The Pennsylvania State University
The Graduate School

DESIGN AND ANALYSIS OF SCHEDULING TECHNIQUES FOR THROUGHPUT PROCESSORS

A Dissertation in
Computer Science and Engineering
by
Adwait Jog

© 2015 Adwait Jog

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2015

The dissertation of Adwait Jog was reviewed and approved* by the following:

Chita R. Das
Distinguished Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Mahmut T. Kandemir
Professor of Computer Science and Engineering
Dissertation Co-Advisor

Yuan Xie
Professor of Computer Science and Engineering

W. Kenneth Jenkins
Professor of Electrical Engineering Department

Ravishankar Iyer
Senior Principal Engineer, Intel Labs
Special Member

Onur Mutlu
Associate Professor of Electrical and Computer Engineering
Carnegie Mellon University
Special Member

Lee D. Coraor
Associate Professor of Computer Science and Engineering
Director of Academic Affairs

*Signatures are on file in the Graduate School.

Abstract

Throughput Processors such as Graphics Processing Units (GPUs) are becoming an inevitable part of every computing system because of their ability to accelerate applications consisting of abundant parallelism. They are not only used to accelerate big data analytics in cloud data centers or high-performance computing (HPC) systems, but are also employed in mobile and wearable devices for efficient execution of multimedia rich applications and smooth rendering of display. In spite of the highly parallel structure of GPUs and their ability to execute multiple threads concurrently, they are far from achieving their theoretically achievable peak performance. This is attributed to several reasons such as contention for limited shared resources (e.g., caches and memory), high control-flow divergence, and limited off-chip memory bandwidth. Another reason for the low utilization and subpar performance is that the current GPUs are not well-equipped to efficiently and fairly execute multiple applications concurrently, potentially originating from different users. This dissertation is focused on managing contention in GPUs for shared cache and memory resources caused by concurrently executing threads. This contention causes severe loss in performance, fairness, locality, and parallelism. To manage this contention, this dissertation proposes techniques that are employed at two different places: *core* and *memory*. First, this dissertation shows that by intelligently scheduling the threads at the *core*, the generated memory request patterns can be more amenable for existing resource management techniques such as cache replacement and memory scheduling as well as performance enhancement techniques such as data prefetching. Second, this dissertation shows that considering criticality and other application characteristics to schedule memory requests at the memory controller is an effective way to manage contention at the *memory*.

Table of Contents

List of Figures x

List of Tables xv

Acknowledgments xvii

Chapter 1
Introduction 1

Chapter 2
Graphics Processing Units (GPUs): A Primer 7

Chapter 3
Cache and Memory Aware Warp Scheduling Techniques 12

3.1 Introduction 13

3.2 Motivation and Workload Analysis 16

3.3 The OWL Scheduler 18

3.3.1 *CTA-Aware*: CTA-aware two-level warp scheduling 18

3.3.2 *CTA-Aware-Locality*: Locality aware warp scheduling 20

3.3.3 *CTA-Aware-Locality-BLP*: BLP aware warp scheduling . . . 24

3.3.4 Opportunistic Prefetching 28
3.3.5 Hardware Overheads 30
3.4 Experimental Methodology 31
3.4.1 Workloads and Metrics 31
3.5 Experimental Results 32
3.5.1 Performance Results 33
3.5.2 Sensitivity Studies 36
3.6 Related Work 37
3.7 Chapter Summary 40

Chapter 4
Prefetch Aware Warp Scheduling Techniques **41**

4.1 Introduction 42
4.2 Interaction of Scheduling and Prefetching: Motivation and Basic Ideas 46
4.2.1 Shortcomings of the State-of-the-Art Warp Schedulers 47
4.2.1.1 Round-robin (RR) warp scheduling 47
4.2.1.2 Round-robin (RR) warp scheduling and inter-warp prefetching 48
4.2.1.3 Two-level (TL) warp scheduling 49
4.2.1.4 Two-level (TL) warp scheduling and intra-fetch-group prefetching 50
4.2.1.5 Two-level (TL) warp scheduling and inter-fetch-group prefetching 50
4.2.2 Orchestrating Warp Scheduling and Data Prefetching 51
4.2.2.1 Prefetch-aware (PA) warp scheduling 51
4.2.2.2 Prefetch-aware (PA) warp scheduling and inter-fetch-group prefetching 52

4.3 Mechanism and Implementation 53
4.3.1 Prefetch-Aware Scheduling Mechanism 53
4.3.2 Spatial Locality Detection Based Prefetching 54
4.3.3 Hardware Overhead 57
4.4 Evaluation Methodology 58
4.5 Experimental Results 59
4.6 Related Work 64
4.7 Chapter Summary 66

Chapter 5
Criticality Aware Memory Scheduling Techniques **68**

5.1 Introduction 69
5.2 Background and Motivation 72
5.3 Core Criticality: Basic Ideas and Metrics 74
5.3.1 Latency Tolerance as a Measure of Core-Criticality 74
5.3.2 Understanding Variation of Criticality Across Cores 76
5.3.2.1 Analysis of the PCC metric. 77
5.4 Analyzing Criticality in the Memory System 78
5.5 CLAMS: Design and Implementation 81
5.5.1 Design Challenges of CLAMS 81
5.5.2 Design Overview of CLAMS 82
5.5.3 Design of Static-CLAMS Memory Scheduler 84
5.5.4 Design of Semi-Dyn-CLAMS Memory Scheduler 85
5.5.5 Design of Dyn-CLAMS Memory Scheduler 87
5.5.6 Hardware Overheads 89
5.6 Evaluation Methodology 91

5.7 Experimental Results 91

5.8 Related Work 96

5.9 Chapter Summary 98

Chapter 6
Concurrent Kernel Execution in GPUs: Problems and Some Solutions **99**

6.1 Introduction 100

6.2 Background and Experimental Methodology 102

6.2.1 Baseline GPU Architecture 103

6.2.2 Evaluation Methodology 104

6.2.3 Evaluation Metrics 105

6.3 Concurrent Kernel Execution Challenges 106

6.3.1 Fairness considerations 107

6.3.2 Throughput considerations 109

6.4 Application-Aware Memory Scheduling 110

6.4.1 Designing Application-aware Memory Scheduler 111

6.4.2 Hardware Complexity 112

6.5 Experimental Results 113

6.5.1 Fairness Results 113

6.5.2 Performance Results 114

6.6 Related Work 115

6.7 Chapter Summary 118

Chapter 7
Anatomy of Multi-Application Execution in GPUs **119**

7.1 Introduction 120

7.2 Background 122

7.2.1 Baseline Architecture 122
7.2.2 Evaluation Metrics and Application Suite 123
7.3 Performance Characterization of Many-threaded Architectures . . . 124
7.3.1 A Model for Many-threaded Architectures 124
7.3.2 Application Characterization 126
7.4 Analyzing Memory System Interference 127
7.4.1 The Problem: Application Interference 128
7.4.2 Limitations of Existing Memory Schedulers 131
7.5 A Performance Model for Concurrently Executing Applications . . . 133
7.5.1 Analyzing Instruction Throughput 134
7.5.2 Analyzing Weighted Speedup 136
7.6 Mechanism and Implementation Details 138
7.7 Infrastructure and Evaluation Methodology 139
7.8 Experimental Results 141
7.8.1 Evaluation of ITS 142
7.8.2 Evaluation of WEIS 144
7.8.3 Performance Summary 145
7.8.4 Scalability Analysis 146
7.9 Related Work 147
7.10 Chapter Summary 149

Chapter 8
Conclusions and Future Research Directions **151**

8.1 Summary of Dissertation Contributions 151
8.2 Future Research Directions 153

Bibliography **156**

List of Figures

2.1 (A) GPGPU architecture, (B) CTA data layout, and (C) Main memory layout with CTA's data mapped. 8

2.2 GPGPU application hierarchy. 9

3.1 Fraction of total execution cycles (of all the cores) during which *all* the warps launched on a core are waiting for their respective data to come back from L2 cache/DRAM. This chapter defines the number of cycles where all warps are stalled due to memory as *MemoryBlockCycles*. AVG-T1 is the average (arithmetic mean) value across all Type-1 applications. AVG is the average value across all 38 applications. 16

3.2 An illustrative example showing the working of (A) CTA-aware two-level warp scheduling (*CTA-Aware*) (B) Locality aware warp scheduling (*CTA-Aware-Locality*). Label in each box refers to the corresponding CTA number. 21

3.3 An example illustrating (A) the under-utilization of DRAM banks with *CTA-Aware-Locality*, (B) improved bank-level parallelism with *CTA-Aware-Locality-BLP*, (C1, C2) the positive effects of *Opportunistic Prefetching*. 25

3.4 Effect of *CTA-Aware-Locality-BLP* on DRAM bank-level parallelism and row locality, compared to *CTA-Aware-Locality*. . . . 27

3.5 Performance impact of the schemes on Type-1 applications. Results are normalized to RR. 27

3.6 Impact of different scheduling schemes on *MemoryBlockCycles* for Type-1 applications. Results are normalized to the total execution cycles with baseline RR scheduling. 33

3.7 Sensitivity of IPC to group size (normalized to RR). 37

3.8 Sensitivity of IPC to the number of banks (normalized to RR). . . . 37

3.9 Prefetch degree and throttling threshold sensitivity. 38

4.1 IPC improvement when L1 cache is made perfect on a GPGPU that employs (1) round-robin (RR) warp scheduling policy, (2) two-level (TL) warp scheduling policy, (3) data prefetching together with RR, and (4) data prefetching together with TL. Section 4.4 describes the evaluation methodology and workloads. 43

4.2 An illustrative example showing the working of various scheduling and prefetching mechanisms, motivating the need for the design of the prefetch-aware warp scheduler. 48

4.3 The distribution of main memory requests, averaged across all fetch groups, that are to macro-blocks that have experienced, respectively, 1, 2, and 3-4 unique cache misses. Section 4.4 describes the methodology and workloads. 55

4.4 IPC performance impact of different scheduling and prefetching strategies. Results are normalized to the IPC with the RR scheduler. 59

4.5 (a) Prefetch accuracy, (b) Fraction of late prefetches, and (c) Reduction in L1 data cache miss rate when prefetching is implemented with each scheduler. The results are averaged across all applications. 60

4.6 Effect of various scheduling strategies on DRAM bank-level parallelism (BLP) 60

4.7 Effect of various scheduling and prefetching strategies on L1D miss rate. Results are normalized to miss rates with the TL scheduler. . 61

4.8 Effect of different scheduling and prefetching strategies on DRAM row buffer locality 62

4.9 Effect of prefetching on Evicted Block Reference Rate (EBRR) for various L1 data cache sizes 62

5.1 Average Coefficient of Variation (COV) in average memory access latencies and IPCs across different GPU cores. 73

5.2 Illustrative example to demonstrate PCC. 77

5.3 Variation of criticality across cores with different criticality-rank thresholds. This variation is measured using the PCC metric. . . . 78

5.4 Effect of increase in peak memory bandwidth on PCC. 78

5.5 Variation of criticality across requests at different levels of criticality-rank thresholds. This variation is measured using the PCR. 80

5.6 Distribution of criticality-rank differences across requests. 80

5.7 Distribution of requests in different criticality-rank states. 85

5.8 Execution of CONS and SCP to illustrate the working of Semi-Dyn-CLAMS. Th_{CR} values are calculated dynamically. 86

5.9 Execution of CONS and SCP to illustrate the working of Dyn-CLAMS. Th_{SM} is dynamically updated based on Th_{CR}. 89

5.10 Performance results normalized to FR-FCFS. 92

5.11 Effect on (a) DRAM page hit rates, (b) memory latencies for critical requests, (c) core stall cycles. Results are normalized to FR-FCFS. 93

5.12 Changes in Th_{CR} are observed when Semi-Dyn-CLAMS is employed. When Th_{CR}=8, scheduler is in locality mode. 93

5.13 Changes in Th_{SM} are observed with Dyn-CLAMS. 94

5.14 Effect of Dyn-CLAMS on DRAM bandwidth distribution. 96

6.1 Baseline GPU architecture executing: (A) Single kernel, (B) Multiple kernels concurrently. 101

6.2 Weighted speedup (Application throughput) for the evaluated workloads. The `1st APP` and `2nd APP` are the first and second applications in the workload, respectively, as mentioned in Table 6.3. 107

6.3 Fairness Index for the evaluated workloads when the memory scheduler adopts the baseline FR-FCFS scheduling policy. 107

6.4 DRAM bandwidth utilization distribution across various workloads when memory scheduler adopts the baseline FR-FCFS memory scheduling policy. 109

6.5 Conceptual example showing the working of (A) baseline FR-FCFS memory scheduling, (B) proposed FR-RR-FCFS memory scheduling. 110

6.6 Effect on DRAM page hit rates. The proposed scheduler FR-RR-FCFS preserves the DRAM page hit rates obtained by the baseline FR-FCFS memory scheduler. 111

6.7 Fairness index (FI) of the evaluated workloads when memory scheduler adopts FR-FCFS (baseline, 1st bar) and FR-RR-FCFS (proposed, 2nd bar) scheduling techniques. 115

6.8 DRAM bandwidth utilization distribution across selected workloads when memory scheduler adopts FR-FCFS (baseline, 3rd bar) and FR-RR-FCFS (proposed, 4th bar) scheduling techniques. 115

6.9 Improvement in instruction throughput (IT) across the evaluated workloads. Results are normalized to the case when memory scheduler adopts the baseline FR-FCFS scheduling policy. 116

6.10 Improvement in weighted speedup (WS) across the evaluated workloads. Results are normalized to the case when memory scheduler adopts the baseline FR-FCFS scheduling policy. 116

7.1 Overview of the baseline architecture capable of executing multiple applications. 122

7.2 Application performance obtained via simulation and the model. IPC is normalized with respect to the maximum achievable IPC supported by the architecture. 127

7.3 Absolute relative error between IPCs obtained from real hardware (NVIDIA Kepler K20m) and the model. 127

7.4 Different performance slowdowns obtained when BLK is co-scheduled with three different applications: `GUPS`, `QTC`, and `NN`. Memory scheduling policy is FR-FCFS. 130

7.5 Different performance slowdowns experienced when different memory scheduling schemes are employed. 131

7.6 An illustrative example showing IT and WS for two applications running together. The shaded boxes represent system and application properties. The peak memory bandwidth is 50 units. Application 1 and 2 use 30 and 40 units bandwidth, respectively, when they execute alone. Their $MPKI$s are 20 and 5, respectively. 133

7.7 The effect of FR-FCFS, RR, and ITS on BW_1-BW_2 and $MPKI_1-MPKI_2$. 142

7.8 IT results normalized with respect to FR-FCFS for 25 representative workloads. 142

7.9 Effect of FR-FCFS, RR, and WEIS on WS and $BW_1 - BW_2$. 144

7.10 WS results normalized with respect to FR-FCFS for 25 representative workloads. 144

7.11 Summary IT and WS results for 100 workloads, normalized with respect to FR-FCFS. 145

7.12 HS results for 100 workloads normalized with respect to FR-FCFS. 146

7.13 Core partitioning results. 147

7.14 Evaluation of ITS and WEIS with three GPU applications. 147

List of Tables

3.1 GPGPU application characteristics: *(A) PMEM*: IPC improvement with perfect memory (All memory requests are satisfied in L1 caches), Legend: H = High (>= 1.4x) , L = Low (< 1.4x); *(B) CINV*: The ratio of inactive cycles to the total execution cycles of all the cores. 15

3.2 Reduction in L1 miss rates with the proposed warp scheduling mechanisms over baseline RR scheduling. 19

3.3 GPGPU application characteristics: *Consecutive CTA row sharing*: Fraction of consecutive CTAs (out of all CTAs) accessing the same DRAM row. *CTAs/Row*: Average number of CTAs accessing the same DRAM row. 24

3.4 Baseline configuration 32

4.1 Simulated baseline GPGPU configuration 57

4.2 Evaluated GPGPU applications 59

5.1 Pseudo code for the proposed schemes 88

5.2 Key configuration parameters of the simulated GPU configuration. . 90

5.3 Evaluated applications. Table also shows: 1) Average occupancy (occ) in terms of warps, 2) Average Th_{CR} and Th_{SM} calculated using Semi-Dyn-CLAMS and Dyn-CLAMS, respectively, and 3) % of critical requests (%-cri) served in the criticality-mode. 90

6.1 Simulated baseline GPU configuration 103

6.2 Evaluated applications, along with their DRAM bandwidth utilization when they are executed alone on the entire baseline GPU architecture. 105

6.3 Evaluated 2-application GPU workloads. 106

7.1 Application characteristics: *(A) MPKI*: L2 cache misses per kilo-instructions. *(B) BW/C*: The ratio of attained bandwidth to the peak bandwidth of the system. 128

7.2 Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.1 [116] for full list. 139

Acknowledgments

This dissertation would not have been possible without the help and support of many people with whom I regularly interacted throughout my Ph.D. tenure. First and foremost, I would like to thank my dissertation advisor, Prof. Chita Das. He has been a constant source of inspiration for me. He always inspired me to do excellent research, submit papers in the best venues, take challenging courses, and most importantly to be a humble citizen. I am highly indebted to him for what he has given me so far. I also thank his entire family for their unconditional support.

I would like to thank my dissertation co-advisor, Prof. Mahmut Kandemir, for his constant support. He has been a great critic of my work. I deeply admire his dedication towards research and I hope to imitate that as I continue my career in academia. The chats over coffee with him along with Prof. Das, Prof. Sivasubramaniam, and other HPCL members on a variety of topics (sometimes also on research) were refreshing, and I will remember them for long time.

My first interaction with Prof. Mutlu was during my internship at Intel back in 2012. Since then, we have been involved in many fruitful research collaborations. I deeply admire his work ethics and care towards making the research work polished and more accessible to the readers. I appreciate the time he spent in making this dissertation stronger, clearer, and more readable, in addition to making me more methodical and organized in doing research. I deeply thank Prof. Xie and Dr. Iyer for serving on my committee. I appreciate their support and guidance during my entire Ph.D. tenure. I also thank Prof. Jenkins, Prof. Narayanan, and Prof. Sampson for commenting on my work and sharing their thoughts.

My internship experiences at Intel and NVIDIA had been great learning experiences for me. I would like to thank my internship mentors for providing a lively environment for me to work. In this regard, I thank Ramadass Nagarajan, Xiaowei Jiang, Li Zhao, Ravi Iyer, Srihari Makineni, Evgeny Bolotin, Zvika Guz, Mike Parker, and Steve Keckler. In addition, I would like to thank all my

research collaborators and colleagues with whom I have interacted.

My Ph.D. journey had been incredibly enjoyable because of the support and a friendly environment provided by my lab mates. I have developed a real camaraderie with Onur Kayiran after collaborating with him on so many research projects. His research acumen really helped in making this dissertation stronger. I do not even remember how many impossible ideas I have brainstormed with him. I admire his patience to listen those and more importantly his patience to provide really good feedback on all of that. I deeply thank Asit Mishra for being my first research mentor and teaching me how to do research. My overlap with Bikash Sharma was for four years. Although we never collaborated on the same research project, we used to talk through the obstacles as they came in our respective projects. We were apartment-mates for four years, and I thank him for his friendship. I also thank Nachiappan CN for being a good friend of mine and giving his dollars (not 2 cents) whenever I was stuck at something. The great company of Kashyap Dixit and Ashutosh Pattnaik helped me sail thorough the end years of my Ph.D. Also, special thanks to Neha Sharma for all her support during my Ph.D. tenure.

Much of my research was at the mercy of the compute clusters hosted by my department. I thank the entire CSE department technical staff for managing those clusters. In particular, I thank Eric Prescott for helping me with my requests even at the wee hours on the weekends. I also thank the entire administrative staff of CSE department for being so efficient and getting my paper-work going. Special thanks to Annie Royer for patiently handling tons of my travel reimbursement paper-work.

There have been lots of other people whose company made my experience at Penn State memorable. In this regard, I would like to thank Reetuparna Das, Seung-Hwan Lim, Sai Prashanth, Akbar Sharifi, Niranjan Soundararajan, Abhradeep Guha Thakurta, Shrawan C. Surendar, Emre Kultursay, Prashanth Thinakaran, Tulika Parija, Xulong Tang, Haibo Zhang, Jihyun Ryoo, Mahshid Sedghi, Tuba Kesten, Jagadish Kotra, Diana Guttman, Amin Jadidi, Di Wang, Karthik Swaminathan, Narges Shahidi, Harshal Patanakar, Vivek Kaushal, Abhishek Kar, Anushree Dash, Debanjan Das, Shashank Singhai, Suchismita Sarangi, Berkey Cellik, Giuseppe Salento, Nandhini Chandramoorthy, Nirupama Talele, Cong Xu, Bhaskar Prabhala, Praveen Yedlapalli, Sushama Karumanchi. If you are reading this page and think that your name is missing, excuse me for that and assume it is written with an invisible ink.

Finally, I am running short of words in thanking my parents, grandparents,

cousins, and other family members for providing unconditional support.

Thank you all.

Dedication

To my dear parents and grandparents for being a constant source of inspiration and support.

Chapter 1

Introduction

Graphics Processing Units (GPUs) have recently emerged as a cost-effective *throughput computing* paradigm for a wide range of areas such as science, engineering, medicine, social media, gaming, and finance, due to their immense computing power compared to CPUs [1–10]. Many of the world's Top 500 computers [11, 12] use GPUs both for performance and energy-efficiency. Similarly, orders of magnitude improvements in application performance in medical science [13–15], finance [16–18], and social media [19] have been reported recently by offloading computation to GPUs. It is not only expected that GPUs will play a critical role in the foreseeable computing landscape ranging from supercomputing machines to handheld devices, but they also could be a natural choice for processing *big-data* to advance science and engineering. As an example, US healthcare data reached 150 Exabytes in 2011 and is likely to grow to Zettabytes/Yottabytes soon [20]. Similar trends in data explosion are predicted in many domains such as the environment, traffic control, manufacturing, climate prediction and astrophysics [21]. Mining information in such large datasets requires scalable parallel computing capabilities for which GPUs provide a very promising fit.

Modern GPUs are characterized by numerous programmable computational cores and thousands of simultaneously active fine-grained threads. To facilitate ease of programming on these systems, programming models like CUDA [4, 22, 23] and

OpenCL [24] have been developed. GPU applications are typically divided into several kernels, where each kernel is capable of spawning many threads. The threads are usually grouped together into *thread blocks*, also known as *cooperative thread arrays (CTAs)*. When an application starts its execution on a GPU, the CTA scheduler initiates scheduling of CTAs onto the available GPU cores. All the threads within a CTA are executed on the same core typically in groups of 32 threads. This collection of threads is referred to as a *warp* and all the threads within a warp typically share the same instruction stream, which forms the basis for the term *single instruction multiple threads*, SIMT [3, 25, 26].

The Problem: In spite of having numerous resident threads and theoretically high thread-level parallelism (TLP), GPU cores still suffer from high periods of idle times, resulting in under-utilization of hardware resources. These idle times are primarily a result of the inability of the commonly-employed warp scheduling policies in facilitating a GPU core to completely tolerate the long memory fetch latencies, which are primarily attributed to: (1) contention in caches caused by multiple concurrent threads, (2) DRAM contention caused by various concurrent threads from multiple GPU cores, and (3) limited off-chip DRAM bandwidth available in GPUs. With the commonly-employed warp scheduling policies, for example, round-robin (RR) scheduler, the GPGPU core becomes *inactive* because there may be no warps that are *not* stalling due to a memory operation, which significantly reduces the capability of hiding long memory latencies. Such inactive periods are especially prominent in memory-intensive applications. It is observed that out of 38 applications covering various GPGPU application benchmark suites, 19 applications suffer from *very high* core inactive times (on average 62% of total cycles are spent with no warps executing). In addition to the inefficiencies in warp schedulers, the modern memory access schedulers in GPUs (for example, first-ready, first-come-first-serve (FR-FCFS)) also possess limitations. Typically, they are only optimized for DRAM access locality to enhance memory bandwidth utilization, and implicitly assume all requests from different GPU cores are equally important. Hence, these schedulers do not prioritize critical memory requests over non-critical requests, thereby exhibiting sub-optimal performance.

