balancing for mitigating memory contention. In *Proc. of the 2014 Conference on Timely Results in Operating Systems (TRIOS'14)*, Broomfield, CO, October 2014.

[104] Ankit Sethia and Scott Mahlke. Equalizer: Dynamic tuning of GPU resources for efficient execution. In *Proc. of the 47th Annaul IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, United Kindom, December 2014. doi: 10.1109/MICRO.2014.16.

[105] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and soc interfaces using gem5-Aladdin. In *Proc. of the 49th Annual IEEE/ACM Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, October 2016. doi: 10.1109/MICRO.2016.7783751.

[106] John A. Stratton, Nasser Anssari, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Daniel Liu, and Wen mei Hwu. Optimization and architecture effects on GPU computing workload performance. In *Proc. of the IEEE Innovative Parallel Computing (InPar'12)*, San Jose, CA, May 2012. doi: 10.1109/InPar.2012.6339605.

[107] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W. Hwu. The Parboil technical report, March 2012.

[108] Yifan Sun, Xian Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Vilegas, and David Kaeli. Hetero-Mark, a benchmark suite for CPU-GPU collaborative computing. In *Proc. of the 2016 IEEE International Symposium on Workload Characterization (IISWC'16)*, Providence, RI, September 2016. doi: 10.1109/IISWC.2016.7581262.

[109] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Bechmann, and Daniel A. Jiménez. Adaptive GPU cache bypassing. In *Proc. of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15)*, San Francisco, CA, February 2015. doi: 10.1145/2716282.2716283.

[110] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. XSBench — the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *Proc. of the 2014 International Conference on the Role of Reactor Physics toward a Sustainable Future (PHYSOR'14)*, Kyoto, Japan, September 2014. doi: 10.11484/jaea-conf-2014-003.

[111] Yatish Turakhia, Guangshuo Liu, Siddharth Garg, and Diana Marculescu. Thread progress equalization: Dynamically adaptive power and performance optimization of multi-threaded applications. *IEEE Transactions on Computers*, 66(4):731–799, April 2017. doi: 10.1109/TC.2016.2608951.

[112] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Proc. of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'16)*, Uppsala, Sweden, April 2016. doi: 10.1109/ISPASS.2016.7482091.

[113] Ruisheng Wang and Lizhong Chen. Futility scaling: High-associativity cache partitioning. In *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, United Kindom, December 2014. doi: 10.1109/MICRO.2014.46.

[114] Yuan Wen and Michael F.P. O'Boyle. Merge or separate? multi-job scheduling for OpenCL kernels on CPU/GPU platforms. In *Proc. of the GPGPU-10 General Purpose GPUs (GPGPU'17)*, Austin, TX, February 2017. doi: 10.1145/3038228.3038235.

[115] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *Proc. of the 21st IEEE International Conference on High Performance Computing (HiPC'14)*, Kalamazoo, MI, December 2014. doi: 10.1109/HiPC.2014.7116910.

[116] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel performance counter monitor – a better way to measure CPU utilization, January 2017. URL `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

[117] Carole-Jean Wu and Margaret Martonosi. Adaptive timekeeping replacement: Fine-grained capacity management for shared CMP caches. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(2):3:1–3:26, April 2011. doi: 10.1145/1952998.1953001.

[118] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Proc. of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'11)*, Austin, TX, April 2011. doi: 10.1109/ISPASS.2011.5762710.

[119] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proc. of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*, Porto Alegre, Brazil, December 2011. doi: 10.1145/2155620.2155671.

[120] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. GPGPU performance and power estimation using machine learning. In *Proc. of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*, San Francisco, CA, February 2015. doi: 10.1109/HPCA.2015.7056063.

[121] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on GPUs. In *Proc. of the 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*, San Jose, CA, November 2013. doi: 10.1109/ICCAD.2013.6691165.

[122] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. of the 36th IEEE/ACM International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, June 2009. doi: 10.1145/1555815.1555778.