Dissertation Contributions: This dissertation research addresses the above issues by focusing on the problem of cache and memory resource contention. In this context, this dissertation proposes techniques that are employed at two different places: *core* and *memory*. First, this dissertation shows that by intelligently scheduling the threads at the *core*, the generated memory request patterns can be more amenable for existing resource management techniques such as cache replacement and memory scheduling as well as performance enhancement techniques such as data prefetching. Second, this dissertation shows that considering criticality and other application characteristics to schedule memory requests at the memory controller is an effective way to manage contention at the *memory*.

(A) Managing Contention from Cores via Warp Scheduling

This dissertation shows that the existing warp scheduling techniques employed at the core are oblivious to the underlying shared resource management policies, and therefore, the warp scheduling decisions taken at the core might not always be in harmony with the shared resource management scheduling policies. In this context, this dissertation proposes two warp scheduling techniques: 1) cache and memory-aware warp scheduling; and 2) data prefetching-aware warp scheduling. Both of these schedulers exploit an important property of GPUs that there is no ordering restriction among the execution of warps and the warp scheduler can efficiently choose the desired warps without incurring significant overhead.

Contribution I: Cache and Memory-Aware Warp Scheduling [7]. The key problem with the traditional round-robin warp scheduling policy is that it allows a large number of warps to concurrently access the cache. This makes it harder for the underlying cache management policies to leverage the locality present in many CUDA applications. In order to manage the contention in caches, this dissertation proposes a *cache-aware* warp scheduler that essentially reduces the number of warps that can benefit from caches in a given time interval. Although this scheduler improves the cache hit rates significantly, it turns out that it is not aware of the warp scheduling decisions taken at the *other* cores. This unawareness causes the schedulers at different cores to schedule warps such that they happen to concurrently access a limited set of global memory

banks, consequently leading to inefficient utilization of the available memory bandwidth. To this end, this dissertation proposes a *memory-aware* warp scheduler called OWL by extending the cache-aware warp scheduler such that it can facilitate better coordination between warp scheduling decisions taken at different cores. OWL enabled the warps across different cores to collectively access a larger number of memory banks concurrently, thereby improving the memory-level parallelism and easing the job of the memory scheduler in managing contention at the banks.

Contribution II: Prefetch-Aware Warp Scheduling [27]. Effectiveness of data prefetching is dependent on the prefetching accuracy as well as timeliness of prefetches. The analyses of the existing warp schedulers show that they do not coordinate well with the prefetching mechanisms because they happen to schedule consecutive warps accessing nearby cache blocks in close succession. Therefore, a simple prefetcher that could have prefetched nearby cache blocks with high accuracy will not contribute significantly to the performance because many of them will be tagged as late prefetches. To this end, this dissertation proposes a *prefetch-aware* warp scheduling policy that can coordinate with prefetching decisions in GPUs to better tolerate long memory latencies. This scheduler separates the scheduling of consecutive warps in time, and by not executing them in close succession, it enables effective incorporation of simple prefetching techniques for improving the overall GPU performance.

(B) Managing Contention at Memory via Memory Scheduling

Memory access schedulers employed in GPUs implicitly assume that all requests from different cores are equally important. This dissertation shows that this assumption does not necessarily help in achieving: 1) the best performance when GPU cores concurrently execute threads belonging to a *single* application; and 2) the best system throughput and fairness when GPU cores concurrently execute threads belonging to *multiple* applications. To address these two scenarios, this dissertation proposes criticality-aware and application-aware memory scheduling techniques, respectively.

Contribution III: Criticality-Aware Memory Scheduling. Shared resource

contention causes significant variation in average memory latencies experienced by individual GPU cores. Due to this variation, the number of stalling warps belonging to the cores that suffer from higher memory access latencies is typically higher than that of other cores, making the former type of cores less latency tolerant, i.e., more *critical*. This implies that because different GPU cores have varying degrees of tolerance to latency during the execution of an application, their corresponding memory requests have varying degrees of criticality. Based on this observation, this dissertation proposes a criticality-aware scheduler that takes into account the criticality of memory requests, i.e., the latency-tolerance of the cores that generate memory requests, thereby improving the overall GPU performance over existing schedulers.

Contribution IV: Application-Aware Memory Scheduling [28, 29]. This dissertation finds that an uncoordinated allocation of GPU resources among concurrently executing multiple applications can lead to significant degradation in system throughput and fairness. Therefore, it is imperative to make the GPU memory system aware of the application characteristics. To this end, this dissertation proposes two different application-aware memory scheduling policies. The first scheduler is developed with the aim of sharing the memory bandwidth across concurrent applications in a fair and efficient manner. The second scheduler is more sophisticated and is based on an analytical performance model for GPUs. This dissertation proposes that the common use of misses-per-instruction (MPI) as a proxy for performance is not accurate for many-threaded architectures and memory scheduling decisions based on *both* MPI and attained DRAM bandwidth are more effective in enhancing system throughput and fairness. This dissertation evaluates both schedulers on a newly-developed simulated GPU platform supporting execution of multiple applications.

The rest of this dissertation is organized as follows. Chapter 2 provides preliminaries for GPU architectures. Then, the OWL warp scheduler is presented in Chapter 3. Prefetch-aware warp scheduler is discussed in Chapter 4, followed by the description of the criticality-aware memory scheduler in Chapter 5. Chapters 6 and 7 looks at application-aware memory system design issues for

GPUs. Chapter 7 concludes this dissertation and presents a few directions for future work.

Chapter 2

Graphics Processing Units (GPUs): A Primer

This chapter provides preliminaries for GPU architectures, application design, and typical scheduling and prefetching strategies employed in GPUs.

GPU Architecture: A General Purpose Graphics Processing Unit (GPGPU) consists of many simple cores (streaming multiprocessors), with each core typically having a SIMT width of 8 to 32 (NVIDIA's Fermi series has 16 streaming multiprocessors with a SIMT width of 32 [4]). A typical GPU architecture (as shown in Figure 2.1 (A)) consists of many shader cores connected to memory controllers via an on-chip interconnect. This configuration is similar to the ones studied in prior works [30, 31]. Each core is associated with a private L1 data cache and read-only texture and constant caches along with a low latency shared memory (scratchpad memory). Every memory controller is associated with a slice of the shared L2 cache for faster access to the cached data.

Canonical GPGPU Application Design: A typical CUDA application consists of many kernels (or grids) as shown in Figure 2.2 (A). These kernels implement specific modules of an application. Each kernel is divided into groups of threads, called cooperative thread arrays (CTAs) (Figure 2.2 (B)). A CTA is an abstraction which encapsulates all synchronization and barrier primitives

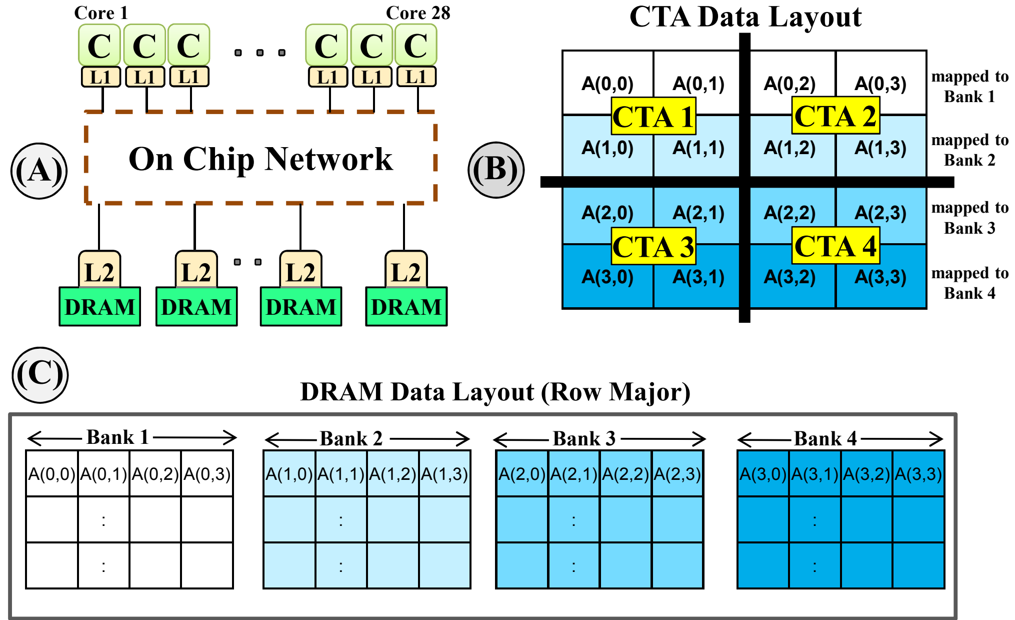


Figure 2.1: (A) GPGPU architecture, (B) CTA data layout, and (C) Main memory layout with CTA's data mapped.

among a group of threads [1]. Having such an abstraction allows the underlying hardware to relax the execution order of the CTAs to maximize parallelism. The underlying architecture in turn, sub-divides each CTA into groups of threads (called warps) (Figure 2.2 (C) and (D)). This sub-division is transparent to the application programmer and is an architectural abstraction.

CTA, Warp, and Thread Scheduling: Execution on GPGPUs starts with the launch of a kernel. All kernels can either execute sequentially or concurrently. In case kernels are executed sequentially, after a kernel is launched, the CTA scheduler schedules available CTAs associated with the kernel in a round-robin and load balanced fashion on all the cores [31]. For example, CTA 1 is assigned to core 1, CTA 2 is assigned to core 2 and so on. After assigning at least one CTA to each core (provided that enough CTAs are available), if there are still unassigned CTAs, more CTAs can be assigned to the same core in a similar fashion. The maximum number of CTAs per core (N) is limited by core resources (number of threads, size of shared memory, register file size, etc. [1, 31]). Given a baseline architecture, N may vary across kernels depending on how much resources are needed by a CTA of a particular kernel. If a CTA of a particular kernel needs

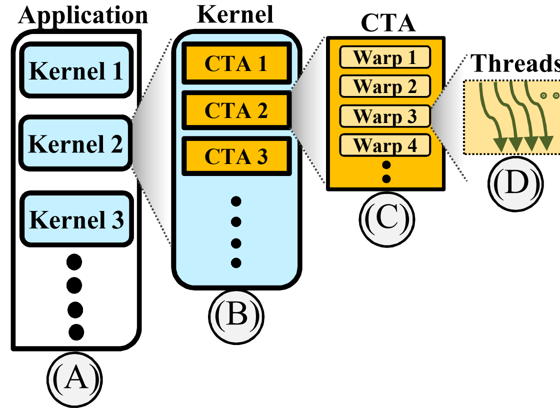


Figure 2.2: GPGPU application hierarchy.

more resources, N will be smaller compared to that of another kernel whose CTAs need fewer resources. For example, if a CTA of kernel X needs 16KB of shared memory and the baseline architecture has 32KB of shared memory available, a maximum of 2 CTAs of kernel X can be executed simultaneously.

The above CTA assignment policy is followed by per-core GPGPU warp scheduling. Warps associated with CTAs are scheduled in a round-robin (RR) fashion on the assigned cores [8, 31] and get *equal* priority. In traditional GPU architecture, every 4 cycles, a warp ready for execution is selected in a round-robin fashion and fed to the 8-way SIMT pipeline of a GPGPU core. At the memory stage of the core pipeline, if a warp gets blocked on a long latency memory operation, the entire warp (32 threads) is scheduled out of the pipeline and moved to the pending queue. At a later instant, when the data for the warp arrives, it proceeds to the write-back stage, and then fetches new instructions.

When multiple *kernels* from multiple *applications* are executed concurrently on a GPU, they can be assigned to the SMs using an equal partitioning mechanism. If two kernels from different applications are concurrently executed, this mechanism assigns half of the SMs to the first application and the second half to the other application. The CTA assignment for each kernel follows the same load-balanced distribution strategy as described before; the only difference is that each kernel is now assigned to only half of the SMs of the baseline GPU architecture.

CTA Data Layout: Current GPU chips support ~10× higher memory

bandwidth compared to CPU chips [7]. In order to take full advantage of the available DRAM bandwidth and to reduce the number of requests to DRAM, a kernel must arrange its data accesses so that each request to the DRAM is for a large number of consecutive DRAM locations. With the SIMT execution model, when all threads in a warp execute a memory operation, the hardware typically detects if the threads are accessing consecutive memory locations; if they are, the hardware coalesces all these accesses into a single consolidated access to DRAM that requests all consecutive locations at once. To understand how data blocks used by CTAs are placed in the DRAM main memory, consider Figure 2.1 (B). This figure shows that all locations in the DRAM main memory form a single, consecutive address space. The matrix elements that are used by CTAs are placed into the linearly addressed locations according to the row major convention as shown in Figure 2.1 (B). That is, the elements of row 0 of a matrix are first placed in order into consecutive locations (see Figure 2.1 (C)). The subsequent row is placed in another DRAM bank. Note that, this example is simplified for illustrative purposes only, and the data layout may vary across applications.

Memory Scheduling in GPUs: First-ready FCFS (FR-FCFS) [32–34] is the commonly employed memory scheduling technique in GPUs. This scheme is targeted at improving DRAM row hit rates, so request prioritization order is as follows: 1) row-hit requests are prioritized over other requests; then 2) older requests are prioritized over younger requests. Among row-hit requests, older requests are prioritized over younger requests.

Prefetching in GPGPUs: Inter-thread L1 data prefetching [35] was recently proposed as a latency hiding technique in GPGPUs. In this technique, a group of threads prefetch data for threads that are going to be scheduled later. This inter-thread prefetcher can also be considered as an inter-warp prefetcher, as the considered baseline architecture attempts to coalesce the memory requests of all the threads in a warp as a single cache block request (e.g., 4B requests per thread $\times$ 32 threads per warp = 128B request per warp, which equals the cache block size). The authors propose that prefetching data for other warps (in turn, threads) can eliminate cold misses, as the warps for which the data is prefetched will find

their requested data in the cache. In the case where the threads demand their data before the prefetched data arrives, the demand requests can be merged with the already-sent prefetch requests (if accurate) via miss status handling registers (MSHRs). In this case, the prefetch can partially hide some of the memory latency.

In the next sections, this dissertation will detail the developed warp and memory scheduling techniques.

Chapter 3

Cache and Memory Aware Warp Scheduling Techniques

Emerging GPGPU architectures, along with programming models like CUDA and OpenCL, offer a cost-effective platform for many applications by providing high thread level parallelism at lower energy budgets. Unfortunately, for many general-purpose applications, available hardware resources of a GPGPU are not efficiently utilized, leading to lost opportunity in improving performance. A major cause of this is the inefficiency of current warp scheduling policies in tolerating long memory latencies.

This chapter identifies that the scheduling decisions made by such policies are agnostic to thread-block, or cooperative thread array (CTA), behavior, and as a result inefficient. This chapter presents a coordinated CTA-aware scheduling policy that utilizes four schemes to minimize the impact of long memory latencies. The first two schemes, CTA-aware two-level warp scheduling and locality aware warp scheduling, enhance per-core performance by effectively reducing cache contention and improving latency hiding capability. The third scheme, bank-level parallelism aware warp scheduling, improves overall GPGPU performance by enhancing DRAM bank-level parallelism. The fourth scheme employs opportunistic memory-side prefetching to further enhance performance by taking advantage of open DRAM rows. Evaluations on a 28-core GPGPU

platform with highly memory-intensive applications indicate that the proposed mechanism can provide 33% average performance improvement compared to the commonly-employed round-robin warp scheduling policy.

3.1 Introduction

The goal of this work is to tackle the under-utilization of cores for improving the overall GPGPU performance. In this context, the *c(O)operative thread array a(W)are warp schedu(L)ing policy*, called OWL[1] is developed. OWL is based on the concept of *focused* CTA-aware scheduling, which attempts to mitigate the various components that contribute to long memory fetch latencies by *focusing* on a selected subset of CTAs scheduled on a core (by *always* prioritizing them over others until they finish). The proposed OWL policy is a four-pronged concerted approach and the associated **contributions** are:

• First, a CTA-aware two-level warp scheduler is developed that exploits the architecture and application interplay to intelligently schedule CTAs onto the cores. This scheme groups all the available CTAs (N CTAs) on a core into smaller groups (of n CTAs) and schedules all *groups* in a round-robin fashion. As a result, it performs better than the commonly-used baseline RR warp scheduler because 1) it allows a smaller group of warps/threads to access the L1 cache in a particular interval of time, thereby reducing cache contention, 2) improves latency hiding capability and reduces inactive periods as not all warps reach long latency operations around the same time. This technique improves the average L1 cache hit rate by 8% over RR for 19 highly memory intensive applications, providing a 14% improvement in IPC performance.

• Second, a locality aware warp scheduler is developed to improve upon the CTA-aware two-level warp scheduler, by further reducing L1 cache contention. This is achieved by *always* prioritizing a group of CTAs (n CTAs) in a core over the rest

[1]Owl is a bird known for exceptional *vision* and *focus* while it hunts for food. The proposed scheduling policy also follows an owl's philosophy. It intelligently selects (*visualizes*) a subset of CTAs (out of many CTAs launched on a core) and *focuses* on them to achieve performance benefits.

of the CTAs (until they finish). Hence, unlike the base scheme, where each group of CTAs (consisting of n CTAs) is executed one after another and thus, does not utilize the caches effectively, this scheme always prioritizes one group of CTAs over the rest whenever a particular group of CTA is ready for execution. The major goal is to take advantage of the locality between nearby threads and warps (associated with the same CTA) [36]. With this scheme, average L1 cache hit rate is further improved by 10% over the CTA-aware two-level warp scheduler, leading to an 11% improvement in IPC performance.

• Third, the first two schemes are aware of different CTAs but do not exploit any properties common among different CTAs. Across 38 GPGPU applications, there is significant DRAM page locality between consecutive CTAs. On average, the same DRAM page is accessed by consecutive CTAs 64% of the time. Hence, if two *consecutive* CTA groups are scheduled on two different cores and are *always* prioritized according to the locality aware warp scheduling, they would access a *small* set of DRAM banks more frequently. This increases the queuing time at the banks and reduces memory bank level parallelism (BLP) [37]. On the other hand, if non-consecutive CTA groups are scheduled and *always* prioritized on two different cores, they would concurrently access a larger number of banks. This reduces the contention at the banks and improves BLP. This proposed scheme (called the bank-level parallelism aware warp scheduler), increases average BLP by 11% compared to the locality aware warp scheduler, providing a 6% improvement in IPC performance.

• Fourth, a drawback of the previous scheme is that it reduces DRAM row locality. This is because rows opened by a CTA cannot be completely utilized by its consecutive CTAs since consecutive CTAs are not scheduled simultaneously any more. To recover the loss in DRAM row locality, an opportunistic prefetching mechanism is developed, in which some of the data from the opened row is brought to the nearest on-chip L2 cache partition. The mechanism is opportunistic because the degree of prefetching depends upon the number of pending demand requests at the memory controller.

The performance of the OWL scheduler is evaluated on a 28-core GPGPU platform simulated via GPGPU-Sim [31] and a set of 19 highly memory intensive

applications. The results show that OWL improves GPGPU performance by 33% over the baseline RR warp scheduling policy. OWL also outperforms the recently-proposed two-level scheduling policy [8] by 19%.

Table 3.1: GPGPU application characteristics: *(A) PMEM*: IPC improvement with perfect memory (All memory requests are satisfied in L1 caches), Legend: H = High (>= 1.4x) , L = Low (< 1.4x); *(B) CINV*: The ratio of inactive cycles to the total execution cycles of all the cores.

#	App. Suite	**Type-1 Applications**	Abbr.	PMEM	CINV
1	Parboil	Sum of Abs. Differences	SAD	H (6.39x)	91%
2	MapReduce	PageViewCount	PVC	H (4.99x)	93%
3	MapReduce	SimilarityScore	SSC	H (4.60x)	85%
4	CUDA SDK	Breadth First Search	BFS	H (2.77x)	81%
5	CUDA SDK	MUMerGPU	MUM	H (2.66x)	72%
6	Rodinia	CFD Solver	CFD	H (2.46x)	66%
7	Rodinia	Kmeans Clustering	KMN	H (2.43x)	65%
8	CUDA SDK	Scalar Product	SCP	H (2.37x)	58%
9	CUDA SDK	Fast Walsh Transform	FWT	H (2.29x)	58%
10	MapReduce	InvertedIndex	IIX	H (2.29x)	65%
11	Parboil	Sparse-Matrix-Mul.	SPMV	H (2.19x)	65%
12	3rd Party	JPEG Decoding	JPEG	H (2.12x)	54%
13	Rodinia	Breadth First Search	BFSR	H (2.09x)	64%
14	Rodinia	Streamcluster	SC	H (1.94x)	52%
15	Parboil	FFT Algorithm	FFT	H (1.56x)	37%
16	Rodinia	SRAD2	SD2	H (1.53x)	36%
17	CUDA SDK	Weather Prediction	WP	H (1.50x)	54%
18	MapReduce	PageViewRank	PVR	H (1.41x)	46%
19	Rodinia	Backpropogation	BP	H (1.40x)	33%
20	CUDA SDK	Separable Convolution	CON	L (1.23x)	20%
21	CUDA SDK	AES Cryptography	AES	L (1.23x)	51%
22	Rodinia	SRAD1	SD1	L (1.17x)	20%
23	CUDA SDK	Blackscholes	BLK	L (1.16x)	17%
24	Rodinia	HotSpot	HS	L (1.15x)	21%
25	CUDA SDK	Scan of Large Arrays	SLA	L (1.13x)	17%
26	3rd Party	Denoise	DN	L (1.12x)	22%
27	CUDA SDK	3D Laplace Solver	LPS	L (1.10x)	12%
28	CUDA SDK	Neural Network	NN	L (1.10x)	13%
29	Rodinia	Particle Filter (Native)	PFN	L (1.08x)	10%
30	Rodinia	Leukocyte	LYTE	L (1.08x)	15%
31	Rodinia	LU Decomposition	LUD	L (1.05x)	64%
32	Parboil	Matrix Multiplication	MM	L (1.04x)	4%
33	CUDA SDK	StoreGPU	STO	L (1.02x)	3%
34	CUDA SDK	Coulombic Potential	CP	L (1.01x)	4%
35	CUDA SDK	N-Queens Solver	NQU	L (1.01x)	95%
36	Parboil	Distance-Cutoff CP	CUTP	L (1.01x)	2%
37	Rodinia	Heartwall	HW	L (1.01x)	9%
38	Parboil	Angular Correlation	TPAF	L (1.01x)	6%

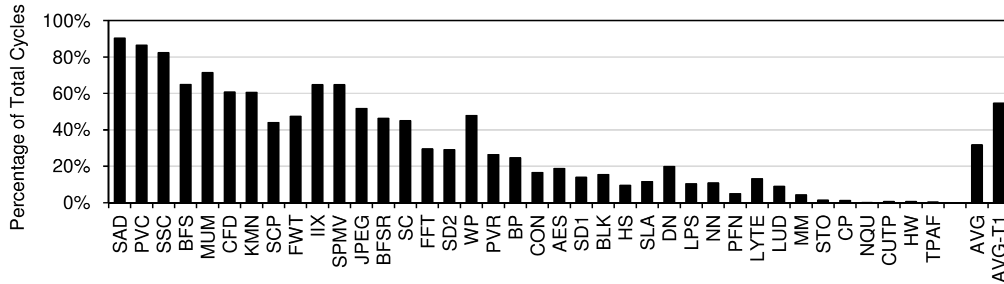


Figure 3.1: Fraction of total execution cycles (of all the cores) during which *all* the warps launched on a core are waiting for their respective data to come back from L2 cache/DRAM. This chapter defines the number of cycles where all warps are stalled due to memory as *MemoryBlockCycles*. AVG-T1 is the average (arithmetic mean) value across all Type-1 applications. AVG is the average value across all 38 applications.

3.2 Motivation and Workload Analysis

Round robin (RR) scheduling of warps causes almost all warps to execute the same long latency memory operation (with different addresses) at roughly the same time, as previous work has shown [8]. For the computation to resume in the warps and the core to become active again, these long-latency memory accesses need to be completed. This inefficiency of RR scheduling hampers the latency hiding capability of GPGPUs. To understand it further, let us consider 8 CTAs that need to be assigned to 2 cores (4 CTAs per core). According to the load-balanced CTA assignment policy described in Section 2, CTAs 1, 3, 5, 7 are assigned to core 1 and CTAs 2, 4, 6, 8 are assigned to core 2. With RR, warps associated with CTAs 1, 3, 5 and 7 are executed with equal priority on core 1 and are executed in a round-robin fashion. This execution continues until all the warps are blocked (when they need data from main memory). At this point, there may be no ready warps that can be scheduled, making core 1 inactive. Typically, this inactive time is very significant in memory intensive applications, as multiple requests are sent to the memory subsystem by many cores in a short period of time. This increases network and DRAM contention, which in turn increases queuing delays, leading to very high core inactive times.

To evaluate the impact of RR scheduling on GPGPU applications, this

chapter first characterizes the application set. This work quantifies how much IPC improvement each application gains if *all* memory requests magically hit in the L1 cache. This improvement, called PMEM, is depicted in Table 3.1, where the 38 applications are sorted in descending order of PMEM. Applications that have high PMEM ($>= 1.4\times$) are classified as Type-1, and the rest as Type-2. This work observed that the warps of highly memory intensive applications (Type-1) wait longer for their data to come back than warps of Type-2 applications. If this wait is eliminated, the performance of `SAD`, `PVC`, and `SSC` would improve by $6.39\times$, $4.99\times$ and $4.60\times$, respectively (as shown by the PMEM values for these applications).

Across Type-1 applications, average core inactive time (CINV) is 62% of the total execution cycles of all cores (Table 3.1). During this inactive time, no threads are being executed in the core. The primary reason behind this high core inactivity is *MemoryBlockCycles*, which is defined as the number of cycles during which all the warps in the core are stalled waiting for their memory requests to come back from L2 cache/DRAM (i.e., there are warps on the core but they are all waiting for memory). Figure 3.1 shows the fraction of *MemoryBlockCycles* of all the cores out of the total number of cycles taken to execute each application. Across all 38 applications, *MemoryBlockCycles* constitute 32% of the total execution cycles, i.e., 70% of the total inactive cycles. These results clearly highlight the importance of reducing the *MemoryBlockCycles* to improve the utilization of cores, and thus GPGPU performance.

Another major constituent of inactive cycles is *NoWarpCycles*, which is defined as number of cycles during which a core has *no warps* to execute, but an application has *not* completed its execution as some other cores are still executing warps. This might happen due to two reasons: (1) availability of a small number of CTAs within an application (due to an inherently small amount of parallelism) [9] or (2) the CTA load imbalance phenomenon [31], where some of the cores finish their assigned CTAs earlier than the others. This work finds that *NoWarpCycles* is prominent in `LUD` and `NQU`, which are Type-2 applications. Table 3.1 shows that although core inactive time is very high in `LUD` and `NQU` (64% and 95%, respectively), *MemoryBlockCycles* is very low (Figure 3.1).

Note that Type-1 applications are present across all modern workload suites like MapReduce, Parboil, Rodinia, and CUDA SDK, indicating that memory stalls are a fundamental bottleneck in improving the performance of these applications. This work finds that Type-1 applications are most affected by limited off-chip DRAM bandwidth, which leads to long memory stall times. The goal of this work is to devise new warp scheduling mechanisms to both reduce and tolerate long memory stall times in GPGPUs.

3.3 The OWL Scheduler

This section describes OWL, c(O)operative thread array a(W)are warp schedu(L)ing policy, which consists of four schemes: CTA-aware two-level warp scheduling, locality aware warp scheduling, bank-level parallelism aware warp scheduling, and opportunistic prefetching, where each scheme builds on top of the previous.

3.3.1 *CTA-Aware*: CTA-aware two-level warp scheduling

To address the problem posed by RR scheduling, a CTA-aware two-level warp scheduler is proposed, where all the available CTAs launched on a core (N CTAs) are divided into smaller *groups* of n CTAs. Assume that the size of each CTA is k warps (which is pre-determined for an application kernel). This corresponds to each group having $n \times k$ warps. *CTA-Aware* selects a single group (having n CTAs) and prioritizes the associated warps ($n \times k$) for execution over the remaining warps ($(N - n) \times k$) associated with the other group(s). Warps within the same group have equal priority and are executed in a round-robin fashion. Once all the warps associated with the first selected group are blocked due to the unavailability of data, a group switch occurs giving opportunity to the next CTA group for execution (and this process continues in a round-robin fashion among all the CTA groups). This is an effective way to hide long memory latencies, as now, a core can execute the group(s) of warps that are not waiting for memory while waiting for the data for the other group(s).

Table 3.2: Reduction in L1 miss rates with the proposed warp scheduling mechanisms over baseline RR scheduling.

#	App.	*CTA-Aware*	*CTA-Aware-Locality*	#	App.	*CTA-Aware*	*CTA-Aware-Locality*
1	SAD	6%	42%	11	SPMV	0%	8%
2	PVC	89%	90%	12	JPEG	0%	0%
3	SSC	1%	8%	13	BFSR	2%	16%
4	BFS	1%	17%	14	SC	0%	0%
5	MUM	1%	2%	15	FFT	1%	1%
6	CFD	1%	2%	16	SD2	0%	0%
7	KMN	27%	49%	17	WP	0%	0%
8	SCP	0%	0%	18	PVR	1%	2%
9	FWT	0%	0%	19	BP	0%	0%
10	IIX	27%	96%		**AVG-T1**	**8%**	**18%**

How to choose *n:* A group with n CTAs should have enough warps to keep the core pipeline busy in the absence of long latency operations [8]. Based on the GPU core's scheduling model described in Section 2, the minimum number of warps in a group is set to the number of pipeline stages (5 in this case). It means that, the minimum value of $n \times k$ should be 5. Since k depends on the GPGPU application kernel, the group size can vary for different application kernels. As each group can only have integral number of CTAs (n), the initial value of $n = 1$. If $n \times k$ is still smaller than the minimum number of warps in a group, n is increased by 1 until there are enough warps in the group for a particular application kernel. After the first group is formed, remaining groups are also formed in a similar fashion. For example, assume that the total number of CTAs launched on a core is $N = 10$. Also, assume that the number of pipeline stages is 5, and the number of warps in a CTA (k) is 2. In this case, the size of the first group (n) will be set to 3 CTAs, as now, a group will have 6 (3×2) warps, satisfying the minimum requirement of 5 (number of pipeline stages). The second group will follow the same method and have 3 CTAs. Now, note that the third group will have 4 CTAs to include the remaining CTAs. The third group cannot have only 3 CTAs ($n = 3$), because that will push the last CTA (10th CTA) to become the fourth group by itself, violating the minimum group size (in warps) requirement for the fourth group. This scheme is called as CTA-aware two-level scheduling (*CTA-Aware*), as the groups are formed taking CTA boundaries into consideration and a two-level scheduling policy is employed, where scheduling within a group (level 1) and switching among different groups (level 2) are both done in a round-robin fashion.

The need to be CTA-aware: Two types of data locality are primarily present in GPGPU applications [8,36,38]: (1) Intra-warp data locality, and (2) Intra-CTA (inter-warp) data locality. Intra-warp locality is due to the threads in a warp that share contiguous elements of an array, which are typically coalesced to the same cache line. This locality is exploited by keeping the threads of a warp together. Intra-CTA locality results from warps within the same thread-block sharing blocks or rows of data. Typically, data associated with one CTA is first moved to the on-chip memories and is followed by the computation on it. Finally, the results are written back to the main global memory. Since the difference between access latencies of on-chip and off-chip memories is very high [31], it is critical to optimally utilize the data brought on-chip and maximize reuse opportunities. Prioritizing some group of warps agnostic to the CTA boundaries may not utilize the data brought on-chip to the full extent (because it may cause eviction of data that is reused across different warps in the same CTA). Thus, it is important to be CTA-aware when forming groups.

3.3.2 *CTA-Aware-Locality*: Locality aware warp scheduling

Although *CTA-Aware* scheduling is effective in hiding the long memory fetch latencies, it does not effectively utilize the private L1 cache capacity associated with every core. Given the fact that L1 data caches of the state-of-the art GPGPU architectures are in the 16-64 KB range [4] (as well as in CMPs [39]), in most cases, the data brought by a *large number* of CTAs executing simultaneously does not fit into the cache (this is true for a majority of the memory-intensive applications). This hampers the opportunity of reusing the data brought by warps, eventually leading to a high number of L1 misses. In fact, this problem is more severe with the RR scheduling policy, where the number of simultaneously executing CTAs taking advantage of the caches in a given interval of time is more than that with the CTA-Aware scheduling policy. One might argue that, this situation can be addressed by increasing the size of L1 caches, but that would lead to (1) higher cache access latency, and (2) reduced hardware

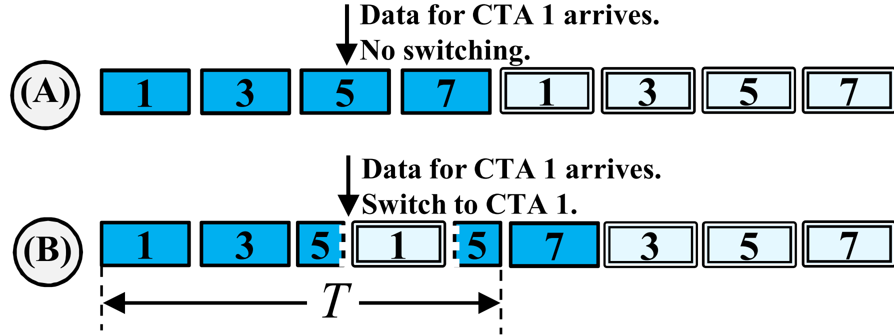


Figure 3.2: An illustrative example showing the working of (A) CTA-aware two-level warp scheduling (*CTA-Aware*) (B) Locality aware warp scheduling (*CTA-Aware-Locality*). Label in each box refers to the corresponding CTA number.

resources dedicated for computation, thereby hampering parallelism and the ability of the architecture to hide memory latency further.

Problem: In order to understand the problem with *CTA-Aware* scheme, consider Figure 3.2 (A). Without the loss of generality, let us assume that the group size is equal to 1. Further, assume that at core 1, CTA 1 belongs to group 1, CTA 3 belongs to group 2, etc., and each CTA has enough warps to keep the core pipeline busy ($1 \times k \geq$ number of pipeline stages). According to *CTA-Aware*, the warps of group 1 are prioritized until they are blocked waiting for memory. At this point, the warps of CTA 3 are executed. If the warps of CTA 1 become ready to execute (because their data arrives from memory) when the core is executing warps of CTA 5 (Figure 3.2 (A)), *CTA-Aware* will keep executing the warps of CTA 5 (and will continue to CTA 7 after that). It will not choose the warps from CTA 1 even though they are ready because it follows a strict round-robin policy among different CTAs. Thus, the data brought by the warps of CTA 1 early on (before they were stalled) becomes more likely to get evicted by other CTAs' data as the core keeps on executing the CTAs in a round-robin fashion. This strict round robin scheduling scheme allows larger number of threads to bring data to the relatively small L1 caches, thereby increasing cache contention due to the differences in the data sets of different CTAs and hampering the effective reuse of data in the caches. Although *CTA-Aware* performs better in utilizing L1 caches compared to RR (because it restricts the number of warps sharing the L1 cache simultaneously), it is far from optimal.

Solution: To achieve better L1 hit rates, this scheme strives to reduce the number of simultaneously executing CTAs taking advantage of L1 caches in a particular time interval. Out of N CTAs launched on a core, the goal is to *always* prioritize only one of the CTA groups of size n. n is chosen by the method described in Section 3.3.1. In general, on a particular core, *CTA-Aware-Locality* starts scheduling warps from group 1. If warps associated with group 1 (whose size is n CTAs) are blocked due to unavailability of data, the scheduler can schedule warps from group 2. This is essential to keep the core pipeline busy. However, as soon as any warps from group 1 are ready (i.e., their requested data has arrived), *CTA-Aware-Locality again* prioritizes these group 1 warps. If all warps belonging to group 1 have completed their execution, the next group (group 2) is chosen and is *always* prioritized. This process continues until all the launched CTAs finish their execution.

The primary motivation of using this scheme is that, in a particular time interval, only n CTAs are given higher priority to keep their data in the private caches such that they get the opportunity to reuse it. Since this scheme reduces contention and increases reuse in the L1 cache, it is called as locality aware warp scheduling (*CTA-Aware-Locality*). Note that, as n is closer to N, *CTA-Aware-Locality* degenerates into RR, as there can be only one group with N CTAs.

Typically, a GPGPU application kernel does not require fairness among the completion of different CTAs. CTAs can execute and finish in any order. The only important metric from the application's point of view is the total execution time of the kernel. A fair version of *CTA-Aware-Locality* can also be devised, where the CTA group with highest priority is changed (and accordingly, priorities of all groups will change) in a round-robin fashion (among all the groups) after a fixed interval of time. The design of such schemes is left as a part of the future work.

Figure 3.2 (B) shows how *CTA-Aware-Locality* works. Again, without loss of generality, let us assume that the group size is equal to 1. The *CTA-Aware-Locality* scheme starts choosing warps belonging to CTA 1 (belonging to group 1) once they become ready, unlike *CTA-Aware*, where scheduler keeps on choosing warps from CTA 5 (group 3), 7 (group 4) and so on.

In other words, this scheme *always* prioritize a small group of CTAs (in this case, group 1 with $n = 1$) and shift the priority to the next CTA only after CTA 1 completes its execution. During time interval T, this scheme observes that only 3 CTAs are executing and taking advantage of the private caches, contrary to 4 CTAs in the baseline system (Figure 3.2 (A)). This implies that a smaller number of CTAs gets the opportunity to use the L1 caches concurrently, increasing L1 hit rates and reducing cache contention.

Discussion: *CTA-Aware-Locality* aims to reduce the L1 cache misses. Table 3.2 shows the reduction in L1 miss rates (over baseline RR) when *CTA-Aware* and *CTA-Aware-Locality* schemes are incorporated. On average, for Type-1 applications, *CTA-Aware* reduces the overall miss rate by 8%. *CTA-Aware-Locality* is further able to reduce the overall miss rate (by 10%) by scheduling warps as soon as the data arrives for them, rather than waiting for their turn, thereby reducing the number of CTAs currently taking advantage of the L1 caches. With *CTA-Aware-Locality*, this scheme observes maximum benefits with Map-Reduce applications `PVC` and `IIX`, where the reduction in L1 miss rates is 90% and 96%, respectively, leading to significant IPC improvements (see Section 3.5). Since these applications are very memory intensive (highly ranked among Type-1 applications in Table 3.1) and exhibit good L1 data reuse within CTAs, they significantly benefit from *CTA-Aware-Locality*. Interestingly, this scheme finds that 8 out of 19 Type-1 applications show negligible reduction in L1 miss rates with both *CTA-Aware* and *CTA-Aware-Locality*. Detailed analysis shows that these applications do not exhibit significant cache sensitivity, thus, do not provide sufficient L1 data reuse opportunities. In `WP`, because of resource limitations posed by the baseline architecture (Section 2), there are only 6 warps that can be simultaneously executed. This restriction eliminates the possibility of getting benefits from *CTA-Aware-Locality*, as only one group (with 6 warps) can be formed, and no group switching/prioritization occurs.

3.3.3 *CTA-Aware-Locality-BLP*: BLP aware warp scheduling

The previous section discussed how *CTA-Aware-Locality* helps in hiding memory latency along with reducing L1 miss rates. This section proposes *CTA-Aware-Locality-BLP*, which not only incorporates the benefits of *CTA-Aware-Locality*, but also improves DRAM bank-level parallelism (BLP) [37].

Table 3.3: GPGPU application characteristics: *Consecutive CTA row sharing*: Fraction of consecutive CTAs (out of all CTAs) accessing the same DRAM row. *CTAs/Row*: Average number of CTAs accessing the same DRAM row.

#	App.	Cons. CTA row.share	CTAs/Row	#	App.	Cons. CTA row.share	CTAs/Row
1	SAD	42%	32	11	SPMV	98%	6
2	PVC	36%	2	12	JPEG	99%	16
3	SSC	20%	2	13	BFSR	71%	8
4	BFS	23%	5	14	SC	1%	2
5	MUM	17%	32	15	FFT	14%	5
6	CFD	81%	10	16	SD2	98%	35
7	KMN	66%	2	17	WP	93%	7
8	SCP	0%	1	18	PVR	38%	2
9	FWT	85%	2	19	BP	99%	4
10	IIX	36%	2		**AVG**	**64%**	**15**

Problem: In the study of 38 applications, the observation is that the same DRAM row is accessed (shared) by consecutive CTAs 64% of the time. Table 3.3 shows these row sharing percentages for all the Type-1 applications. This metric is determined by calculating the average fraction of consecutive CTAs (out of total CTAs) accessing the same DRAM row, averaged across all rows. For example, if a row is accessed by CTAs 1, 2, 3, and 4, its consecutive CTA row sharing percentage is deemed to be 100% (as all CTAs are consecutive). The observation is that for many GPGPU applications, the consecutive CTA row sharing percentages are very high (up to 99% in `JPEG`). For example, in Figure 2.1 (B), the observation is that the row sharing percentage is 100%, as CTA 1 opens 2 rows in Bank 1 (A(0,0) and A(0,1)) and Bank 2 (A(1,0) and A(1,1)); and, CTA 2 opens the same rows again as the data needed by it to execute is also mapped to the same rows. These high consecutive CTA row sharing percentages are not surprising, as CUDA programmers are encouraged to form CTAs such that the data required by the consecutive CTAs is mapped to the same DRAM row for high DRAM row locality, improving DRAM bandwidth utilization [1].

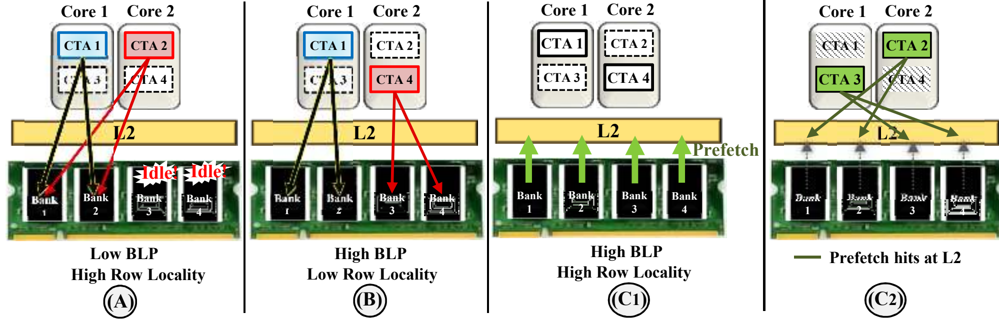


Figure 3.3: An example illustrating (A) the under-utilization of DRAM banks with *CTA-Aware-Locality*, (B) improved bank-level parallelism with *CTA-Aware-Locality-BLP*, (C1, C2) the positive effects of *Opportunistic Prefetching*.

Section 3.3.2 proposed *CTA-Aware-Locality* where a subset of CTAs (one group) is *always* prioritized over others. Although this scheme is effective at reducing cache contention and improving per-core performance, it takes decisions agnostic to inter-CTA row sharing properties. Consider a scenario where two consecutive CTA groups are scheduled on two different cores and are being *always* prioritized according to *CTA-Aware-Locality*. Given that the consecutive CTAs (in turn warps) share DRAM rows, the CTA groups access a *small* set of DRAM banks more frequently. This increases the queuing time at the banks and reduces the bank level parallelism (BLP). To understand this problem in-depth, let us revisit Figure 2.1 (C), which shows the row-major data layout of CTAs in DRAM [1]. The elements in row 0 of the matrix in Figure 2.1 (B) are mapped to a single row in bank 1, elements in row 1 are mapped to bank 2, and so on. To maximize row locality, it is important that the row that is loaded to a row buffer in a bank is utilized to the maximum, as row buffer hit latency (10 DRAM cycles (t_{CL})) is almost twice cheaper than row closed latency (22 DRAM cycles (t_{RCD} + t_{CL})), and almost three times cheaper than row conflict latency (32 DRAM cycles (t_{RP} + t_{RCD} + t_{CL})) [37]. *CTA-Aware-Locality* prioritizes CTA 1 (group 1) at core 1 and CTA 2 (also, group 1) at core 2. When both the groups are blocked, their memory requests access the same row in both bank 1 and bank 2, as CTA 1 and CTA 2 share the same rows (row sharing = 100%).

Figure 3.3 (A) depicts this phenomenon pictorially. Since consecutive CTAs (CTAs 1 and 2) share the same rows, prioritizing them in different cores enables

them to access these same rows concurrently, thereby providing high row buffer hit rate. Unfortunately, for the exact same reason, prioritizing consecutive CTAs in different cores leads to low BLP because all DRAM banks are not utilized as consecutive CTAs access the same banks (In Figure 3.3 (A), two banks stay idle). The goal is to develop a series of techniques that achieve both high BLP and high row buffer hit rate. First, a bank-level parallelism aware warp scheduling mechanism, *CTA-Aware-Locality-BLP*, is described which improves BLP at the expense of row locality.

Solution: To address the above problem, *CTA-Aware-Locality-BLP* is proposed, which not only inherits the positive aspects of *CTA-Aware-Locality* (better L1 hit rates), but also improves DRAM bank level parallelism. The key idea is to still *always* prioritize one CTA group in each core, but to ensure that *non-consecutive CTAs* (i.e., CTAs that do not share rows) are *always* prioritized in different cores. This improves the likelihood that the executing CTA groups (warps) in different cores access different banks, thereby improving bank level parallelism.

Figure 3.3 (B) depicts the working of *CTA-Aware-Locality-BLP* pictorially with an example. Instead of prioritizing consecutive CTAs (CTAs 1 and 2) in the two cores, *CTA-Aware-Locality-BLP* prioritizes non-consecutive ones (CTAs 1 and 4). This enables all four banks to be utilized concurrently, instead of two banks staying idle, which was the case with *CTA-Aware-Locality* (depicted in Figure 3.3 (A)). Hence, prioritizing non-consecutive CTAs in different cores leads to improved BLP. Note that this comes at the expense of row buffer locality, which will be restored with the next proposal, *Opportunistic Prefetching* (Section 3.3.4).

One way to implement the key idea of *CTA-Aware-Locality-BLP* is to prioritize different-numbered CTA groups in consecutive cores concurrently, instead of prioritizing the same-numbered CTA groups in each core concurrently. In other words, the warp scheduler in each core prioritizes, for example, the first CTA group in core 1, the second CTA group in core 2, the third CTA group in core 3, and so on. Since different-numbered CTA groups are unlikely to share DRAM rows, this technique is likely to maximize parallelism. Algorithm 1 more formally depicts the group formation and group priority assignment strategies for

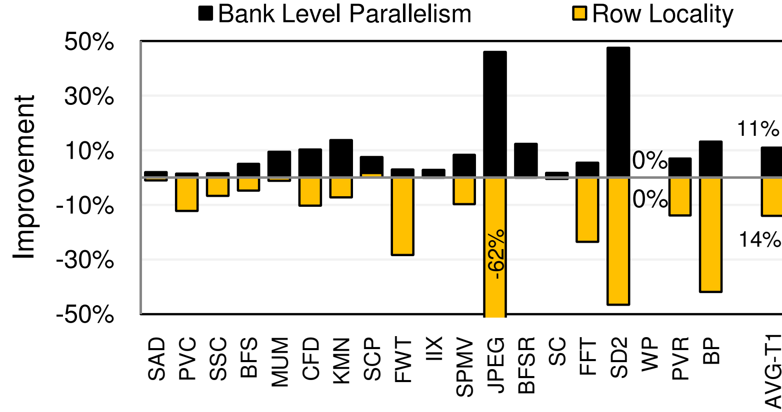


Figure 3.4: Effect of *CTA-Aware-Locality-BLP* on DRAM bank-level parallelism and row locality, compared to *CTA-Aware-Locality*.