[123] Mohamed Zahran. Heterogeneous computing: Here to stay. *ACM Queue*, 14 (6):40, November 2016. doi: 10.1145/3028687.3038873.

[124] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. Towards high performance paged memory for GPUs. In *Proc. of the 22nd IEEE/ACM International Symposium on High Performance Computer Architecture (HPCA'16)*, Barcelona, Spain, March 2016. doi: 10.1109/HPCA.2016.7446077.

[125] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Adaptive cache and concurrency allocation on GPGPUs. *IEEE Computer Architecture Letters*, 14(2): 90–93, September 2014. doi: 10.1109/LCA.2014.2359882.

APPENDIX A

REGRESSION MODELS AND COEFFICIENTS FOR HETEROPDP

This appendix shows the coefficients used for the regression models in the HeteroPDP design. Note that, to better correlate each factor, the regression models are trained as the interactive mode. Therefore, each term in the regression model is a product of two factors as shown in Equations A.1 and A.2. The coefficients of each term are listed in Tables A.1 to A.4

$$term_i = factor_1_i \times factor_2_i \tag{A.1}$$

$$Time = \sum_i coefficient_i \times term_i = \sum_i coefficient_i \times factor_1_i \times factor_2_i \tag{A.2}$$

Table A.1: Coefficients for predicting OpenCL kernel execution time *alone* on the Intel Core i7-3770 CMP

Term		Coefficient
Factor_1	Factor_2	
# of threads	1	0.000000009173792
# of scalar ALU inst.	1	-0.001291950266492
# of scalar MEM inst.	1	-0.000708449667275
# of vector ALU inst.	1	-0.005027805769920
# of vector MEM inst.	1	0.001527459744997
# of branch inst.	1	0.020511412895212
# of atomic inst.	1	0
buffer size	1	0.000000001752382
# of threads	# of scalar ALU inst.	-0.000000000858425
# of threads	# of scalar MEM inst.	0.000000000774906
# of threads	# of vector ALU inst.	-0.000000001312955
# of threads	# of vector MEM inst.	0.000000002987590
# of threads	# of branch inst.	0
# of threads	# of atomic inst.	0.000000001526019
# of threads	buffer size	-0.000000705248815
# of scalar ALU inst.	# of scalar MEM inst.	0.000224981312450
# of scalar ALU inst.	# of vector ALU inst.	-0.000349975851307
# of scalar ALU inst.	# of vector MEM inst.	-0.000291671393497
# of scalar ALU inst.	# of branch inst.	0
# of scalar ALU inst.	# of atomic inst.	0.000714793051486
# of scalar ALU inst.	buffer size	-0.000000829173211
# of scalar MEM inst.	# of vector ALU inst.	0.000120515879912
# of scalar MEM inst.	# of vector MEM inst.	0.000231489317965
# of scalar MEM inst.	# of branch inst.	0
# of scalar MEM inst.	# of atomic inst.	0.000223155952517

Continued on next page

Table A.1 – continued from previous page

Term		Coefficient
Factor_1	Factor_2	
# of scalar MEM inst.	buffer size	0.000066733510292
# of vector ALU inst.	# of vector MEM inst.	-0.000364289913766
# of vector ALU inst.	# of branch inst.	0
# of vector ALU inst.	# of atomic inst.	0.002190538539154
# of vector ALU inst.	buffer size	-0.000000013133745
# of vector MEM inst.	# of branch inst.	0
# of vector MEM inst.	# of atomic inst.	-0.001299923605429
# of vector MEM inst.	buffer size	0.000576849602103
# of branch inst.	# of atomic inst.	-0.009194662724052
# of branch inst.	buffer size	0
# of atomic inst.	buffer size	0.000000000044371

Table A.2: Coefficients for predicting OpenCL kernel execution time *alone* on the AMD FirePro S9150 GPU