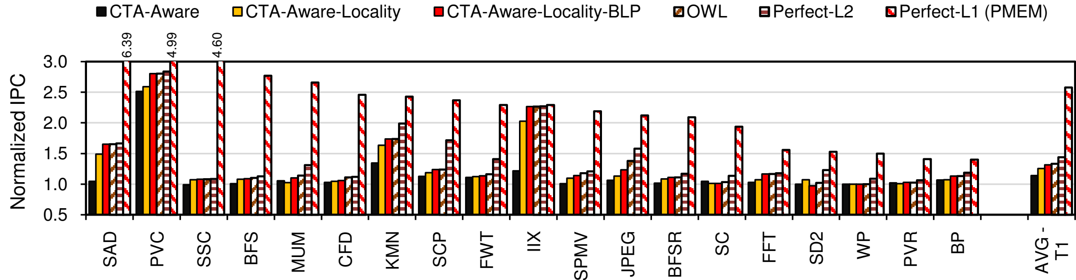


Figure 3.5: Performance impact of the schemes on Type-1 applications. Results are normalized to RR.

the three schemes discussed so far.

Discussion: Figure 3.4 shows the change in BLP and row buffer hit rate with *CTA-Aware-Locality-BLP* compared to *CTA-Aware-Locality*. Across Type-1 applications, there is an 11% average increase in BLP (AVG-T1), which not only reduces the DRAM queuing latency by 12%, but also reduces overall memory fetch latency by 22%. In `JPEG`, the BLP improvement is 46%. When *CTA-Aware-Locality-BLP* is incorporated, this scheme observes 14% average reduction in row locality among all Type-1 applications. Even though there is a significant increase in BLP, the decrease in row locality (e.g., in `JPEG`, `SD2`) is a concern, because reduced row locality adversely affects DRAM bandwidth utilization. To address this problem, the final scheme, memory-side *Opportunistic Prefetching* is proposed.

Algorithm 1 Group formation and priority assignment

▷ k is the number of warps in a CTA
▷ N is the number of CTAs scheduled on a core
▷ n is the minimum number of CTAs in a group
▷ *g_size* is the minimum number of warps in a group
▷ *g_core* is the number of groups scheduled on a core
▷ *num_cores* is the total number of cores in GPGPU
▷ *group_size*[i] is the group size (in number of CTAs) of the ith group
▷ *g_pri*[i][j] is the group priority of the ith group scheduled on the jth core.
▷ The lower the *g_pri*[i][j], the higher the scheduling priority. Once a group is chosen, the scheduler cannot choose warps from different group(s) unless all warps of the already-chosen group are blocked because of unavailability of data.

```
procedure FORM_GROUPS
    n ← 1
    while (n × k) < g_size do
        n ← n + 1
    g_core ← ⌊N/n⌋
    for g_num = 0 → (g_core − 1) do
        group_size[g_num] ← n
    if (N mod n) ≠ 0 then
        group_size[g_core − 1] ← group_size[g_core − 1] + (N mod n)

procedure CTA-AWARE
    FORM_GROUPS
    for core_ID = 0 → (num_cores − 1) do
        for g_num = 0 → (g_core − 1) do
            g_pri[g_num][core_ID] ← 0
                ▷ All groups have equal priority and executed in RR fashion.

procedure CTA-AWARE-LOCALITY
    FORM_GROUPS
    for core_ID = 0 → (num_cores − 1) do
        for g_num = 0 → (g_core − 1) do
            g_pri[g_num][core_ID] ← g_num

procedure CTA-AWARE-LOCALITY-BLP
    FORM_GROUPS
    for core_ID = 0 → (num_cores − 1) do
        for g_num = 0 → (g_core − 1) do
            g_pri[g_num][core_ID] ← (g_num − core_ID) mod g_core
```

3.3.4 Opportunistic Prefetching

The previous section discussed how *CTA-Aware-Locality-BLP* improves bank-level parallelism, but this comes at the cost of row locality. The evaluations show that, on average, 15 CTAs access the same DRAM row (shown under CTAs/row in Table 3.3). If these CTAs do not access the row when the row is fetched into the row buffer the first time, data in the row buffer will not be efficiently utilized. In fact, since *CTA-Aware-Locality-BLP* tries to schedule different CTAs that access the same row at *different times* to improve BLP, these different CTAs will need to re-open the row over and over before accessing it. Hence, large losses in row

locality are possible (as shown in Figure 3.4), which can hinder performance. The goal is to restore row buffer locality (and hence efficiently utilize an open row as much as possible) while keeping the benefits of improved BLP.

Solution: The observation is that prefetching cache blocks of an *already open row* can achieve this goal: if the prefetched cache blocks are later needed by other CTAs, these CTAs will find the prefetched data in the cache and hence do not need to access DRAM. As such, in the best case, even though CTAs that access the same row get scheduled at different times, they would not re-open the row over and over because opportunistic prefetching would prefetch all the needed data into the caches.

The key idea of *opportunistic prefetching* is to prefetch the so-far-unfetched cache lines in an already open row into the L2 caches, just before the row is closed (i.e., after all the demand requests to the row in the memory request buffer are served). This is called as *opportunistic* because the prefetcher, sitting in the memory controller, takes advantage of a row that was already opened by a demand request, in an opportunistic way. The prefetched lines can be useful for both currently executing CTAs, as well as, CTAs that will be launched later. Figure 3.3 (C1, C2) depicts the potential benefit of this scheme. In Figure 3.3 (C1), during the execution of CTAs 1 and 4, this proposal prefetches the data from the open rows that could potentially be useful for other CTAs (CTAs 2 and 3 in this example). If the prefetched lines are useful (Figure 3.3 (C2)), when CTAs 2 and 3 execute and require data from the same row, their requests will hit in the L2 cache and hence they will not need to access DRAM for the same row.

Implementation: There are two key design decisions in the opportunistic prefetcher: *what cache lines to prefetch* and *when to stop prefetching*. This section explores simple mechanisms to provide an initial study. However, any previously proposed prefetching method can be employed (as long as they generate requests to the same row that is open) – the exploration of such sophisticated techniques are left as a part of the future work.

What to prefetch? The evaluated prefetcher starts prefetching when there are no more demand requests to an open row. It sequentially prefetches the cache

lines that were *not* accessed by demand requests (after the row was opened the last time) from the row to the L2 cache slice associated with the memory controller.

When to stop opportunistic prefetching? Two possible schemes are studied, although there are many design choices possible. In the first scheme, the prefetcher stops immediately after a demand request to a *different* row arrives. The intuition is that a demand request is more critical than a prefetch, so it should be served immediately. However, this intuition may not hold true because servicing useful row-hit prefetch requests before row-conflict demand requests can eliminate future row conflicts, thereby improving performance (also shown by Lee et al. [40]). In addition, additional latency incurred by the demand request if prefetches were continued to be issued to the open row even after the demand arrives can be hidden in GPGPUs due to the existence of a large number of warps. Hence, it may be worthwhile to keep prefetching even after a demand to a different row arrives. Therefore, the second scheme prefetches at least a minimum number of cache lines (C) regardless of whether or not a demand arrives. The value of C is set to a value *lower* initially. The prefetcher continuously monitors the number of demand requests at the memory controller queue. If that number is less than a *threshold*, the value of C is set to a value *higher*. The idea is that if there are few demand requests waiting, it could be beneficial to keep prefetching. In the baseline implementation, the *lower* is set to 8, *higher* to 16, and *threshold* to the average number of pending requests at the memory controller. Section 3.5.1 explores sensitivity to these parameters. More sophisticated mechanisms are left as part of future work.

3.3.5 Hardware Overheads

CTA-aware scheduling: The nVIDIA warp scheduler has low warp-switching overhead [25] and warps can be scheduled according to their pre-determined priorities. The proposed schemes take advantage of such priority-based warp scheduler implementations already available in existing GPGPUs. Extra hardware is needed to dynamically calculate the priorities of the warps using the proposed schemes (Algorithm 1). In addition, every core should have a group

formation mechanism similar to Narasiman et al.'s proposal [8]. The RTL design of the hardware required for the proposed warp scheduler using the 65nm TSMC libraries in the Synopsys Design Compiler is synthesized. For a 28-core system, the area overhead is 0.18 mm^2.

Opportunistic prefetching: Opportunistic prefetching requires the prefetcher to know which cache lines in a row were already sent to the L2. To keep track of this for the currently-open row in a bank, n bits are added to the memory controller, corresponding to n cache lines in the row. When the row is opened, the n bits are reset. When a cache block is sent to the L2 cache from a row, its corresponding bit is set. For 8 MCs, each controlling 4 banks, with a row size of 32 cache blocks (assuming column size of 64B), the hardware overhead is 1024 bits ($8 \times 4 \times 32$ bits). The proposed second prefetching mechanism also requires extra hardware to keep track of the average number of pending requests at the memory controller. This range of this register is 0-127 and its value is computed approximately with the aid of shift registers.

3.4 Experimental Methodology

3.4.1 Workloads and Metrics

Application Suite: There is increasing interest in executing various general-purpose applications on GPGPUs in addition to the traditional graphics rendering applications [31, 41]. In this spirit, this work considers a wide range of emerging GPGPU applications implemented in CUDA, which include NVIDIA SDK [23], Rodinia [42], Parboil [43], MapReduce [44], and a few third party applications. In total, this work studies 38 applications. While Rodinia applications are mainly targeted for heterogeneous platforms, Parboil benchmarks primarily stress throughput computing focused architectures. Data-intensive MapReduce and third party applications are included for diversity. The applications are executed on GPGPU-Sim, which simulates the baseline architecture described in Table 3.4. The applications are run until completion or

Table 3.4: Baseline configuration

Shader Core Config.	1300MHz, 5-Stage Pipeline, SIMT width = 8
Resources / Core	Max. 1024 Threads, 32KB Shared memory, 32684 Registers
Caches / Core	32KB 8-way L1 Data cache, 8KB 4-way Texture cache 8KB 4-way Constant cache, 64B line size
L2 Cache	16-way 512 KB/Memory channel, 64B line size
Scheduling	Round-robin warp scheduling, (among ready warps), Load balanced CTA scheduling
Features	Memory coalescing enabled, 32 MSHRs/core, Immediate post dominator based branch divergence handling
Interconnect	2D Mesh (6×6; 28 cores + 8 Memory controllers), 650MHz, 32B channel width
DRAM Model	FR-FCFS (Maximum 128 requests/MC), 8MCs, 4 DRAM banks/MC, 2KB row size
GDDR3 Timing	800MHz, $t_{CL} = 10$, $t_{RP} = 10$, $t_{RC} = 35$, $t_{RAS} = 25$ $t_{RCD} = 12$, $t_{RRD} = 8$, $t_{CDLR} = 6$, $t_{WR} = 11$

for 1 billion instructions (whichever comes first), except for `IIX` where it is executed only for 400 million instructions because of infrastructure limitations.

Evaluation Metrics: In addition to using *instructions per cycle (IPC)* as the primary performance metric for evaluation, auxiliary metrics like bank level parallelism and row buffer locality are also considered. Bank level parallelism (BLP) is defined as the number of average memory banks that are accessed when there is at least one outstanding memory request at any of the banks [37, 45–47]. Improving BLP enables better utilization of DRAM bandwidth. Row-buffer locality (RBL) is defined as the average hit-rate of the row buffer across all memory banks [45]. Improving RBL increases the memory service rate and hence also enables better DRAM bandwidth utilization.

3.5 Experimental Results

This section evaluates the proposed scheduling and memory-side prefetching schemes with 19 Type-1 applications, where main memory is the main cause of core idleness.

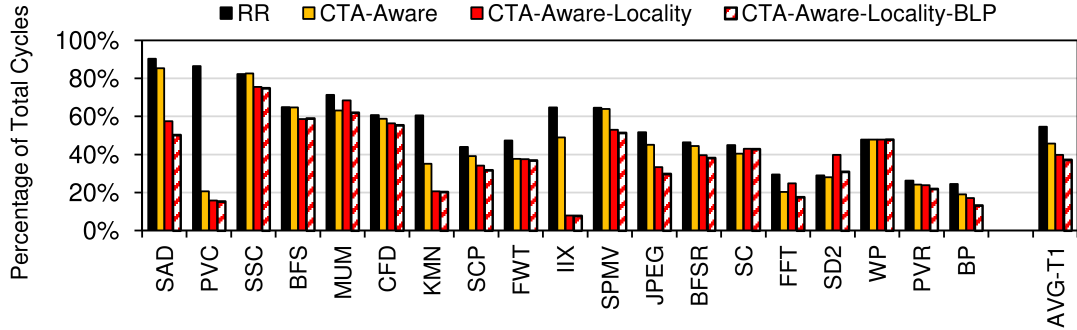


Figure 3.6: Impact of different scheduling schemes on *MemoryBlockCycles* for Type-1 applications. Results are normalized to the total execution cycles with baseline RR scheduling.

3.5.1 Performance Results

This section starts with evaluating the performance impact of the proposed scheduling schemes (in the order of their appearance in the chapter) against the *Perfect-L2* case, where all memory requests are L2 cache hits. This section also shows results with Perfect-L1 (PMEM), which is the ultimate upper bound of the proposed optimizations. Recall that each scheme builds on top of the previous.

Effect of *CTA-Aware*: Section 3.3.1 discussed that this scheme not only helps in hiding memory latency, but also partially reduces cache contention. Figure 3.5 shows the IPC improvements of Type-1 applications (normalized to RR). Figure 3.6 shows the impact of the scheduling schemes on *MemoryBlockCycles* as described in Section 3.2. On average (arithmetic mean), *CTA-Aware* provides 14% (9% harmonic mean (hmean), 11% geometric mean (gmean)) IPC improvement, with 9% reduction in memory waiting time (*MemoryBlockCycles*) over RR. The primary advantage comes from the reduction in L1 miss rates and improvement in memory latency hiding capability due to CTA grouping. Significant IPC improvements are observed in `PVC` (2.5×) and `IIX` (1.22×) applications, as the miss rate drastically reduces by 89% and 27%, respectively. As expected, significant performance improvements are not observed in `SD2`, `WP`, and `SPMV` as there is no reduction in miss rate compared to RR. Improvements are observed in `JPEG` (6%) and `SCP` (19%), even though there is no reduction in miss-rates (see Table 3.2). Most of the benefits in these benchmarks

are due to the better hiding of memory latency, which comes inherently from the CTA-aware two-level scheduling. It is further observed (not shown) that *CTA-Aware* achieves similar performance benefits compared to the recently proposed two-level warp scheduling [8]. In contrast to [8], by introducing awareness of CTAs, the *CTA-Aware* warp scheduling mechanism provides a strong foundation for the remaining three developed schemes.

Effect of *CTA-Aware-Locality*: The main advantage of this scheme is further reduced L1 miss rates. This scheme observes 11% average IPC improvement (6% decrease in *MemoryBlockCycles*) over *CTA-Aware*, and 25% (17% hmean, 21% gmean) over RR. This scheme observes 81% IPC improvement in `IIX`, primarily because of 69% in L1 miss rates. Because of the row locality and BLP trade-off (this scheme sacrifices BLP for increased row locality), this scheme observes that some applications may not attain optimal benefit from *CTA-Aware-Locality*. For example, in `SC`, IPC decreases by 4% and *MemoryBlockCycles* increases by 3% compared to *CTA-Aware*, due to a 26% reduction in BLP (7% increase in row locality). This scheme also observes similar results in `MUM`: 1% increase in row locality, 10% reduction in BLP, which causes 3% reduction in performance compared to *CTA-Aware*. In `SD2`, this scheme observes a 7% IPC improvement over *CTA-Aware* on account of a 14% increase in row-locality, with a 21% reduction in BLP. Nevertheless, the primary advantage of *CTA-Aware-Locality* is the reduced number of memory requests due to better cache utilization (Section 3.3.2), and as a result of this, this scheme also observes an improvement in DRAM bandwidth utilization due to reduced contention in DRAM banks.

Effect of *CTA-Aware-Locality-BLP*: This scheme strives to achieve better BLP at the cost of row locality. On average, this scheme observes 6% IPC (4% hmean, 4% gmean) improvement, and 3% decrease in *MemoryBlockCycles* over *CTA-Aware-Locality*. BLP increases by 11%, which also helps in the observed 22% reduction in overall memory fetch latency (12% reduction in queuing latency). In `SD2`, this scheme sees a significant increase in BLP (48%) over *CTA-Aware-Locality*, but performance still reduces (by 10%) compared to *CTA-Aware-Locality*, due to a 46% reduction in row locality. In contrast, in `JPEG`, the effects of the 62% reduction

in row locality is outweighed by the 46% increase in BLP, yielding a 10% IPC improvement over *CTA-Aware-Locality*. This shows that both row locality and BLP are important for GPGPU performance.

Combined Effect of OWL (Integration of *CTA-Aware-Locality-BLP* and opportunistic prefetching): The fourth bar from the left in Figure 3.5 shows the performance of the system with OWL. Four main conclusions can be drawn from this graph. First, using opportunistic prefetching on top of *CTA-Aware-Locality-BLP* consistently either improves performance or has no effect. Second, on average, even a simple prefetching scheme like ours can provide an IPC improvement of 2% over *CTA-Aware-Locality-BLP*, which is due to a 12% improvement in L2 cache hit rate. Overall, OWL achieves 19% (14% hmean, 17% gmean) IPC improvement over *CTA-Aware* and 33% (23% hmean, 28% gmean) IPC improvement over RR. Third, a few applications, such as `JPEG`, gain significantly (up to 15% in IPC) due to opportunistic prefetching, while others, such as `FWT`, `SPMV`, and `SD2`, gain only moderately (around 5%), and some do not have any noticeable gains, e.g., `SAD`, `PVC`, and `WP`. The variation seen in improvements across different applications can be attributed to their different memory latency hiding capabilities and memory access patterns. It is interesting to note that, in `SCP`, `FWT`, and `KMN`, some rows are accessed by only one or two CTAs. The required data in these rows are demanded when they are opened for the first time. In these situations, even if we prefetch all the remaining lines, significant improvements are not observed. Fourth, the scope of improvement available for opportunistic prefetching over *CTA-Aware-Locality-BLP* is limited: *Perfect-L2* can provide only 13% improvement over *CTA-Aware-Locality-BLP*. This is mainly because if an application inherently has a large number of warps ready to execute, the application will also be able to efficiently hide the long memory access latency. This scheme observes that prefetching might not be beneficial in these applications even if the prefetch-accuracy is 100%.

This section concludes that the proposed schemes are effective at improving GPGPU performance by making memory less of a bottleneck. As a result, OWL enables the evaluated GPGPU to have performance within 11% of a hypothetical

GPGPU with a perfect L2.

3.5.2 Sensitivity Studies

This section describes the critical sensitivity studies performed related to group size, DRAM configuration and opportunistic prefetching.

Sensitivity to group size: Section 3.3.1 mentioned that the minimum number of warps in a group should be at least equal to the number of pipeline stages. Narasiman et al. [8] advocated that, if the group size is too small, the data fetched in DRAM row buffers is not completely utilized, as fewer warps are prioritized together. If the group size is too large, the benefits of two-level scheduling diminishes. Figure 3.7 shows the effect of the group size on performance. The results are normalized to RR and averaged across all Type-1 applications. The observation is that when the minimum group size is 8 warps, best IPC improvements (14% for *CTA-Aware*, 25% for *CTA-Aware-Locality* and 31% for *CTA-Aware-Locality-BLP* over RR) are achieved, and thus, throughout this work, a minimum group size of 8 is used, instead of 5 (which is the number of pipeline stages).

Sensitivity to the number of DRAM banks: Figure 3.8 shows the change in performance of *CTA-Aware-Locality-BLP* with the number of DRAM banks per MC. The observation is that as the number of banks increases, the effectiveness of *CTA-Aware-Locality-BLP* also increases. This is because having additional banks enables more benefits from exposing higher levels of BLP via the proposed techniques. As a result, the performance improvement of the proposal is 2% higher with 8 banks per MC than with 4 banks per MC (baseline system). It is envisioned that the proposed techniques are likely to become more effective in future systems with more banks.

Sensitivity to Opportunistic Prefetching Parameters: Experiments are performed with all combinations of *lower* and *upper* values for the prefetch degree in the range of 0 (no-prefetching) to 32 (prefetching all the columns in a row) with a step size of 8. The value of *threshold* is also varied similarly, along with the case

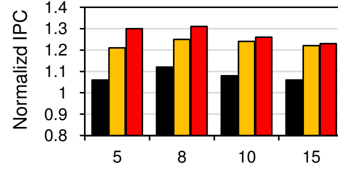


Figure 3.7: Sensitivity of IPC to group size (normalized to RR).

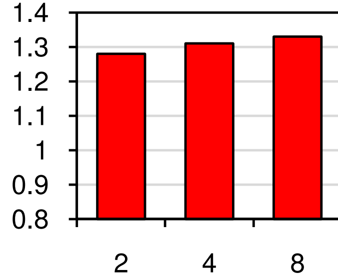


Figure 3.8: Sensitivity of IPC to the number of banks (normalized to RR).

when it is equal to the average memory controller queue length. Figure 3.9 shows the best case values achieved across all evaluated combinations (*Best OWL*). The average performance improvement achievable by tuning these parameter values is only 1% (compare *Best OWL* vs. OWL). This can possibly be achieved by implementing a sophisticated prefetcher that can dynamically adjust its parameters based on the running application's characteristics, which comes at the cost of increased hardware complexity. The design of such application-aware memory-side prefetchers are left as a part of the future work, along with more sophisticated techniques to determine what parts of a row to prefetch.

3.6 Related Work

This section briefly describes and compares to the closely related works.

Scheduling in GPGPUs: The two-level warp scheduling mechanism proposed by Narasiman et al. [8] increases the core utilization by creating larger warps and employing a two-level warp scheduling scheme. This mechanism is not aware of CTA boundaries. This work proposes CTA-aware warp scheduling policies, which improve not only L1 hit rates, but also DRAM bandwidth

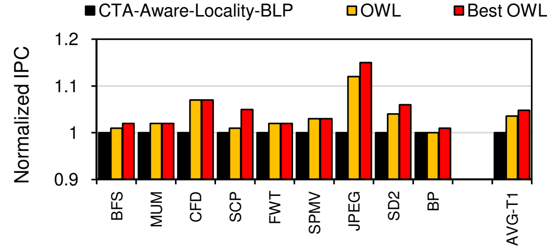


Figure 3.9: Prefetch degree and throttling threshold sensitivity.

utilization. This work finds that the combination of all of the techniques, OWL, provides approximately 19% higher performance than two-level warp scheduling. Gebhart et al. [48] also proposed a two-level warp scheduling technique. Energy reduction is the primary purpose of their approach. Even though this work does not evaluate it, OWL is also likely to provide energy benefits as reduced execution time (with low hardware overhead) is likely to translate into reduced energy consumption. Concurrent work by Rogers et al. [38] proposed a cache-conscious warp scheduling policy. Their work improves L1 hit rates for cache-sensitive applications. OWL not only reduces cache contention, but also improves DRAM bandwidth utilization for a wide range of applications. Work from Kayiran et al. [9] dynamically estimates the amount of thread-level parallelism that would improve GPGPU performance by reducing cache and DRAM contention. The approach of this work is orthogonal to theirs as the CTA-aware scheduling techniques improve cache and DRAM utilization for a *given* amount of thread-level parallelism.

BLP and Row Locality: Bank-level parallelism and row buffer locality are two important characteristics of DRAM performance. Several memory request scheduling [34, 37, 45, 46, 49–51] techniques have been proposed to improve one or both within the context of multi-core, GPGPU, and heterogeneous CPU-GPU systems [51]. This work can be combined with these approaches. Mutlu and Moscibroda [37] describe parallelism-aware batch scheduling, which aims to preserve each thread's BLP in a multi-core system. Hassan et al. [52] suggest that optimizing BLP is more important than improving row buffer hits, even though there is a trade-off. This chapter uses this observation to focus on enhancing BLP, while restoring the lost row locality by memory-side prefetching.

This is important because, in some GPGPU applications, both BLP and row locality are important. Similar to this work, Jeong et al. [53] observe that both BLP and row locality are important for maximizing benefits in multi-core systems. The memory access scheduling proposed by Yuan et al. [49] restores the lost row access locality caused by the in-order DRAM scheduler, by incorporating an arbitration mechanism in the interconnection network. The staged memory scheduler of Ausavarungnirun et al. [51] batches memory requests going to the same row to improve row locality while also employing simple in-order request scheduling at the DRAM banks. Lakshminarayana et al. [50] propose a potential function that models the DRAM behavior in GPGPU architectures and a SJF DRAM scheduling policy. The scheduling policy essentially chooses between SJF and FR-FCFS at run-time based on the number of requests from each thread and their potential of generating a row buffer hit. This work proposes low-overhead *warp scheduling and prefetching* schemes to improve *both* row locality and BLP. Exploration of the combination of the warp scheduling techniques with memory request scheduling and data partitioning techniques is a promising area of future work.

Data Prefetching: This work uses a memory-side prefetcher in GPUs. The opportunistic prefetcher complements the CTA-aware scheduling schemes by taking advantage of open DRAM rows. The most relevant work on hardware prefetching in GPUs is the L1 prefetcher proposed by Lee et al. [35]. Carter et al. [54] present one of the earliest works done in the area of memory-side prefetching in the CPU domain. Many other prefetching mechanisms (e.g., [55–57]) have been proposed within the context of CPU systems. The contribution in this work is a specific prefetching algorithm (in fact, the proposal can potentially use the algorithms proposed in literature), but to employ the idea prefetching in conjunction with new BLP-aware warp scheduling techniques to restore row buffer locality and improve L1 hit rates in GPGPUs.

3.7 Chapter Summary

This work proposes a new warp scheduling policy, OWL, to enhance GPGPU performance by overcoming the resource under-utilization problem caused by long latency memory operations. The key idea in OWL is to take advantage of characteristics of cooperative thread arrays (CTAs) to concurrently improve cache hit rate, latency hiding capability, and DRAM bank parallelism in GPGPUs. OWL achieves these benefits by 1) selecting and prioritizing a group of CTAs scheduled on a core, thereby improving both L1 cache hit rates and latency tolerance, 2) scheduling CTA groups that likely do not access the same memory banks on different cores, thereby improving DRAM bank parallelism, and 3) employing opportunistic memory-side prefetching to take advantage of already-open DRAM rows, thereby improving both DRAM row locality and cache hit rates. The experimental evaluations on a 28-core GPGPU platform demonstrate that OWL is effective in improving GPGPU performance for memory-intensive applications: it leads to 33% IPC performance improvement over the commonly-employed baseline round-robin warp scheduler, which is not aware of CTAs.

Chapter 4

Prefetch Aware Warp Scheduling Techniques

This chapter presents techniques that coordinate the thread scheduling and prefetching decisions in a General Purpose Graphics Processing Unit (GPGPU) architecture to better tolerate long memory latencies. This chapter demonstrates that existing warp scheduling policies in GPGPU architectures are unable to effectively incorporate data prefetching. The main reason is that they schedule consecutive warps, which are likely to access nearby cache blocks and thus prefetch accurately for one another, back-to-back in consecutive cycles. This either 1) causes prefetches to be generated by a warp too close to the time their corresponding addresses are actually demanded by another warp, or 2) requires sophisticated prefetcher designs to correctly predict the addresses required by a future "far-ahead" warp while executing the current warp.

This chapter proposes a new *prefetch-aware* warp scheduling policy that overcomes these problems. The key idea is to separate in time the scheduling of consecutive warps such that they are not executed back-to-back. This policy not only enables a simple prefetcher to be effective in tolerating memory latencies but also improves memory bank parallelism, even when prefetching is not employed. Experimental evaluations across a diverse set of applications on a 30-core simulated GPGPU platform demonstrate that the prefetch-aware warp

scheduler provides 25% and 7% average performance improvement over baselines that employ prefetching in conjunction with, respectively, the commonly-employed round-robin scheduler or the recently-proposed two-level warp scheduler. Moreover, when prefetching is not employed, the prefetch-aware warp scheduler provides higher performance than both of these baseline schedulers as it better exploits memory bank parallelism.

4.1 Introduction

The memory subsystem is a critical determinant of performance in General Purpose Graphics Processing Units (GPGPUs). And, it will become more so as more compute resources continue to get integrated into the GPGPUs and as the GPGPUs are placed onto the same chip with CPU cores and other accelerators, resulting in higher demands for memory performance.

Traditionally, GPGPUs tolerate long memory access latencies by concurrently executing many threads. These threads are grouped into fixed-sized batches known as *warps* or *wavefronts*. Threads within a warp share the same instruction stream and execute the same instruction at the same time, forming the basis for the term *single instruction multiple threads*, SIMT [3, 25, 26]. The capability to rapidly context switch between warps in the state-of-the-art GPGPUs allows the execution of other warps when one warp stalls (on a long-latency memory operation), thereby overlapping memory access latencies of different warps. The effectiveness of the *warp scheduling policy*, which determines the order and time in which different warps are executed, has a critical impact on the memory latency tolerance and memory bandwidth utilization, and thus the performance, of a GPGPU. An effective warp scheduling policy can facilitate the concurrent execution of many warps, potentially enabling all compute resources in a GPGPU to be utilized without idle cycles (assuming there are enough threads).

Unfortunately, commonly-employed warp schedulers are ineffective at tolerating long memory access latencies, and therefore lead to significant underutilization of compute resources, as shown in previous work [7–9,35]. The commonly-used round-

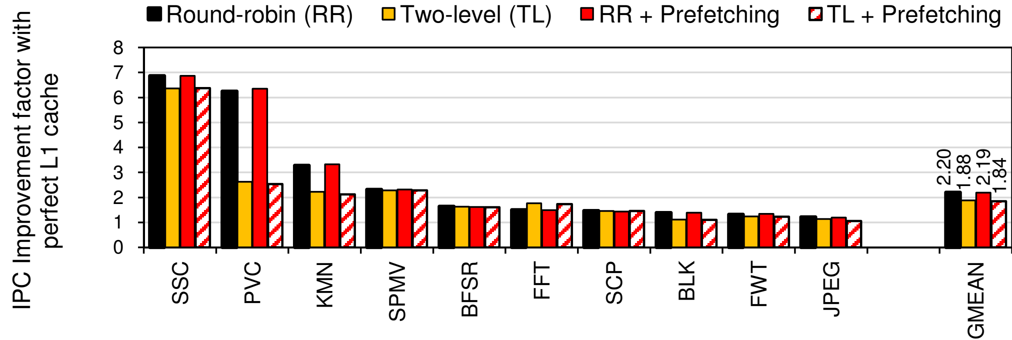


Figure 4.1: IPC improvement when L1 cache is made perfect on a GPGPU that employs (1) round-robin (RR) warp scheduling policy, (2) two-level (TL) warp scheduling policy, (3) data prefetching together with RR, and (4) data prefetching together with TL. Section 4.4 describes the evaluation methodology and workloads.

robin (RR) policy schedules *consecutive warps*[1] in consecutive cycles, assigning all warps equal priority in scheduling. As a result of this policy, most of the warps arrive at a long-latency memory operation roughly at the same time [7, 8]. The core can therefore become *inactive* as there may be no warps that are *not* stalling due to a memory operation. To overcome this disadvantage of the RR policy, the two-level (TL) warp scheduling policy [8] was proposed. This scheduler divides all warps into *fetch groups* and prioritizes warps from a single fetch group until they reach a long-latency operation. When one fetch group stalls due to a long-latency memory operation, the next fetch group is scheduled. The scheduling policy of warps within a fetch group is round robin, and so is the scheduling policy across fetch groups. The primary insight is that each fetch group reaches the long-latency operations at different points in time. As a result, when warps in one fetch group are stalled on memory, warps in another fetch group can continue to execute, thereby effectively tolerating the memory latency in a fetch group by performing computation in another.

The leftmost two bars for each application in Figure 4.1 show the potential performance improvement achievable if the L1 caches were perfect on 1) a GPGPU that employs the RR policy and 2) the same GPGPU but with the TL policy, across a set of ten diverse applications. The two-level warp scheduler reduces the

[1]Two warps that have consecutive IDs are called consecutive warps. Due to the way data is usually partitioned across different warps, it is very likely that consecutive warps access nearby cache blocks [1, 7, 8, 36].

performance impact of L1 cache misses on performance, as shown by the lower IPC improvement obtained by making the L1 data cache perfect on top of the TL policy. However, a significant performance potential remains: IPC would improve by 1.88× if the L1 cache were perfect, showing that there is significant potential for improving memory latency tolerance in GPGPU systems.

Data prefetching, commonly employed in CPU systems (e.g., [56, 58–60]), is a fundamental latency hiding technique that can potentially improve the memory latency tolerance of GPGPUs and achieve the mentioned performance potential.[2] However, this work finds that employing prefetching naively does not significantly improve performance in GPGPU systems. The effect of this is shown quantitatively in the rightmost two bars for each application in Figure 4.1: employing a data prefetcher (based on a spatial locality detector [61], as described in detail in Section 4.3.2) with either the RR or the TL scheduling policy does not significantly improve performance.

This chapter observes that existing warp scheduling policies in GPGPUs are unable to effectively incorporate data prefetching mechanisms. The main reason is that they schedule consecutive warps, which are likely to access nearby cache blocks and thus prefetch accurately for one another, back to back in consecutive cycles. Consider the use of a simple streaming prefetcher with the RR policy: when one warp stalls and generates its demand requests, the prefetcher generates requests for the next N cache blocks. Soon after, and long before these prefetch requests complete, the succeeding warps get scheduled and likely require these cache blocks due to the high spatial locality between consecutive warps [1, 7, 8, 36]. Unfortunately, these succeeding warps cannot take advantage of the issued prefetches because the prefetch requests were issued *just before* the warps were scheduled. The TL scheduling policy suffers from the same problem: since consecutive warps within a fetch group are scheduled consecutively, the prefetches issued by a preceding warp are immediately demanded by the succeeding one, and as a result, even though prefetches are accurate, they do not provide performance improvement as they are too late. One could potentially

[2] A form of data prefetching was developed in [35] for GPGPUs, where one warp prefetches data for another.

solve this problem by designing a prefetcher that prefetches data for a "far-ahead", non-consecutive warp that will be scheduled far in the future while executing the current warp such that the prefetch requests are complete by the time the "far-ahead" warp gets scheduled. Unfortunately, this requires a more sophisticated prefetcher design: accurately predicting the addresses required by a "far-ahead", non-consecutive warp is fundamentally more difficult than accurately predicting the addresses required by the next consecutive warp due to two reasons: 1) non-consecutive warps do not exhibit high spatial locality among each other [7, 8]; in fact, the sets of addresses required by two non-consecutive warps may have no relationship with each other, 2) the time at which the far-ahead warp gets scheduled may vary depending on the other warp scheduling decisions that happen in between the scheduling of the current and the far-ahead warps.

This work observes that orchestrating the warp scheduling and prefetching decisions can enable a simple prefetcher to provide effective memory latency tolerance in GPGPUs. To this end, this work proposes a new *prefetch-aware (PA)* warp scheduling policy. The core idea is to separate in time the scheduling of consecutive warps such that they are *not* executed back-to-back, i.e., one immediately after another. This way, when one warp stalls and generates its demand requests, a simple prefetcher can issue prefetches for the next N cache blocks, which are likely to be completed by the time the consecutive warps that need them are scheduled. While the prefetch requests are in progress, other non-consecutive warps that do not need the prefetched addresses are executed.

The prefetch-aware warp scheduling policy is based on the TL scheduler, with a key difference in the way the fetch groups are formed. Instead of placing consecutive warps in the same fetch group as the TL scheduler does, the PA scheduler places *non-consecutive* warps in the same fetch group. In addition to enabling a simple prefetcher to be effective, this policy also improves memory bank-level parallelism because it enables non-consecutive warps, which are likely to access different memory banks due to the lack of spatial locality amongst each other, to generate their memory requests concurrently. Note that the PA scheduler causes a loss in row buffer locality due to the exact same reason, but

the use of simple spatial prefetching can restore the row buffer locality by issuing prefetches to an already-open row.

This work coordinates thread scheduling and prefetching decisions for improving memory latency tolerance in GPGPUs. The major **contributions** are as follows:

- This work shows that the state-of-the-art warp scheduling policies in GPGPUs are unable to effectively take advantage of data prefetching to enable better memory latency tolerance.

- This work proposes a *prefetch-aware* warp scheduling policy, which not only enables prefetching to be more effective in GPGPUs but also improves memory bank-level parallelism even when prefetching is not employed.

- This work shows that the proposed *prefetch-aware* warp scheduler can work in tandem with a simple prefetcher that uses spatial locality detection [61].

- The conducted experimental results show that this orchestrated scheduling and prefetching mechanism achieves 25% and 7% average IPC improvement across a diverse set of applications, over state-of-the-art baselines that employ the same prefetcher with the round-robin and two-level warp schedulers, respectively. Moreover, when prefetching is *not* employed, the proposed *prefetch-aware* warp scheduler provides respectively 20% and 4% higher IPC than these baseline schedulers as it better exploits memory bank parallelism.

4.2 Interaction of Scheduling and Prefetching: Motivation and Basic Ideas

This section first illustrates the shortcomings of state-of-the-art warp scheduling policies in integrating data prefetching effectively. Then it illustrates the proposal, the prefetch-aware warp scheduling policy, which aims to orchestrate scheduling and prefetching.

The illustrations revolve around Figure 4.2. The left portion of the figure shows

the execution and memory request timeline of a set of eight warps, W1-W8, with eight different combinations of warp scheduling and prefetching. The right portion shows the memory requests generated by these eight warps and the addresses and banks accessed by their memory requests. Note that consecutive warps W1-W4 access a set of consecutive cache blocks X, X+1, X+2, X+3, mapped to DRAM Bank 1, whereas consecutive warps W5-W8 access another set of consecutive cache blocks Y, Y+1, Y+2, Y+3, mapped to DRAM Bank 2. The legend of the figure, shown on the rightmost side, describes how different acronyms and shades should be interpreted.

4.2.1 Shortcomings of the State-of-the-Art Warp Schedulers

4.2.1.1 Round-robin (RR) warp scheduling

Figure 4.2 (A) shows the execution timeline of the eight warps using the commonly-used round-robin (RR) warp scheduling policy, without data prefetching employed. As described before in Section 4.1, since all warps make similar amounts of progress due to the round robin nature of the policy, they generate their memory requests (D1-D8) roughly at the same time, and as a result stall roughly at the same time (i.e., at the end of the first compute phase, C1, in the figure). Since there are no warps to schedule, the core remains idle until the data for at least one of the warps arrives, which initiates the second compute phase, C2, in the figure. The stall time between the two compute phases is called as *MemoryBlockCycles*. Figure 4.2 (A') shows the effect of RR scheduling on the DRAM system. The RR policy exploits both row buffer locality (RBL) and bank-level parallelism (BLP) in DRAM because: i) as consecutive warps W1-W4 (W5-W8) access consecutive cache blocks, their requests D1-D4 (D5-D8) access the same row in Bank 1 (Bank 2), thereby exploiting RBL, ii) warp sets W1-W4 and W5-W8 access different banks and generate their requests roughly at the same time, thereby exploiting BLP.

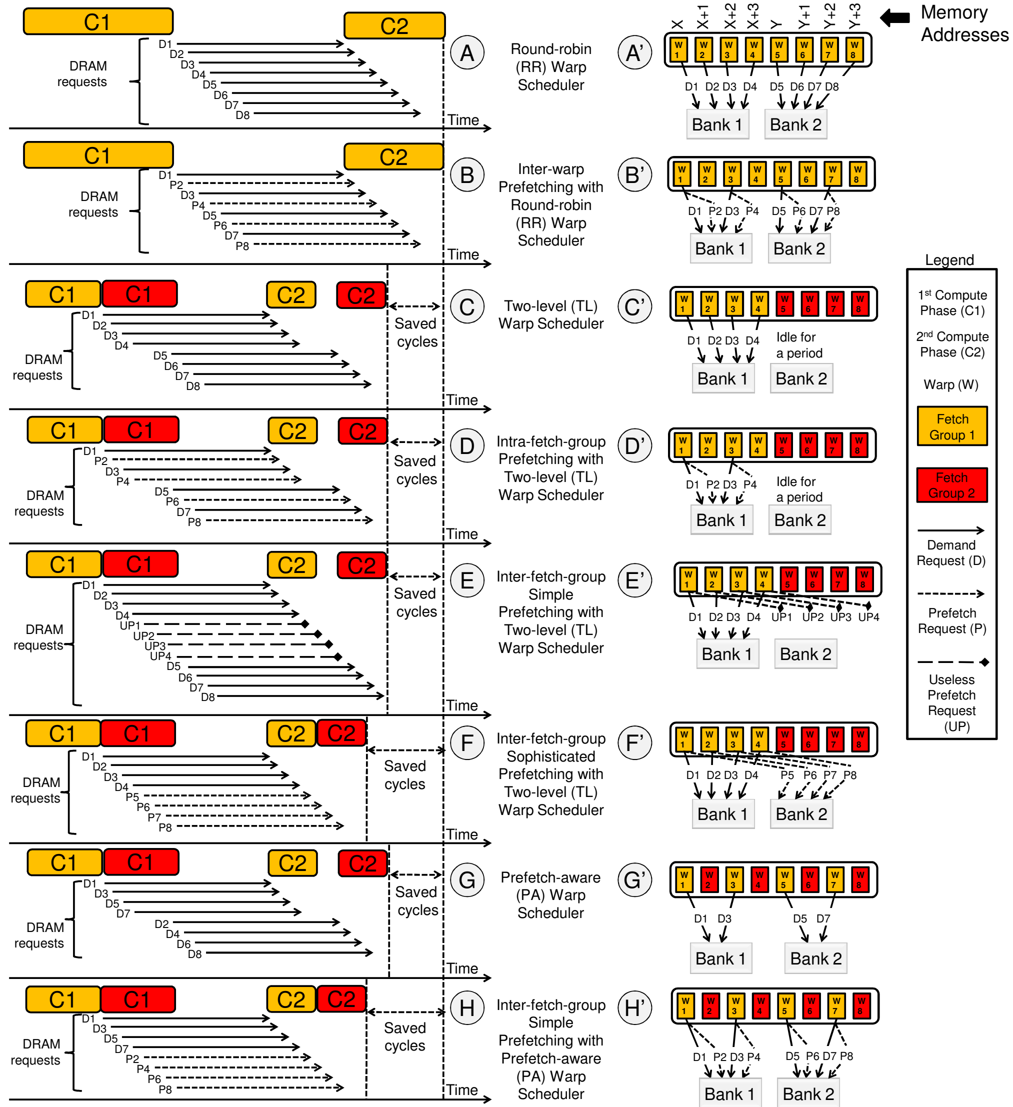


Figure 4.2: An illustrative example showing the working of various scheduling and prefetching mechanisms, motivating the need for the design of the prefetch-aware warp scheduler.

4.2.1.2 Round-robin (RR) warp scheduling and inter-warp prefetching

Figures 4.2 (B) and (B') show the execution timeline and DRAM state when an inter-warp prefetcher is incorporated on top of the baseline RR scheduler. The goal of the inter-warp prefetcher is to reduce *MemoryBlockCycles*. When a warp generates a memory request, the inter-warp prefetcher generates a prefetch request for the next warp (in this example, for the next sequential cache block).

The prefetched data is placed in a core's private L1 data cache, which can serve the other warps. For example, the issuing of demand request D1 (to cache block X) by W1 triggers a prefetch request P1 (to cache block X+1) which will be needed by W2. Figure 4.2 (B) shows that, although the prefetch requests are accurate, adding prefetching on top of RR scheduling does not improve performance (i.e., reduce *MemoryBlockCycles*). This is because the next-consecutive warps get scheduled soon after the prefetch requests are issued and generate demand requests for the same cache blocks requested by the prefetcher, long before the prefetch requests are complete (e.g., W2 generates a demand request to block X+1 right after W1 generates a prefetch request for the same block). Hence, this example illustrates that RR scheduling cannot effectively take advantage of simple inter-warp prefetching.

4.2.1.3 Two-level (TL) warp scheduling

Figure 4.2 (C) shows the execution timeline of the eight warps using the recently-proposed two-level round-robin scheduler [8], which was described in Section 4.1. The TL scheduler forms smaller fetch groups out of the concurrently executing warps launched on a core and prioritizes warps from a single fetch group until they reach long-latency operations. The eight warps in this example are divided into two fetch groups, each containing 4 warps. The TL scheduler first executes warps in group 1 (W1-W4) until these warps generate their memory requests D1-D4 and stall. After that, the TL scheduler switches to executing warps in group 2 (W5-W8) until these warps generate their memory requests D5-D8 and stall. This policy thus overlaps some of the latency of memory requests D1-D4 with computation done in the second fetch group, thereby reducing *MemoryBlockCycles* and improving performance compared to the RR policy, as shown via *Saved cycles* in Figure 4.2 (C). Figure 4.2 (C') shows the effect of TL scheduling on the DRAM system. The TL policy exploits row buffer locality but it does not fully exploit bank-level parallelism because there are times when a bank remains idle without requests because not all warps generate memory requests at roughly the same time: during the first compute period of fetch group 2, bank 2 remains idle.

4.2.1.4 Two-level (TL) warp scheduling and intra-fetch-group prefetching

Figures 4.2 (D) and (D') show the effect of using an *intra-fetch-group prefetcher* along with the TL policy. The prefetcher used is the same inter-warp prefetcher as the one described in Section 4.2.1.2, where one warp generates a prefetch request for the next-consecutive warp in the same fetch group. Adding this prefetching mechanism on top of TL scheduling does not improve performance since the warp that is being prefetched for gets scheduled immediately after the prefetch is generated. This limitation is the same as what it has been observed when adding simple inter-warp prefetching over the RR policy in Section 4.2.1.2.

This work concludes that prefetching for warps within the same fetch group is ineffective because such warps will be scheduled soon after the generation of prefetches.[3] Henceforth, this work assumes that the prefetcher employed is an inter-fetch-group prefetcher.

4.2.1.5 Two-level (TL) warp scheduling and inter-fetch-group prefetching

The idea of an *inter-fetch-group* prefetcher is to prefetch data for the next (or a future) fetch group. There are two cases. First, if the prefetch requests are accurate, in which case a sophisticated prefetching mechanism is required as the addresses generated by the next fetch group may not have any easy-to-predict relationship to the addresses generated by the previous one, they can potentially improve performance because the prefetches would be launched long before they are needed. However, if the prefetches are inaccurate, which could be the case with a simple prefetcher, such an inter-fetch group prefetcher will issue useless prefetches.

Figures 4.2 (E) and (E') depict the latter case: they show the effect of using a *simple inter-fetch-group* prefetcher along with the TL policy. Since the simple

[3]Note that in RR warp scheduling, although there is no fetch group formation, all the launched warps can be considered to be part of a single large fetch group.

prefetcher cannot accurately predict the addresses to be accessed by fetch group 2 (Y, Y+1, Y+2, Y+3) when observing the accesses made by fetch group 1 (X, X+1, X+2, X+3), it ends up issuing useless prefetches (UP1-UP4). This not only wastes valuable memory bandwidth and cache space, but may also degrade performance (although the example in the figure shows no performance loss).

Figures 4.2 (F) and (F') depict the former case: they show the effect of using a *sophisticated* inter-fetch-group prefetcher along with the TL policy. Since the sophisticated prefetcher *can* accurately predict the addresses to be accessed by fetch group 2 (Y, Y+1, Y+2, Y+3) when observing the accesses made by fetch group 1 (X, X+1, X+2, X+3), it improves performance compared to the TL scheduler without prefetching. Unfortunately, as explained in Section 4.1, designing such a sophisticated prefetcher is fundamentally more difficult than designing a simple (e.g., next-line [62] or streaming [63]) prefetcher because non-consecutive warps (in different fetch groups) do not exhibit high spatial locality among each other [7, 8]. In fact, the sets of addresses required by two non-consecutive warps *may* have no predictable relationship with each other, potentially making such sophisticated prefetching practically impossible.