Term		Coefficient
Factor_1	Factor_2	
# of threads	1	-0.000000461723788
# of MEM inst.	1	-0.000004961109670
# of INT inst.	1	-0.001555207427860
# of FP inst.	1	0.007097661815404
# of math inst.	1	0
# of branch inst.	1	0
# of barrier inst.	1	0
buffer size	1	0.000000004301985
# of threads	# of MEM inst.	-0.000000000162581
# of threads	# of INT inst.	0.000000000012263
# of threads	# of FP inst.	0.000000000278928
# of threads	# of math inst.	-0.000000843926919
# of threads	# of branch inst.	0.000000000255150
# of threads	# of barrier inst.	-0.000000001623380
# of threads	buffer size	-0.000000000051832
# of MEM inst.	# of INT inst.	0.000000605781747
# of MEM inst.	# of FP inst.	0.000143796621602
# of MEM inst.	# of math inst.	0
# of MEM inst.	# of branch inst.	0.000014196556354
# of MEM inst.	# of barrier inst.	-0.001386335986034
# of MEM inst.	buffer size	0.000000000706951
# of INT inst.	# of FP inst.	0.000013853357393
# of INT inst.	# of math inst.	-0.000724741599509
# of INT inst.	# of branch inst.	0.000006526006563
# of INT inst.	# of barrier inst.	-0.000300708162003

Continued on next page

Table A.2 – continued from previous page

Term		Coefficient
Factor_1	Factor_2	
# of INT inst.	buffer size	0
# of FP inst.	# of math inst.	0.000059471218335
# of FP inst.	# of branch inst.	-0.001039671674845
# of FP inst.	# of barrier inst.	0.001405577586221
# of FP inst.	buffer size	-0.000000005460802
# of math inst.	# of branch inst.	0
# of math inst.	# of barrier inst.	0
# of math inst.	buffer size	0
# of branch inst.	# of barrier inst.	0.003844312920723
# of branch inst.	buffer size	0
# of barrier inst.	buffer size	-0.000000000378301

Table A.3: Coefficients for predicting OpenCL kernel execution time *co-located* on the Intel Core i7-3770 CMP

Term		Coefficient
Factor_1	Factor_2	
# of threads	1	0.000000464380971
# of scalar ALU inst.	1	0
# of scalar MEM inst.	1	0
# of vector ALU inst.	1	0
# of vector MEM inst.	1	0
# of branch inst.	1	0
# of atomic inst.	1	0
buffer size	1	0.000000000321057
LLC miss count	1	-0.000000009308307
DRAM bandwidth	1	0
# of threads	# of scalar ALU inst.	0.000000045932829
# of threads	# of scalar MEM inst.	-0.000000037158650
# of threads	# of vector ALU inst.	-0.000000012339376
# of threads	# of vector MEM inst.	0.000000033186243
# of threads	# of branch inst.	0.000000050576133
# of threads	# of atomic inst.	0
# of threads	buffer size	-0.000000034050131
# of threads	LLC miss count	0
# of threads	DRAM bandwidth	0.000000000284800
# of scalar ALU inst.	# of scalar MEM inst.	0.000583138870637
# of scalar ALU inst.	# of vector ALU inst.	-0.010686656463813
# of scalar ALU inst.	# of vector MEM inst.	0.017553583556379
# of scalar ALU inst.	# of branch inst.	0
# of scalar ALU inst.	# of atomic inst.	0
# of scalar ALU inst.	buffer size	0