The next two sections illustrate the working of the proposed prefetch-aware warp scheduling policy which enables the benefits of a sophisticated prefetcher without requiring the implementation of one.

4.2.2 Orchestrating Warp Scheduling and Data Prefetching

4.2.2.1 Prefetch-aware (PA) warp scheduling

Figures 4.2 (G) and (G') show the execution timeline, group formation, and DRAM state of the eight warps using the proposed prefetch-aware (PA) scheduler. The PA policy is based on the TL scheduler (shown in Figures 4.2 (C and C')), with a key difference in the way the fetch groups are formed: the PA policy groups *non-consecutive* warps in the same group. Figure 4.2 (G') shows that warps W1, W3, W5, W7 are in fetch group 1 and warps W2, W4, W6, W8 are in fetch group

2. Similar to the TL policy, since a group is prioritized until it stalls, this policy enables the overlap of memory access latency in one fetch group with computation in another, thereby improving performance over the baseline RR policy. Different from the TL policy, the PA policy exploits bank-level parallelism but may not fully exploit row buffer locality. This is because non-consecutive warps, which do not have spatial locality, generate their memory requests roughly at the same time, and exactly because these requests do not have spatial locality, they are likely to access different banks. In this example, D1 and D3, which access the same row in Bank 1, are issued concurrently with D5 and D7, which access Bank 2, enabling both banks to be busy with requests most of the time when there are outstanding requests. However, D2 and D4, which access the same row D1 and D3 access, are issued much later, which can degrade row buffer locality if the row is closed since the time D1 and D3 accessed it. A detailed description of the PA scheduler will be provided in Section 4.3.1.

4.2.2.2 Prefetch-aware (PA) warp scheduling and inter-fetch-group prefetching

Figures 4.2 (H) and (H') show the execution timeline and DRAM state of the eight warps using the proposed prefetch-aware (PA) scheduler along with a simple inter-fetch-group prefetcher. The simple prefetcher is a simple next-line prefetcher, as assumed in earlier sections. Adding this prefetching mechanism on top of PA scheduling *improves* performance compared to TL scheduling, and in fact achieves the same performance as TL scheduling combined with a *sophisticated, and likely difficult-to-design* prefetcher, since the warp that is being prefetched for gets scheduled long after the warp that prefetches for it. Concretely, consecutive warps (e.g., W1 and W2) that have high spatial locality are located in different fetch groups. When the preceding warp (W1) generates its prefetch (P2 to address X+1), the succeeding warp (W2) does not get scheduled immediately afterwards but after the previous fetch group completes. As a result, there is some time distance between when the prefetch is generated and when it is needed, leading to the prefetch covering the memory access latency partially or fully, resulting in a reduction in *MemoryBlockCycles* and

execution time. Figure 4.2 (H') also shows that using the PA policy in conjunction with a simple prefetcher fully exploits both row buffer locality and bank-level parallelism.

Conclusion: This illustration concludes that the prefetch-aware warp scheduling policy can enable a simple prefetcher to provide significant performance improvements by ensuring that consecutive warps, which are likely to accurately prefetch for each other with a simple prefetcher, are placed in different fetch groups. The next section delves into the design of the prefetch-aware warp scheduling policy and a simple prefetcher that can take advantage of this policy.

4.3 Mechanism and Implementation

This section describes the mechanism and implementation of the *prefetch-aware* warp scheduler (Section 4.3.1), describe a simple spatial locality detector based prefetcher that can take advantage of it (Section 4.3.2), and provide a hardware overhead evaluation of both techniques (Section 4.3.3).

4.3.1 Prefetch-Aware Scheduling Mechanism

As discussed earlier, the PA scheduler is based on the TL scheduler, but its primary difference is in the way the fetch groups are formed. The main goal of the fetch group formation algorithm is to ensure consecutive warps are *not* in the same group such that they can effectively prefetch for each other by executing far apart in time. A second goal of this algorithm is to improve memory bank-level parallelism by enabling non-consecutive warps, which do not have good spatial locality, to generate their memory requests roughly at the same time and spread them across DRAM banks. However, to enable the better exploitation of row buffer locality within a group, the developed algorithm can assign *some* number of consecutive warps into the same group. Algorithm 2 depicts how group formation is performed in the experiments. This section briefly describes its operation using an example.

Group formation depends on the number of warps available on the core (n_warps), and the number of warps in a fetch group (g_size). The number of fetch groups is equal to $\frac{n_warps}{g_size}$. To understand how fetch groups are formed, consider 32 as the maximum number of warps launched on a core, and the group size to be 8 warps. In this case, 4 fetch groups (n_grp) are formed. Warps are enumerated from 0 to 31, and fetch groups are enumerated from 0 to 3. W0 (warp 0) is always assigned to G0 (group 0). The 8th (as group size is equal to 8) warp (W8) is also assigned to G0. Similarly, W16 and W24 are assigned to G0, in a modular fashion until W31 is reached. Since G0 has only 4 warps, 4 more warps need to be assigned to G0. The modular assignment procedure continues with the first unassigned warp, which is W1 in this case, and places it in G0 along with W9, W17 and W25. Note that, in this example, two consecutive warps belong to the same fetch group, e.g., G0 contains both W0 and W1. The number of consecutive warps in a fetch group, n_cons_warps, is equal to $\lfloor \frac{g_size}{n_grp} \rfloor$. Having placed 8 warps in G0, in order to form G1, the algorithm starts from W2. In a similar manner, first, W2, W10, W18 and W26, and then, W3, W11, W19 and W27 are assigned to G1. The group assignment policy exemplified above can be formulated by $g_num[i] = \lfloor \frac{i \bmod g_size}{n_cons_warps} \rfloor$, where $g_num[i]$ denotes the group number of warp i. If there are no consecutive warps in the same group, the above formula simplifies to $g_num[i] = i \bmod g_size$. Algorithm 2 more formally depicts how the evaluated PA scheduler forms fetch groups.[4] Note that, in the evaluations, the fetch group size of 8 is used.[5] The number of warps on the cores depends on the application and the programming model, and is limited by the core resources.

4.3.2 Spatial Locality Detection Based Prefetching

This work developes a *simple* prefetching algorithm that tries to prefetch for consecutive warps (which belong to different fetch groups in the PA scheduler). The key idea of the algorithm is to first detect the regions of memory that are

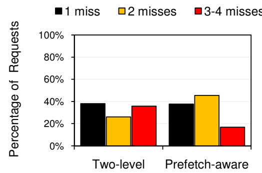
[4]Note that the group formation used in Figure 4.2 is for illustrative purposes only and does not strictly follow this algorithm.

[5]Past works [7,8] that developed scheduling algorithms that form groups showed that a group size of 8 provides the best performance on experimental setups similar to ours.

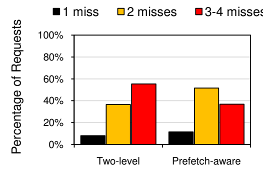
Algorithm 2 Fetch group formation in the PA scheduler

▷ warp and fetch group numbers start from 0
▷ n_warp is the number of concurrently-executing warps on a core
▷ g_size is the number of warps in a fetch group
▷ n_warp is assumed to be divisible by g_size
▷ n_grp is the number of fetch groups
▷ n_cons_warps is the number of consecutive warps in a group. Its minimum value 1.
▷ $g_num[i]$ is the fetch group number of warp i

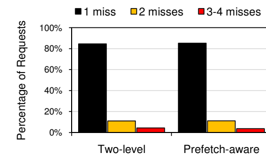
procedure FORM_GROUPS(n_warp, g_size)
  $n_grp \leftarrow \frac{n_warp}{g_size}$
  $n_cons_warps \leftarrow \lfloor \frac{g_size}{n_grp} \rfloor$
  for $i = 0 \rightarrow n_warp - 1$ **do**
    $g_num[i] \leftarrow \lfloor \frac{i \bmod g_size}{n_cons_warps} \rfloor$
  return g_num



(a) Average across all applications



(b) PVC (MapReduce)



(c) BFSR (Rodinia)

Figure 4.3: The distribution of main memory requests, averaged across all fetch groups, that are to macro-blocks that have experienced, respectively, 1, 2, and 3-4 unique cache misses. Section 4.4 describes the methodology and workloads.

frequently accessed (i.e., *hot* regions), and based on this information predict the addresses that are likely to be requested soon.

Spatial locality detection: In order to detect the frequently-accessed memory regions, a spatial locality detector (SLD) similar to the one used in [61] is used. This technique involves the tracking of cache misses that are mapped to the same *macro-block*, which is defined as a group of consecutive cache blocks. In the evaluations, the size of a macro-block is 512 bytes (i.e., 4 cache blocks). SLD maintains a fixed number of macro-block entries in a per-core SLD table (which is organized as a fully-associative 64-entry table in the evaluations). After a main memory request is generated, its macro-block address is searched in the SLD table. The SLD table entry records, using a bit vector, which cache blocks in the macro-block have already been requested. If no matching entry is found, the least-recently-used entry is replaced, a new entry is created, and the corresponding bit in the bit vector is set. If a matching entry is found, simply the

corresponding bit in the bit vector is set. The number of bits that are set in the bit vector indicate the number of unique cache misses to the macro-block.

Prefetching mechanism: The key idea is to issue prefetch requests for the cache lines in the *hot* macro-block that have not yet been demanded. The prefetching mechanism considers a macro-block hot if at least C cache blocks in the macro-block were requested. The value of C is set to 2 in the conducted experiments. For example, if the macro-block size is 4 cache blocks and C is 2, as soon as the number of misses to the macro-block in the SLD table reaches 2, the prefetcher issues prefetch requests for the remaining two cache blocks belonging to that macro-block. Note that this SLD based prefetcher is relatively conservative, as it has a prefetch degree of 2 and a low prefetch distance, because it is optimized to maximize accuracy and minimize memory bandwidth wastage in a large number of applications running on a GPGPU where memory bandwidth is at premium.

Analysis: This work analyzes the effect of warp scheduling, the TL and PA schedulers in particular, on macro-block access patterns. Figure 4.3 shows the distribution of main memory requests, averaged across all fetch groups, that are to macro-blocks that have experienced, respectively, 1, 2, and 3-4 cache misses. Figure 4.3 (a) shows this distribution averaged across all applications, (b) shows it on `PVC`, and (c) shows it on `BFSR`. These two applications are chosen as they show representative access patterns – `PVC` exhibits high spatial locality, `BFSR` does not. Several observations are in order. First, with the TL scheduler, on average across all applications, 36% of memory requests of a fetch group access all cache blocks of a particular macro-block. This means that 36% of the requests from a fetch group have *good* spatial locality. This confirms the claim that the warps in a fetch group have good spatial locality when the TL scheduler is used. However, this percentage goes down to 17% in the PA scheduler. This is intuitive because the PA scheduler favors non-consecutive warps to be in the same fetch group. This results in a reduction in spatial locality between memory requests of a fetch group, but on the flip side, spatial locality can be regained by issuing prefetch requests (as explained in Section 4.2.2.2) to the unrequested cache blocks in the macro-block. Second, 38% of the memory requests from a fetch group access only one cache

Table 4.1: Simulated baseline GPGPU configuration

Parameter	Value
Core Configuration	1300MHz, SIMT width = 8
Resources / Core	Max. 1024 threads (32 warps, 32 threads/warp) 32KB shared memory, 32684 registers
Caches / Core	32KB 8-way L1 data cache, 8KB 4-way texture cache 8KB 4-way constant cache, 128B cache block size
L2 Cache	16-way 128 KB/memory channel, 128B cache block size
Default Warp Scheduling	Round-robin warp scheduling (among ready warps)
Advanced Warp Scheduling	Two-level warp scheduling [8] (fetch group size = 8 warps)
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Interconnect	1 crossbar/direction (30 cores, 8 MCs), concentration = 3, 650MHz
Memory Model	8 GDDR3 Memory Controllers (MC), FR-FCFS scheduling, 8 DRAM-banks/MC, 2KB row size, 1107 MHz memory clock
GDDR3 Timing [64]	$t_{CL} = 10$, $t_{RP} = 10$, $t_{RC} = 35$, $t_{RAS} = 25$ $t_{RCD} = 12$, $t_{RRD} = 8$, $t_{CDLR} = 6$, $t_{WR} = 11$

block of a particular macro-block with the TL scheduler. In `BFSR`, the percentage of such requests goes up to 85%. If an application has a high percentage of such requests, macro-block prefetching is not useful, because these request patterns do not exhibit high spatial locality.

4.3.3 Hardware Overhead

Thjs work evaluates the hardware overhead of the PA scheduler and the spatial locality based prefetcher. Thjs work implemented the two mechanisms in RTL using Verilog HDL and synthesized them using the Synopsys Design Compiler on 65nm TSMC libraries [65].

Scheduling: Lindholm et al. [25] suggest that the warp scheduler used in NVIDIA GPUs has zero-cycle overhead, and warps can be scheduled according to their pre-determined priorities. Since the difference between PA and TL schedulers is primarily in the fetch group formation approach, the hardware overhead of the proposal is similar to that of the TL scheduler. The implementation of the PA scheduler requires assignment of appropriate scheduling priorities (discussed in Section 4.3.1) to all warps on a core. Thjs work synthesized the RTL design of the PA scheduler and found that it occupies 814 μm^2 on each core.

Prefetching: This work implemented the spatial locality detection based prefetcher as described in Section 4.3.2. This work modeled a 64-entry SLD table

and 4 cache blocks per macro-block. This design requires 0.041 mm^2 area per core. Note that the prefetcher is not on the critical path of execution.

Overall Hardware Overhead: For a 30-core system, the required hardware occupies 1.25 mm^2 chip area. This overhead, calculated using a 65nm design, corresponds to 0.27% of the area of the Nvidia GTX 285, which is also a 30-core system, yet is implemented in a smaller, 55nm process technology.

4.4 Evaluation Methodology

The proposed schemes are evaluated using an extensively modified GPGPU-Sim 2.1.2b [31], a cycle-accurate GPGPU simulator. Table 4.1 provides the details of the simulated platform. This work studies 10 CUDA applications derived from representative application suites (shown in Table 4.2). These applications get significant performance improvement over the baseline, when presented with perfect L1 cache, as shown in Figure 4.1. These applications are executed until they complete their execution or reach 1B instructions, whichever comes first.

In addition to using *instructions per cycle (IPC)* as the primary performance metric for evaluation, this work also considers auxiliary metrics such as bank-level parallelism (BLP) and row buffer locality (RBL). BLP is defined as the average number of memory banks that are accessed when there is at least one outstanding memory request at any of the banks [37, 45–47]. Improving BLP enables better utilization of DRAM bandwidth. RBL is defined as the average hit-rate of the row buffer across all memory banks [45]. Improving RBL increases the memory service rate and also enables better DRAM bandwidth utilization. This work also measures the L1 data cache miss rates when prefetching is employed. Accurate and timely prefetches can lead to a reduction in miss rates.

Table 4.2: Evaluated GPGPU applications

#	Suite	Application	Abbr.
1	MapReduce [7, 44]	SimilarityScore	SSC
2	MapReduce [7, 44]	PageViewCount	PVC
3	Rodinia [42]	Kmeans Clustering	KMN
4	Parboil [43]	Sparse Matrix Multiplication	SPMV
5	Rodinia [42]	Breadth First Search	BFSR
6	Parboil [43]	Fast Fourier Transform	FFT
7	CUDA SDK [23]	Scalar Product	SCP
8	CUDA SDK [23]	Blackscholes	BLK
9	CUDA SDK [23]	Fast Walsh Transform	FWT
10	Third Party	JPEG Decoding	JPEG

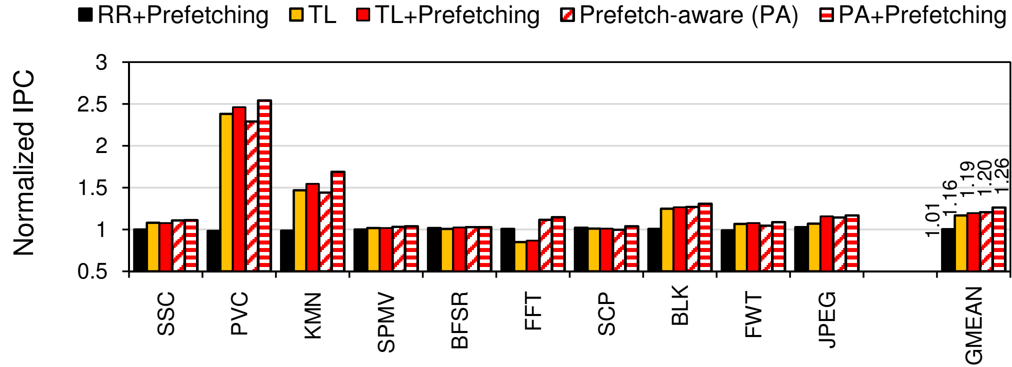
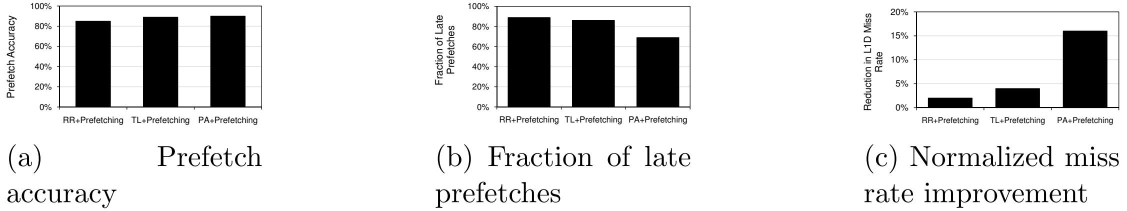


Figure 4.4: IPC performance impact of different scheduling and prefetching strategies. Results are normalized to the IPC with the RR scheduler.

4.5 Experimental Results

Figure 4.4 shows the IPC improvement of five different combinations of warp scheduling and prefetching normalized to the baseline RR scheduler: 1) RR scheduler with data prefetching, 2) TL scheduler, 3) TL scheduler with spatial locality detection based prefetching, 4) PA scheduler, 5) PA scheduler with prefetching. Overall, without prefetching, the PA scheduler provides 20% average IPC improvement over the RR scheduler and 4% over the TL scheduler. When prefetching is employed, the PA scheduler provides 25% improvement over the RR scheduler and 7% over the TL scheduler, leading to performance within 1.74× of a perfect L1 cache. The rest of this section analyzes the different combinations of scheduling and prefetching.

Effect of prefetching with the RR scheduler: On average, adding the SLD-based prefetcher over the RR scheduler provides only 1% IPC improvement



(a) Prefetch accuracy

(b) Fraction of late prefetches

(c) Normalized miss rate improvement

Figure 4.5: (a) Prefetch accuracy, (b) Fraction of late prefetches, and (c) Reduction in L1 data cache miss rate when prefetching is implemented with each scheduler. The results are averaged across all applications.

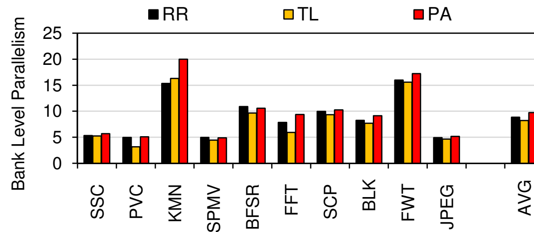


Figure 4.6: Effect of various scheduling strategies on DRAM bank-level parallelism (BLP)

over the RR scheduler without prefetching. The primary reason is that the prefetcher cannot lead to timely transfer of data as described in Section 4.2.1.2. Figure 4.5 shows that even though 85% of the issued prefetches are accurate with the RR scheduler, 89% of these accurate prefetches are late (i.e., are needed before they are complete), and as a result the prefetcher leads to an L1 data cache miss reduction of only 2%.

Effect of the TL scheduler: As discussed in Section 4.2.1.3, the TL scheduler improves latency tolerance by overlapping memory stall times of some fetch groups with computation in other fetch groups. This leads to a 16% IPC improvement over the RR scheduler due to better core utilization. One of the primary limitations of TL scheduling is its inability to maximize DRAM bank-level parallelism (BLP). Figure 4.6 shows the impact of various scheduling strategies on BLP. The TL scheduler leads to an average 8% loss in BLP over the RR scheduler. In `FFT`, this percentage goes up to 25%, leading to a 15% performance loss over the RR scheduler.

Effect of prefetching with the TL scheduler: Figure 4.7 shows the L1 data cache miss rates when prefetching is incorporated on top of the TL

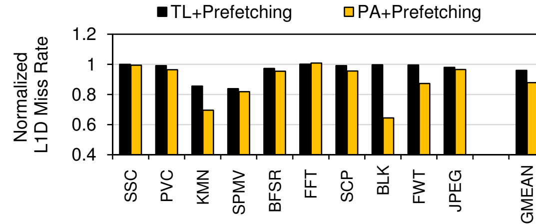


Figure 4.7: Effect of various scheduling and prefetching strategies on L1D miss rate. Results are normalized to miss rates with the TL scheduler.

scheduler. Using the spatial locality detection based prefetcher decreases the L1 miss rate by 4% over the TL scheduler without prefetching. In `KMN` and `JPEG`, this reduction is 15% and 2%, respectively. This L1 miss rate reduction leads to a 3% performance improvement over the TL scheduler. Note that the performance improvement provided by prefetching on top of the TL scheduler is relatively low because the issued prefetches are most of the time too late, as described in Section 4.2.1.4. However, the performance improvement of prefetching on top of the TL scheduler is still higher than that on top of the RR scheduler because the proposed prefetcher is not *too* simple (e.g., not as simple as a next-line prefetcher) and its combination with the TL scheduler enables some inter-fetch-group prefetching to happen. Figure 4.5 shows that 89% of the issued prefetches are accurate with the TL scheduler. 85% of these accurate prefetches are still late, and as a result the prefetcher leads to an L1 data cache miss reduction of only 4%.

Effect of the PA scheduler: As described in Section 4.2.2.1, the PA scheduler is likely to provide high bank-level parallelism at the expense of row buffer locality because it concurrently executes non-consecutive warps, which are likely to access different memory banks, in the same fetch group. Figures 4.6 and 4.8 confirm this hypothesis: on average, the PA scheduler improves BLP by 18% over the TL scheduler while it degrades RBL by 24%. The PA scheduler is expected to work well in applications where the loss in row locality is less important for performance than the gain in bank parallelism. For example, the PA scheduler provides a 31% increase in IPC (see Figure 4.4) over the TL scheduler in `FFT` due to a 57% increase in BLP, even though there is a 44% decrease in row locality compared to the TL scheduler. On the contrary, in `PVC`, there is 4% reduction in IPC compared to

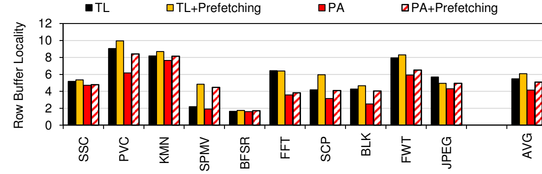


Figure 4.8: Effect of different scheduling and prefetching strategies on DRAM row buffer locality

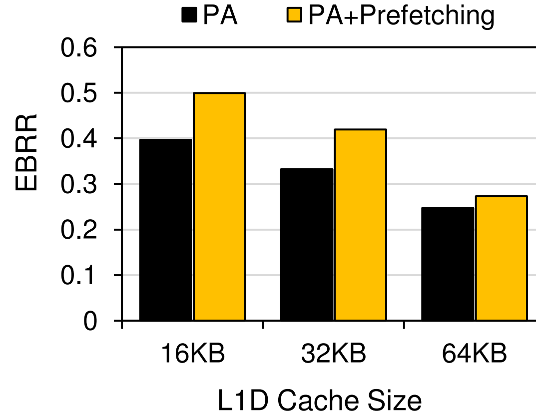


Figure 4.9: Effect of prefetching on Evicted Block Reference Rate (EBRR) for various L1 data cache sizes

TL scheduler due to a 31% reduction in row-locality, even though there is a 62% increase in BLP. This shows that both row locality and bank-level parallelism are important for GPGPU applications, which is in line with the observations made in [53] and [7]. On average, the PA scheduler provides 20% IPC improvement over the RR scheduler and 4% improvement over the TL scheduler.

Effect of prefetching with the PA scheduler: The use of prefetching together with the proposed PA scheduler provides two complementary benefits: (1) the PA scheduler enables prefetching to become more effective by ensuring that consecutive warps that can effectively prefetch for each other get scheduled far apart in time from each other, (2) prefetching restores the row buffer locality loss due to the use of the PA scheduler. This dissertation work evaluates both sources of benefits in Figures 4.7 and 4.8, respectively: (1) the use of prefetching together with the PA scheduler reduces the L1 miss rate by 10% compared to the TL scheduler with prefetching, (2) the addition of prefetching over the PA scheduler improves RBL such that the RBL of the resulting system is within 16% of the RBL

of the TL scheduler with prefetching. Figure 4.5 shows that the PA scheduler significantly improves prefetch timeliness compared to the TL and RR schedulers while also slightly increasing prefetch accuracy: with the PA scheduler, 90% of the prefetches are accurate, of which only 69% are late (as opposed to the 85% late prefetch fraction with the TL prefetcher). Overall, the orchestrated spatial prefetching and prefetch-aware scheduling mechanism improves performance by 25% over the RR scheduler with prefetching and by 7% over the TL scheduler with prefetching (as seen in Figure 4.4). This work concludes that the new prefetch-aware warp scheduler effectively enables latency tolerance benefits from a simple spatial prefetcher.

Case analyses of workload behavior: In `KMN`, prefetching with the PA scheduler provides 10% IPC improvement over prefetching with the TL scheduler on account of a 31% decrease in L1 miss rate and 6% increase in row locality over the PA scheduler. Note that in `KMN`, prefetching on top of the PA scheduler leads to the restoration of 100% of the row-locality lost by the PA scheduler over the TL scheduler. In `SPMV`, the use of prefetching improves row buffer locality by 2× over the TL scheduler. This enhancement, along with a 9% increase in BLP over the TL scheduler leads to 3% higher IPC over TL+Prefetching. In `BFSR`, this work does not observe any significant change in performance with the proposal because of the high number of unrelated memory requests that do not have significant spatial locality: in this application, many macro-blocks have only one cache block accessed, as was shown in Figure 4.3 (c). In `FFT`, adding prefetching on top of the PA scheduler provides 3% additional performance benefit. Yet, as described earlier, `FFT` benefits most from the improved BLP provided by the PA scheduler. In `FFT`, the combination of prefetching and PA scheduling actually *increases* the L1 miss rate by 1% compared to the combination of prefetching and TL scheduling (due to higher cache pollution), yet the former combination has 3% higher performance than the latter as it has much higher bank-level parallelism. This shows that improving memory bank-level parallelism, and thereby reducing the *cost* of each cache miss by increasing the overlap between misses can actually be more important than reducing the L1 cache miss rate, as was also observed previously for CPU workloads [66].

Analysis of cache pollution due to prefetching: One of the drawbacks of prefetching is the potential increase in cache pollution, which triggers early evictions of cache blocks that are going to be needed later. The cache pollution is calculated using the Evicted Block Reference Rate (EBRR) Equation 4.1. This metric indicates the fraction of read misses that are to cache blocks that were in the cache but were evicted due to a conflict.

$$EBRR = \frac{\#read\ misses\ to\ already\ evicted\ cache\ blocks}{\#read\ misses} \tag{4.1}$$

Figure 4.9 shows that prefetching causes a 26% increase in EBRR when a 32KB L1 data cache (as in the evaluations) is used. When cache size is increased to 64KB, the increase in EBRR goes down to 10%. This is intuitive as a large cache will have fewer conflict misses. One can thus reduce pollution by increasing the size of L1 caches (or by incorporating prefetch buffers), but that would lead to reduced hardware resources dedicated for computation, thereby hampering thread-level parallelism and the ability of the architecture to hide memory latency via thread-level parallelism. Cache pollution/contention can also be reduced via more intelligent warp scheduling techniques, as was shown in prior work [7,67]. This work leaves the development of warp scheduling mechanisms that can reduce pollution in the presence of prefetching as a part of future work.

4.6 Related Work

This section briefly describes and compares to the closely related works.

Scheduling techniques in GPUs: The two-level warp scheduler proposed by Narasiman et al. [8] splits the concurrently executing warps into groups to improve memory latency tolerance. This work has already provided extensive qualitative and quantitative comparisons of this proposal to the two-level scheduler. Rogers et al. [67] propose cache-conscious wavefront scheduling to improve the performance of cache-sensitive GPGPU applications. Gebhart et al. [48] propose a two-level warp scheduling technique that aims to reduce energy consumption in GPUs. Kayiran et al. [9] propose a CTA scheduling mechanism that dynamically estimates the

amount of thread-level parallelism to improve GPGPU performance by reducing cache and DRAM contention. Jog et al. [7] propose OWL, a series of CTA-aware scheduling techniques to reduce cache contention and improve DRAM performance for various GPGPU applications. None of these works consider the effects of warp scheduling on data prefetching. This dissertation work examines the interaction of scheduling and prefetching, and develops a new technique to orchestrate these two methods of latency tolerance.

Data prefetching: Lee et al. [35] propose a many-thread aware prefetching strategy in the context of GPGPUs. The prefetch-aware warp scheduling technique can be synergistically combined with this prefetcher for better performance. Jog et al. [7] propose a memory-side prefetching technique that improves L2 cache hit rates in GPGPU applications. This dissertation work describes how a spatial locality detection mechanism can be used to perform core-side data prefetching. Lee et al. [68] evaluate the benefits and limitations of both software and hardware prefetching mechanisms for emerging high-end processor systems. Many other prefetching and prefetch control mechanisms (e.g., [55–57, 69, 70]) have been proposed within the context of CPU systems. The proposed prefetch-aware scheduler is complementary to these techniques. This work also provides a specific core-side prefetching mechanism for GPGPUs that is based on spatial locality detection [61].

Row buffer locality and bank-level parallelism: Several memory request scheduling techniques for improving bank-level parallelism and row buffer locality [34, 37, 45, 46, 49–51, 71–73] have been proposed. In particular, the work by Hassan et al. [52] quantifies the trade-off between BLP and row locality for multi-core systems, and concludes that bank-level parallelism is more important. The results show that the prefetch-aware warp scheduler, which favors bank-level parallelism, provides higher average performance than the two-level scheduler [8], which favors row buffer locality (but this effect could be due to the characteristics of the evaluated applications.). On the other hand, Jeong et al. [53] observe that both bank-level parallelism and row buffer locality are important in multi-core systems. This work also finds that improving row locality at the expense of bank parallelism improves performance in some applications yet

reduces performance in others, as evidenced by the two-level scheduler outperforming the prefetch-aware scheduler in some benchmarks and vice versa in others. Lakshminarayana et al. [50] propose a DRAM scheduling policy that essentially chooses between Shortest Job First and FR-FCFS [33, 34] scheduling policies at run-time, based on the number of requests from each thread and their potential of generating a row buffer hit. Yuan et al. [49] propose an arbitration mechanism in the interconnection network to restore the lost row buffer locality caused by the interleaving of requests in the network when an in-order DRAM request scheduler is used. Ausavarungnirun et al. [51] propose a staged memory scheduler that batches memory requests going to the same row to improve row locality while also employing a simple in-order request scheduler at the DRAM banks. Lee et al. [71, 72] explore the effects of prefetching on row buffer locality and bank-level parallelism in a CPU system, and develop memory request scheduling [71, 72] and memory buffer management [71] techniques to improve both RBL and BLP in the presence of prefetching. Mutlu and Moscibroda [37, 73] develop mechanisms that preserve and improve bank-level parallelism of threads in the presence of inter-thread interference in a multi-core system. None of these works propose a *warp scheduling* technique that exploits bank-level parallelism, which this dissertation work does. This work explores the interplay between row locality and bank parallelism in GPGPUs, especially in the presence of prefetching, and aim to achieve high levels of both by intelligently orchestrating warp scheduling and prefetching.

4.7 Chapter Summary

This chapter shows that state-of-the-art thread scheduling techniques in GPGPUs cannot effectively integrate data prefetching. The main reason is that *consecutive thread warps*, which are likely to generate accurate prefetches for each other as they have good spatial locality, are scheduled closeby in time with each other. This gives the prefetcher little time to hide the memory access latency before the address prefetched by one warp is requested by another warp.

To orchestrate thread scheduling and prefetching decisions, this chapter

introduces a *prefetch-aware (PA) warp scheduling* technique. The main idea is to form groups of thread warps such that those that have good spatial locality are in separate groups. Since warps in different thread groups are scheduled at separate times, *not* immediately after each other, this scheduling policy enables the prefetcher to have more time to hide the memory latency. This scheduling policy also better exploits memory bank-level parallelism, even when employed without prefetching, as threads in the same group are more likely to spread their memory requests across memory banks.

Experimental evaluations show that the proposed prefetch-aware warp scheduling policy improves performance compared to two state-of-the-art scheduling policies, when employed with or without a hardware prefetcher that is based on spatial locality detection. This chapter concludes that orchestrating thread scheduling and data prefetching decisions in a GPGPU architecture via prefetch-aware warp scheduling can provide a promising way to improve memory latency tolerance in GPGPU architectures.

Chapter 5

Criticality Aware Memory Scheduling Techniques

Modern memory access schedulers employed in GPUs typically optimize for throughput and implicitly assume that all requests from different cores are equally important. However, different cores have different amounts of tolerance to latency during execution. In particular, cores with a larger fraction of warps waiting for data to come back from DRAM are less likely to tolerate the latency of an outstanding memory request. Requests from such cores are more critical than requests from others. Based on this observation, new memory scheduler is developed, called *(C)ritica(L)ity (A)ware (M)emory (S)cheduler* (CLAMS), which takes into account the latency-tolerance of the cores that generate memory requests. The key idea is to use the percentage of critical requests in the memory request buffer to switch between scheduling policies optimized for criticality and locality. If the percentage is below a threshold, CLAMS prioritizes critical requests to ensure cores that cannot tolerate latency are serviced faster. Otherwise, CLAMS optimizes for locality anticipating that there are too many critical requests and prioritizing one over another would not significantly benefit performance. A core-criticality estimation mechanism is first developed for determining critical cores and requests, and then various issues are discussed related to finding a balance between criticality and locality in the memory

scheduler. The results indicate that a GPU memory system that considers both core-criticality and DRAM request locality can provide significant improvement in performance.

5.1 Introduction

Graphics Processing Units (GPUs) are becoming increasingly popular for general purpose computing because of their capability in providing large improvements in performance and energy efficiency compared to CPUs [74–83]. Although high-bandwidth memory systems have increased GPU performance substantially, memory bandwidth is still precious and a critical performance determinant [7, 9, 84–86]. In fact, it will be more so as compute resources increase on GPUs [7, 9, 28, 87]. To address this, a large body of work has focused on improving bandwidth utilization (e.g., [7, 26, 27, 51]), and caching efficiency in GPUs (e.g., [7, 36, 67]). However, all these works primarily focus on improving application performance by treating all threads and memory requests with *equal* importance. This phenomenon stems from the fact that GPUs typically focus on improving the collective performance of multiple concurrently executing threads. In the same vein, the commonly-used memory scheduling policy – First-ready FCFS (FR-FCFS) [32–34], implicitly assumes that all memory requests are *equally critical* for overall performance, and hence, it optimizes for throughput rather than for latency of specific requests. However, this work observes that because of the contention of memory requests from different cores in the memory system, and the inability of the FR-FCFS memory scheduler to distinguish between the memory requests originating from different cores, they experience significant variation in average memory access latencies. Due to such variation, the number of stalling warps that belong to the cores that suffer from higher memory access latencies is typically higher than that of other cores, making the former type of cores less latency tolerant, i.e., more *critical* for overall performance. This implies that because different cores have varying degrees of tolerance to latency during execution, their corresponding memory requests have varying degrees of criticality.

In contrast to the purely locality-focused memory schedulers, the goal in this work is to design a memory scheduler that is cognizant of the latency tolerance of cores. One simple idea based on the observation is to *always* prioritize critical requests over non-critical requests. As the cores that lack enough warps to hide the long memory latencies are more likely to quickly stall for the data to come back, prioritizing requests from such cores in the memory controller provides a way of proactively avoiding them from getting stalled. However, this work finds that such a memory scheduler that is focused *purely* on core criticality degrades DRAM access locality significantly. This motivates us to explore more intelligent memory scheduling schemes that consider *both criticality and locality.* In this context, this chapter introduces a *(C)ritica(L)ity (A)ware (M)emory (S)cheduler* (CLAMS) for GPUs, which achieves a fine balance between core criticality and DRAM access locality.

There are four steps in designing CLAMS. First, this work periodically calculates the current level of latency tolerance of a GPU core. This work does so by periodically calculating the fraction of short-latency warps on the core.[1] A core is expected to be more latency tolerant if most of the launched warps are short-latency warps that execute compute (ALU) instructions or that find their required data in privates caches. Second, this work periodically ranks the cores based on their current level of latency tolerance, and tag the memory requests with the core's rank. The ranking is done in such a way that the cores that have lower latency tolerance are ranked lower. Third, based on the value of the rank, this work determines if a request should be considered critical or not. To do so, this work uses a criticality-rank threshold (Th_{CR}), which specifies *up to which rank* a request should be considered as *critical.* Fourth, this work decides if a critical request should be prioritized or not by the memory scheduler by also taking DRAM access (row-buffer) locality into account. This work does so by periodically calculating the percentage of requests that are considered as critical in the memory request buffer, and comparing it with the scheduling-mode threshold (Th_{SM}). If the percentage of critical requests is below Th_{SM}, CLAMS goes into criticality mode, where it prioritizes critical requests to ensure cores

[1] Section 5.3 details the procedure to measure the latency tolerance of a core.

that cannot tolerate latency are serviced faster. Otherwise, CLAMS goes into locality mode, where it optimizes for locality anticipating that there are too many critical requests to prioritize one over another.

This work observes latency tolerance differences between GPU cores and exploits differences in such tolerance to improve GPU resource management. In this context, this chapter makes the following **contributions**,

• This work introduces the concept of core-criticality in GPUs. This work shows that all GPU cores do *not* possess the same latency tolerance at all times, and this variation in latency tolerance across cores is one of the key reasons for different levels of criticality among memory requests, which is not exploited by current GPU memory schedulers.

• This work introduces the first GPU memory scheduler, CLAMS to take into account core-criticality and achieves a fine balance between criticality and locality via proposed dynamic criticality estimation mechanism. This work proposes three different designs for CLAMS: static, semi-dynamic (Semi-Dyn) and dynamic (Dyn) based on how required thresholds (Th_{CR} and Th_{SM}) are computed, and find that the Dyn-CLAMS is the best performer because of its ability to compute these thresholds at runtime and thereby adapt dynamically to application demands.

• This work presents a comprehensive experimental evaluation of three CLAMS designs along with FR-FCFS and FR-FCFS-Cap using a variety of CUDA workloads. The results show that Dyn-CLAMS reduces latency of critical memory requests by 35%, resulting in an average 9% IPC improvement (maximum 15%) over the FR-FCFS scheduler, and is within 1% of the scheduler that uses best threshold values profiled separately for *each* application. Further, performance of none of the evaluated applications degrades with the final scheme.

5.2 Background and Motivation

A typical GPU consists of multiple simple cores, also called streaming-multiprocessors[2] [4] in NVIDIA terminology. Each core is associated with private L1 data, texture and constant caches, along with software-managed scratchpad memory. The cores are connected to memory channels (partitions) via an interconnection network. Each memory partition is associated with a shared L2 cache, and its associated memory requests are handled by GDDR5 memory controller. When an application kernel is launched on a GPU, the memory requests originating from different cores interfere at various levels of the GPU hierarchy, such as interconnect, last-level caches and main-memory. At each of these levels, the underlying shared resource management policies do not consider the source core-ids of the requests while making decisions, and therefore might allocate shared resources across different cores in an *uneven* fashion. The detailed analysis shows that such uneven allocation can lead to significant variation in average memory latencies across different GPU cores. To understand this variation, *coefficient of variation (COV)* in memory access latencies is calculated, which is defined as the ratio of standard deviation to arithmetic mean of the average memory access latencies experienced by different cores.

Figure 5.1 shows the COV in memory access latencies as well as IPCs across different cores for 11 different CUDA applications, averaged across fixed epochs[3], for three different scenarios: applications when executed on a GPU that has 1) equal to (1xB), 2) double (2xB), and 3) quadruple (4xB) the peak memory bandwidth of the baseline GPU architecture. In the baseline scenario (1xB), significant COV in latencies is observed across cores for applications like `LUH` (13%), `RAY` (33%), `SCAN` (19%), and `RED` (18%). Because of such variation, many GPU cores experience higher memory access latencies than the others. Therefore, such cores have high number of stalling warps on memory, making them *less latency tolerant* than other cores. This might also result in higher stall times for these cores, leading to also significant COV in IPCs across different cores as

[2]This chapter uses the term *core* for streaming-multiprocessor (SM).

[3]Epoch length is 10K cycles.

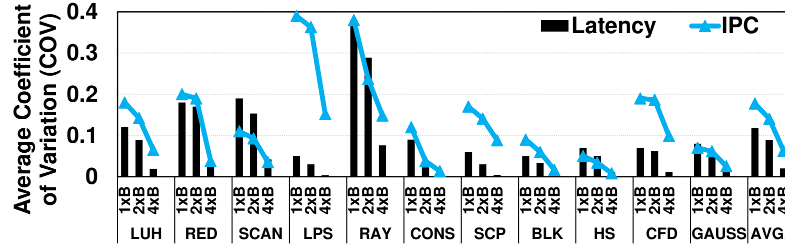


Figure 5.1: Average Coefficient of Variation (COV) in average memory access latencies and IPCs across different GPU cores.

shown in Figure 5.1.

Significant COV both in latencies and IPCs is observed for `RAY`, `LUH`, and `RED`, stressing the fact that variation in latencies can lead to significant variation in IPCs across cores. In addition to this observation, two contrasting cases are noticed. First, in `SCAN`, variation in latencies across cores is higher than variation in IPCs. This is because `SCAN` is able to tolerate higher latencies up to certain level, thereby reducing the variation in IPCs. Second, in `LPS` and `CFD`, much lower variation in latencies across cores is present compared to variation in IPCs. This is because CTA and instruction-mix load imbalance [9,31,88,89] across cores also causes IPC variation during application execution. It is further observed that increasing peak memory bandwidth causes significant decrease in COV in latencies as well as IPCs, leading to the conclusion that contention of different cores in the memory system is the major cause of COV in latencies as well as IPCs.

The goal is to develop a mechanism that prioritizes the cores that suffer from lower latency tolerance. Improving the performance of these cores would improve overall system performance by enabling these cores to make progress (Section 5.3.1). Such mechanism is expected to specifically benefit those applications that have significant COV both in latencies and IPCs (e.g., `RAY`, `LUH`, and `RED`), and not so for those applications (e.g., `BLK`, `HS`) that have limited variation. Such a mechanism can be employed at various levels. For example, a warp scheduler can be employed to control the progress of each core separately. However, since the cores contend for the memory system resources, a memory scheduling mechanism can be more effective to expedite the requests of cores with lower latency tolerance, and is the focus of this chapter. Thus, in this work,

a new GPU memory scheduler is designed that is aware of the latency tolerance of individual cores.

5.3 Core Criticality: Basic Ideas and Metrics

This section provides details on the metrics to gauge the latency tolerance of a core and its variance across cores.

5.3.1 Latency Tolerance as a Measure of Core-Criticality

Each GPU core is capable of executing multiple warps concurrently. The advantage of this is that if some of the warps are blocked, for example, because of pending DRAM accesses, the remaining warps can continue their execution and potentially mask the performance penalties of the blocked warps. However, the number of warps is not always enough to keep the core busy with useful work. There are various reasons for this. For example, some warps cannot be issued to the SIMT pipelines at a given time, because of data and control dependencies, or pipeline register stalls. This implies that only a fraction of the total warps launched on a core can actually be used to provide latency tolerance at a given time.

In order to define a metric to gauge the criticality of a core, this work employs a two-step strategy. The first step is to classify the *issued* warps as *long-latency* and *short-latency* warps. The number of issued warps is equal to the number of unique warps that reserve a register in the scoreboard. The long-latency warps are the issued warps that have at-least one pending L1 miss. The remaining issued warps are called short-latency warps because either they contain ALU instructions or all the data required by them is present in the L1 caches. These short-latency warps finish their instructions faster compared to the long-latency warps, whose executions are dependent on data that needs to be fetched from the memory-partition/DRAM. As observed, the warp classification mechanism is purely based on warps' memory behavior and their thread-level parallelism.

After classifying the warps, the second step is to periodically[4] calculate *the ratio of short-latency warps to the total issued warps.* Note that the sum of short-latency and long-latency warps is always equal to the total issued warps, and thus this ratio can take any value between 0 and 1. *This work uses this ratio as a metric to gauge the latency tolerance of a core.* Note that this work calculates this metric separately for each core. The lower the value of this ratio, the lower is the latency tolerance of the core. The intuition is that, if the ratio is high, the core has high percentage of short-latency warps, and the latencies of long-latency warps can potentially be completely masked, leading to high core performance. On the other hand, if the ratio is low, even if the core has short-latency warps that are ready to execute, the majority of the issued warps are long-latency warps, and the core is more likely to stall soon.

The observation is that relative change in IPC and relative change in the latency tolerance metric has an average correlation of 74% across the application suite. This means that improving the latency tolerance of a core might improve its IPC. However, a mechanism that prefers memory requests of cores that have lower latency tolerance might have higher impact on the overall performance. For example, it is more advantageous to make one more additional warp ready to execute in a core with zero short-latency warps (i.e., no latency tolerance) compared to a core with a large number of short-latency warps (there is already enough latency tolerance).

This work quantizes the latency tolerance metric into eight equal parts.[5] Based on this value, this work assigns a criticality *rank* to a core. Essentially, each equal part of the ratio corresponds to a rank. For example, if this ratio is less than or equal to $\frac{1}{8}$, the core is considered to be the most critical and is in *rank-1* state. Similarly, if it is greater than $\frac{7}{8}$, that core is considered to be the least critical, and is in *rank-8* state.

Formally, this work considers a core to be critical if the current *rank* of the

[4]This work calculates this ratio over an epoch of 32 cycles.

[5]The proposed scheme could have divided this ratio into more than eight parts to get a finer granularity picture of the current latency tolerance of a core, but the detailed studies show that eight parts provide sufficient granularity to understand and distinguish the latency tolerance of different cores.

core is less than or equal to a *Criticality-Rank-Threshold* (Th_{CR}). In other words, the value of Th_{CR} specifies ***up to which rank*** the core should be considered as *critical*. For example, a Th_{CR} value of 4 implies that the core is considered critical only if its current rank is less than or equal to rank 4.

5.3.2 Understanding Variation of Criticality Across Cores

Not only the latency tolerance of a core can change during execution, but it is observed that there is a wide variation of latency tolerance across GPU cores. To measure this variation, a new metric is introduced, called *percentage of critical cores (PCC)*, which is defined as the percentage of GPU cores that are in the critical state. Since, a core is treated as critical based on the chosen value of Criticality-Rank-Threshold (Th_{CR}), PCC needs to be defined for a particular value of Th_{CR}. Hence, PCC(Th_{CR}) is defined as the percentage of critical cores (PCC) for a particular Th_{CR}, where Th_{CR} can take any integer from 1 to 8. If PCC(Th_{CR}) is equal to 100%, it means that all the cores are critical, and have similar tolerance to latencies. Similarly, if PCC(Th_{CR}) is equal to 0%, it means that all cores are non-critical. In both the cases, the variation in criticality across cores is insignificant. On the other hand, if the value of PCC(Th_{CR}) is in mid-range, then some cores are treated as critical and the remaining cores are not. Therefore, the value of PCC(Th_{CR}) gives a notion of the variation in criticality across cores. For a better understanding of the PCC metric, consider Figure 5.2 that hypothetically shows the current rank of cores in a 3-core GPU system.