Continued on next page

Table A.3 – continued from previous page

Term		Coefficient
Factor_1	Factor_2	
# of scalar ALU inst.	LLC miss count	-0.000000000979732
# of scalar ALU inst.	DRAM bandwidth	-0.005239040745939
# of scalar MEM inst.	# of vector ALU inst.	0.004980999755519
# of scalar MEM inst.	# of vector MEM inst.	-0.006079469682139
# of scalar MEM inst.	# of branch inst.	-0.001821087047881
# of scalar MEM inst.	# of atomic inst.	0
# of scalar MEM inst.	buffer size	0.000000058302141
# of scalar MEM inst.	LLC miss count	0.000000000576634
# of scalar MEM inst.	DRAM bandwidth	0.003039032020751
# of vector ALU inst.	# of vector MEM inst.	0.000111461903008
# of vector ALU inst.	# of branch inst.	0.006308844739137
# of vector ALU inst.	# of atomic inst.	0
# of vector ALU inst.	buffer size	0
# of vector ALU inst.	LLC miss count	-0.000000000043673
# of vector ALU inst.	DRAM bandwidth	-0.000216858477800
# of vector MEM inst.	# of branch inst.	-0.018731831118366
# of vector MEM inst.	# of atomic inst.	0
# of vector MEM inst.	buffer size	0.000000007754859
# of vector MEM inst.	LLC miss count	0.000000000162284
# of vector MEM inst.	DRAM bandwidth	0.000762235532871
# of branch inst.	# of atomic inst.	0
# of branch inst.	buffer size	0
# of branch inst.	LLC miss count	-0.000000000087090
# of branch inst.	DRAM bandwidth	0.000546133983535
# of atomic inst.	buffer size	-0.000000000115041
# of atomic inst.	LLC miss count	-0.000000003697363
# of atomic inst.	DRAM bandwidth	-0.023775141869708
buffer size	LLC miss count	-0.000000000923101
buffer size	DRAM bandwidth	-0.000000007331325
LLC miss count	DRAM bandwidth	0.000000002764534

Table A.4: Coefficients for predicting OpenCL kernel execution time *co-located* on the AMD FirePro S9150 GPU

Term		Coefficient
Factor_1	Factor_2	
# of threads	1	-0.000000461723788
# of MEM inst.	1	-0.000004961109670
# of INT inst.	1	-0.001555207427860
# of FP inst.	1	0.007097661815404
# of math inst.	1	0
# of branch inst.	1	0

Continued on next page

Table A.4 – continued from previous page

Term		Coefficient
Factor_1	Factor_2	
# of barrier inst.	1	0
buffer size	1	0.000000004301985
DRAM bandwidth	1	0.0000000004
# of threads	# of MEM inst.	-0.000000000162581
# of threads	# of INT inst.	0.000000000012263
# of threads	# of FP inst.	0.000000000278928
# of threads	# of math inst.	-0.000000843926919
# of threads	# of branch inst.	0.000000000255150
# of threads	# of barrier inst.	-0.000000001623380
# of threads	buffer size	-0.000000000051832
# of threads	DRAM bandwidth	0
# of MEM inst.	# of INT inst.	0.000000605781747
# of MEM inst.	# of FP inst.	0.000143796621602
# of MEM inst.	# of math inst.	0
# of MEM inst.	# of branch inst.	0.000014196556354
# of MEM inst.	# of barrier inst.	-0.001386335986034
# of MEM inst.	buffer size	0.000000000706951
# of MEM inst.	DRAM bandwidth	0
# of INT inst.	# of FP inst.	0.000013853357393
# of INT inst.	# of math inst.	-0.000724741599509
# of INT inst.	# of branch inst.	0.000006526006563
# of INT inst.	# of barrier inst.	-0.000300708162003
# of INT inst.	buffer size	0
# of INT inst.	DRAM bandwidth	0
# of FP inst.	# of math inst.	0.000059471218335
# of FP inst.	# of branch inst.	-0.001039671674845
# of FP inst.	# of barrier inst.	0.001405577586221
# of FP inst.	buffer size	-0.000000005460802
# of FP inst.	DRAM bandwidth	0
# of math inst.	# of branch inst.	0
# of math inst.	# of barrier inst.	0
# of math inst.	buffer size	0
# of math inst.	DRAM bandwidth	0
# of branch inst.	# of barrier inst.	0.003844312920723
# of branch inst.	buffer size	0
# of branch inst.	DRAM bandwidth	0
# of barrier inst.	buffer size	-0.000000000378301
# of barrier inst.	DRAM bandwidth	0
buffer size	DRAM bandwidth	0.000000000370901