It is noticed that if Th_{CR} is chosen as 1, PCC is equal to 0%, as all the cores have higher rank than 1 and none of the cores are considered as critical. With Th_{CR} equal to 4, the value of PCC is equal to 33%, as the rank of core-3 is less than the chosen value of Th_{CR}, making it the only critical core in the system. With Th_{CR} equal to 7, the value of is PCC is equal to 100%, as all the cores are considered critical because ranks of all the cores are less than the Th_{CR}.

It is observed that as Th_{CR} increases, the number of critical cores also increases (or remains the same). Therefore, PCC(Th_{CR}) also increases (or remains the same)

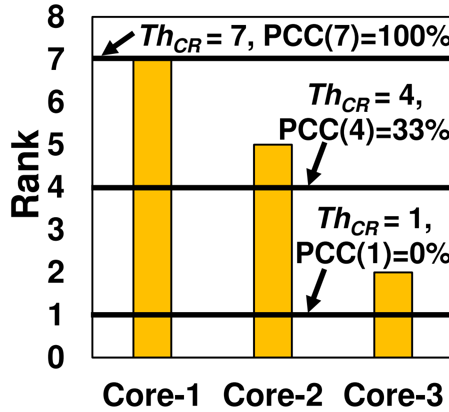


Figure 5.2: Illustrative example to demonstrate PCC.

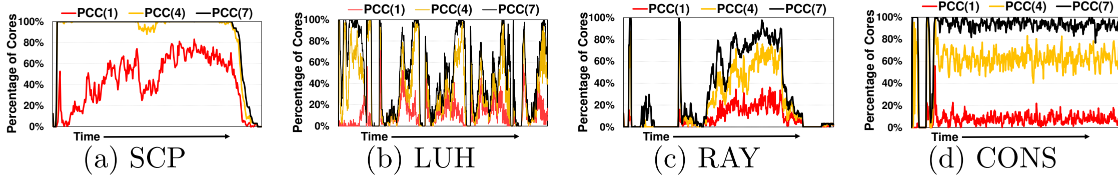
as Th_{CR} increases from 1 to 8. Formally, $[PCC(a) \geq PCC(b)]$, if $(a > b)$.

5.3.2.1 Analysis of the PCC metric.

Figure 5.3 shows PCC(Th_{CR}) over time (sampled at every 1K cycles) for four applications, at three different levels of Th_{CR} ($Th_{CR} = \{1,\ 4,\ 7\}$), over time. Three main observations from this figure are:

Observation-I: PCC is dependent on the chosen criticality-rank threshold. During the application execution, the instantaneous PCC(Th_{CR}) is a function of the chosen Th_{CR} value. For example, in `SCP`, values of PCC($Th_{CR} = 1$) and PCC($Th_{CR} = 7$) at a given time are very different. This is by definition from the previous discussion that, as Th_{CR} increases, the number of critical cores increases, leading to high PCC values. The other three applications (`LUH`, `RAY`, `CONS`) also exhibit this trend.

Observation-II: PCC varies within an application over time. Even for a fixed value of Th_{CR}, PCC(Th_{CR}) may not be constant throughout the execution. For example, as observed prominently in `LUH` and `RAY`, PCC($Th_{CR} = 4$) can be different over time, implying that the number of critical cores for the same value of Th_{CR} is not constant over time. In `CONS`, the change in PCC(Th_{CR}) over time is not very prominent, and PCC($Th_{CR} = 4$) is in the mid-range (40-60%), implying that roughly half the cores are in critical state.



(a) SCP (b) LUH (c) RAY (d) CONS

Figure 5.3: Variation of criticality across cores with different criticality-rank thresholds. This variation is measured using the PCC metric.

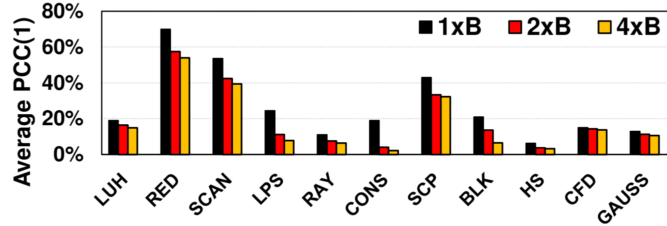


Figure 5.4: Effect of increase in peak memory bandwidth on PCC.

Observation-III: PCC varies across applications. Across applications, even for the same value of Th_{CR}, PCC(Th_{CR}) can be very different. For example, at $Th_{CR} = 4$, `RAY` and `SCP` have very different PCC($Th_{CR} = 4$) values (`SCP`'s is fairly high than `RAY`'s). It implies that, with $Th_{CR} = 4$, higher number of critical cores exist in `SCP` compared to `RAY` application.

Another Observation. Figure 5.4 shows that the PCC metric[6] reduces significantly with increase in memory bandwidth in the system. This trend is consistent with the discussions from Section 5.2 that the variation average memory latencies observed by different cores (i.e., variation in criticality across different cores) is alleviated with increased bandwidth.

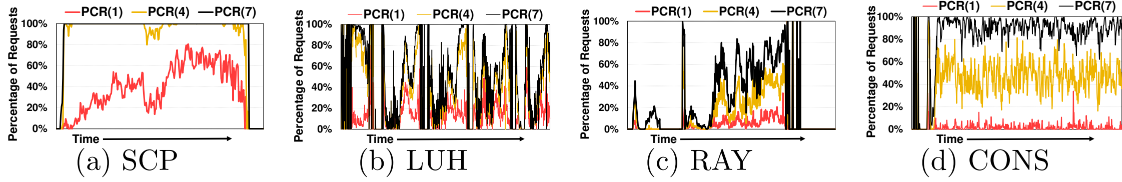
5.4 Analyzing Criticality in the Memory System

Cores with low latency tolerance are less likely to tolerate the latency of an outstanding memory request, making their requests more critical. Thus, the goal is to design a *criticality-aware* memory-scheduler that takes advantage of

[6]Results for PCC with Th_{CR} equals to 1 are shown because the critical cores at this Th_{CR} have the least latency tolerance, and reducing PCC(1) by preferring the memory requests of such cores is advantageous to improve the overall performance (Section 5.3.1).

differences in criticality among requests and prioritizes latency critical requests to improve performance. One of the important steps in designing such a memory scheduler is to gauge the variation in criticality across GPU cores. As discussed in Section 5.3.2, PCC(Th_{CR}) metric is one of the key indicators of existence of different levels of criticality among cores, and in turn their corresponding memory requests. If PCC(Th_{CR}) metric indicates that the latency tolerance variance across cores exists for a particular value of Th_{CR}, the memory scheduler can potentially prioritize requests from cores that have lower ranks. This is because such cores are more likely to have large number of warps stalling due to waiting for memory requests to come back from memory. Therefore, we can prioritize requests from such cores to *proactively* avoid causing these cores to stall. However, note that as PCC(Th_{CR}) is dependent on Th_{CR}, we need to carefully examine the PCC(Th_{CR}) values for all possible values of Th_{CR}, to understand at what level of Th_{CR} does substantial latency criticality variation across cores exists (if any). If PCC(Th_{CR}) metric does not indicate significant variance at any Th_{CR} level, the memory requests from different cores will have similar latency criticality. In such situations, a scheduler can focus on preserving locality as giving some requests higher priority than others may not have any benefit.

The calculation of PCC(Th_{CR}) requires global information exchange across cores, and the hardware overhead of calculating this information and then communicating it directly to the memory controllers (MCs) periodically can be expensive. Instead, this work proposes to capture the variations in latency tolerance across cores directly at the MCs. To do so, the current latency tolerance level of a core is relayed to the GPU memory scheduler by tagging the memory requests originating from that core with its current rank, and then calculating, at the MC, a metric called percentage of critical requests (PCR), which is defined as percentage of critical memory requests present in a MC memory request buffer. Again, because the decision of defining requests (or cores as discussed before in Section 5.3.2) as critical is dependent on the value of Th_{CR}, PCR(Th_{CR}) is defined as the percentage of critical requests (*the requests that are tagged with rank values less than or equal to* Th_{CR}) in MC request buffer. ***Note that the observations discussed for PCC(*** Th_{CR} ***) in Section 5.3.2 hold true for PCR(*** Th_{CR} ***) as well. It is because the only***



(a) SCP (b) LUH (c) RAY (d) CONS

Figure 5.5: Variation of criticality across requests at different levels of criticality-rank thresholds. This variation is measured using the PCR.

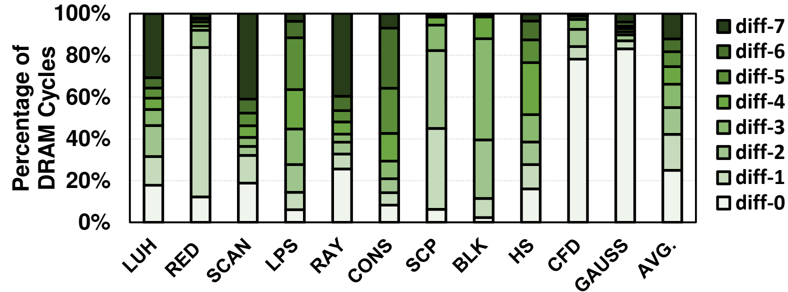


Figure 5.6: Distribution of criticality-rank differences across requests.

difference is that PCR(Th_{CR}) considers criticality of requests instead of their corresponding cores. Relaying the rank information embedded in the memory requests to the MC, and then periodically calculating PCR(Th_{CR}) has three primary benefits: (1) this percentage can be calculated locally at the MC without needing communication across MCs, (2) PCR(Th_{CR}) plots over time (shown in Figure 5.5) are highly correlated to the PCC(Th_{CR}) curves shown in Figure 5.3, and (3) the calculations to find appropriate value for Th_{CR} (discussed in Section 5.5) for applications and other optimizations can also be done locally at the MCs. In Figure 5.5, PCR(Th_{CR}) is plotted over time for the same four applications shown in Figure 5.3, for the same values of $Th_{CR} = \{1, 4, 7\}$. It is observed that in both Figure 5.3 and Figure 5.5, applications have very similar patterns. Thus, a criticality-aware memory scheduler could make scheduling decisions based on PCR(k) $\forall k \in \{1...8\}$ information calculated locally at the MC, instead of using the global PCC(k), $\forall k \in \{1...8\}$ information.

Scope of Criticality Aware Scheduling: To understand the scope of criticality-aware memory scheduling in GPU workloads, the existence of memory requests is investigated with different criticality-ranks in MCs at the same instance. Figure 5.6 depicts the distribution of criticality-rank differences across DRAM requests, where

criticality-rank difference is defined as the difference between the highest and lowest criticality-rank of the memory requests present in the MC at the same instance. This data is for one of the MCs (similar distributions are observed in other MCs), when more than one request is present in the MC queue. In Figure 5.6, `diff-0` denotes the percentage of DRAM cycles during which all the memory requests in the queue have the same criticality-rank. Similarly, `diff-1` denotes the percentage of DRAM cycles during which the difference between the highest and the lowest rank of the memory requests in the MC at the same instant is 1. Note that as the maximum possible rank is 8, the difference of the highest and the lowest rank can range from 0 to 7. It is observed from Figure 5.6, the rank range present in the MC varies across applications. In applications such as `LUH` and `RAY`, the difference in ranks is significant during most of the execution, while in other applications such as `GAUSS` and `CFD`, the difference is mostly 0. Therefore, in these applications, the scope of criticality-aware memory scheduling is not significant. It is observed that many applications (e.g. `RAY`, `LUH`, `CONS`, `RED`) have enough scope for criticality-based prioritization.

These results lead us to a conclusion that a memory scheduler that exploits the criticality within and across cores has scope to improve system performance.

5.5 CLAMS: Design and Implementation

5.5.1 Design Challenges of CLAMS

(I) Co-existence of critical and non-critical requests: In order to allow criticality-based prioritization, one of the important challenges is to find Th_{CR} such that both critical and non-critical requests coexist in the MC queue. From the prior discussions, it is observed that a high value of Th_{CR} might lead to *too* many cores and their corresponding requests to be considered as critical. On the other hand, a very low value might lead to a *very* small number of cores and their corresponding requests to be considered as critical. In both the scenarios, MC queue only contains either only critical requests or only non-critical requests.

This prevents the scheduler to take advantage of the differences between the ranks of the requests in the MC queue. Therefore, *to increase the opportunities for criticality-based prioritization by distinguishing critical requests from the others, it is imperative to find an appropriate value of* Th_{CR} *that can enable substantial coexistence of critical and non-critical requests in the system.*

(II) Balancing DRAM access locality and criticality: Even though choosing an appropriate Th_{CR} provides the co-existence of both critical and non-critical requests in the system, it might not achieve a good balance between locality and criticality. To address this trade-off, PCR(Th_{CR}) is periodically calculated to switch between scheduling policies optimized for criticality or locality. Over the execution, if PCR(Th_{CR}) is *below* a threshold, which is called Scheduling-Mode-Threshold (Th_{SM}), the scheduler prioritizes critical requests to ensure that the cores that cannot tolerate latency are serviced faster. However, during execution, if it exceeds the threshold, it implies that the differences in latency criticality has become insignificant and there are too many critical requests to prioritize one over another. However, it is challenging to find the appropriate value of Th_{SM}, because a higher value of Th_{SM} would push the scheduler to serve critical requests for a longer time, hampering the locality. A lower value of Th_{SM} would not leave enough opportunities for criticality based prioritization. Therefore, it is imperative to find an appropriate Th_{SM} value to balance locality and criticality.

5.5.2 Design Overview of CLAMS

Three different schemes are proposed for calculating Th_{CR} and Th_{SM}. The first scheme is called as Static-CLAMS because it uses a *single and fixed* set of values for Th_{CR} and Th_{SM} for all the applications. However, it is found that these fixed and independent choices of Th_{CR} and Th_{SM} make it difficult to simultaneously address discussed design challenges. Therefore, the second scheme, called Semi-Dyn-CLAMS, dynamically calculates Th_{CR} based on: 1) a fixed value of Th_{SM}, and 2) PCR(k), $\forall k \in \{1...8\}$ information, calculated within an MC (Section 5.4). This scheme dynamically finds Th_{CR} but still uses static values of Th_{SM}. The

third scheme is called Dyn-CLAMS because it dynamically calculates both Th_{CR} and Th_{SM}. It uses Semi-Dyn-CLAMS to calculate Th_{CR} and then dynamically updates Th_{SM} based on the calculated Th_{CR}. The thresholds calculated by these schemes are used to determine the working mode of CLAMS. Two working modes in which CLAMS scheduler can issue requests to the banks are:

(A) Locality mode: This is the default mode in which CLAMS is locality-focused. It prioritizes: 1) row-hit requests over all other requests, 2) critical requests over all other requests, 3) older requests over younger ones. However, if there are no critical requests present, this mode follows the FR-FCFS scheduling policy, which prioritizes: 1) row-hit requests over all other requests, 2) older requests over younger ones.

(B) Criticality mode: In this mode, CLAMS is criticality-focused, and optimizes mainly for criticality. It prioritizes: 1) critical requests over all other requests, 2) older requests over younger ones. However, if there are no critical requests present, this mode falls back to the default locality mode.

Mode Selection: The decision to be in the criticality or locality mode is based on the value of PCR related to every bank ($PCR_b(Th_{CR})$), which is defined as the ratio between the number of critical requests destined to b^{th} bank and the total number of requests destined to b^{th} bank. A particular mode is decided based on Eq.5.1.

$$PCR_b(Th_{CR}) \begin{cases} \leq Th_{SM} & \textit{criticality mode} \\ > Th_{SM} & \textit{locality mode} \\ = 0 & \textit{criticality} = \textit{locality mode.} \end{cases} \tag{5.1}$$

In the special case, when there are no critical requests destined to b^{th} bank ($PCR_b(Th_{CR}) = 0$), criticality and locality mode follow exactly the same request service order.

Inter vs. Intra Core Criticality: The intuition behind switching to the criticality mode is to prioritize critical memory requests over other requests belonging to *different* cores. However, because of the procedure which is followed to tag criticality rank to the memory requests, it might happen that both critical

and non-critical requests from the **same** core might co-exist in an MC. As the schemes (explained next) do not explicitly consider core-ids while serving requests, it might happen that prioritization mechanism may prefer a critical request over another; both belonging to the *same* core. This procedure has limited benefits, because requests from the same core has similar utility unless they have *intra-core* criticality typically caused due to memory divergence [83]. This work is more interested in *inter-core* criticality, which is due to the fact that all the cores do not have the same level of latency tolerance. The detailed analysis shows that the schemes benefit more from *inter-core* criticality. On average, the criticality mode prefers critical requests over non-critical requests 76% times belonging to *different* cores.

5.5.3 Design of Static-CLAMS Memory Scheduler

This is the simplest among the proposed schemes that uses fixed values of both the thresholds to identify the working mode. However, such fixed and independent choices of threshold values make both challenges harder to achieve (Section 5.5.1). To understand this, consider Figure 5.7, where it is shown that the distribution of memory requests for one of the MCs (distribution for other MCs are similar) across different ranks when executed on the baseline architecture that employs FR-FCFS. It is observed from the AVG. bar, that Th_{CR}=4 leads to co-existence of both critical and non-critical requests in the MC queue. However, this value of Th_{CR} does not provide substantial coexistence in *every* application. For example in `SCP`, with Th_{CR}=4, majority of the requests are critical. Therefore, with this value of Th_{CR} along with Th_{SM}=20%, the scheduler will be in locality mode *most of the time*[7], as it will detect that there are too many critical requests in the MC. On the other hand, with Th_{SM}=80%, the scheduler will be mostly in criticality mode. Such set of values can degrade the DRAM row buffer locality, leading to significant loss in performance. *From this discussion, it is concluded that: (1) there is a need for application-based* Th_{CR}*, and (2)* Th_{CR} *and* Th_{SM} *should not be*

[7] Although, actual working mode selection is based on Eq. 5.1, where $PCR_b(Th_{CR})$ (and not PCR(Th_{CR})) is used, in such cases the scheduler most of the time prefers locality over criticality mode, however it is not always true. More details in Section 5.5.4.

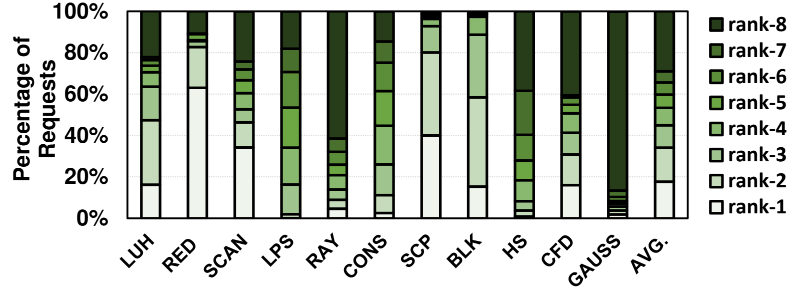


Figure 5.7: Distribution of requests in different criticality-rank states.

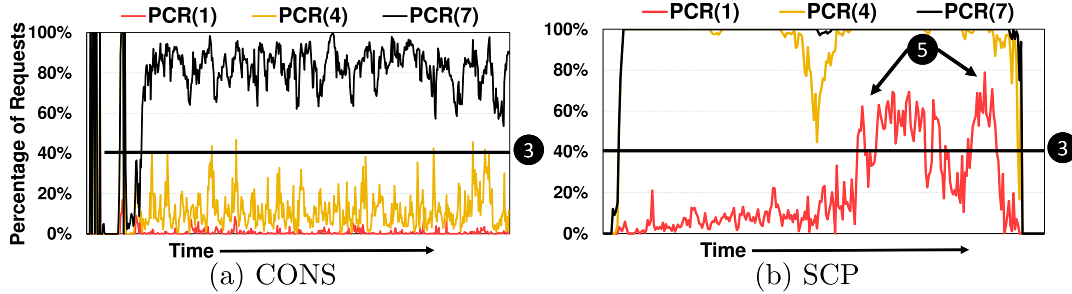
determined independent of each other.

5.5.4 Design of Semi-Dyn-CLAMS Memory Scheduler

The primary goal of Semi-Dyn-CLAMS is to calculate Th_{CR} with the help of a fixed value of Th_{SM} and PCR(k), $\forall k \in \{1...8\}$ information.[8] This procedure achieves two additional sub-goals. First, Semi-Dyn-CLAMS makes Th_{CR} dependent on Th_{SM}, as it calculates Th_{CR} dynamically based on the fixed Th_{SM} value. Therefore, Semi-Dyn-CLAMS does not determine Th_{CR} and Th_{SM} independently, which is desirable based on the discussion in Section 5.5.3. Second, it makes Th_{CR} and Th_{SM} cognizant of all the requests in the MC request buffer, i.e., the PCR(k), $\forall k$ information. This is important because as PCR(k) and PCC(k) values are correlated (Section 5.4), making Th_{CR} and Th_{SM} aware of PCR(k), in turn, make them aware of the current state of variation in criticality across all cores.

This scheme dynamically finds Th_{CR} such that the percentage of requests that are critical, denoted by PCR(Th_{CR}) is less than or equal to a fixed Th_{SM} value, but also as close as possible to Th_{SM}. In other words, we need to find highest Th_{CR} such that PCR(Th_{CR}) is less than or equal to a fixed Th_{SM} value. After obtaining such Th_{CR}, scheduler will mostly be in criticality mode because we have ensured that over a window, PCR(Th_{CR}) is less than or equal to Th_{SM}. However, after being in the criticality mode, if PCR (even with $Th_{CR} = 1$) exceeds the fixed

[8]This information is updated every 512 cycles. Three other sampling size windows (256, 1024, 2048) cycles are also used. The difference in overall average performance is less than 1%, implying that sampling window size does not have a significant impact on the design.



(a) CONS (b) SCP

Figure 5.8: Execution of CONS and SCP to illustrate the working of Semi-Dyn-CLAMS. Th_{CR} values are calculated dynamically.

value of Th_{SM}, the scheduler switches to the locality mode, because the scheme detects that there are *too* many critical requests to prioritize one over another and hence, the latency tolerance variation across cores is not significant. Therefore, in such scenarios, we set Th_{CR}=8, which makes all requests considered to be critical. Such value of Th_{CR} will always drive the scheduler to locality mode, because PCR(8) is always equal to 1 and greater than Th_{SM}. This work finds that Th_{SM} prefers to be in mid-range (40% provided the best average performance results, Section 5.7). This is expected because, mid-range percentage of critical requests allows the coexistence of both critical and non-critical requests in MC.

Figure 5.8 illustrates Semi-Dyn-CLAMS, where we choose Th_{SM}=40% (❸). Assume only three values of Th_{CR} = {1,4,7} are possible, however in the final evaluation, we consider the full range from 1 to 8. We observe from Figure 5.8a that in `CONS` Th_{CR} will be chosen mostly as 4 because this value of Th_{CR} is the highest possible value of Th_{CR} such that PCR(Th_{CR}) is less than or equal to Th_{SM}=40% (❸). Therefore, in `CONS`, the scheduler mostly will be in criticality mode. In `SCP` (Figure 5.8b), the situation is different. By choosing the same value of Th_{SM}=40% (❸) in the first half of the execution, the scheduler will be in criticality mode most of the time, as PCR(1) is lower than Th_{SM}=40% (❸). However, during the second half of the execution, SCP prefers locality mode (❺), where PCR(1) line is above the horizontal line (❸). During this time, there is no Th_{CR} such that PCR(Th_{CR}) is less than Th_{SM}=40% (❸), and hence the scheme detects that there are too many critical requests (even with Th_{CR}=1). We set the scheduler to go in locality mode by setting Th_{CR}=8.

Discussion: Recall that the actual mode selection is based on Eq.5.1, which compares the value of $PCR_b(Th_{CR})$ (and not PCR(Th_{CR})) with the value of Th_{SM}. Therefore, even though Semi-Dyn-CLAMS select a value of Th_{CR} such that PCR(Th_{CR}) is lower than (or equal to) Th_{SM}, it is not necessary that $PCR_b(Th_{CR})$ will also be lower than (or equal to) to Th_{SM}. Therefore, even though Semi-Dyn-CLAMS overall strives to keep the scheduler in the criticality mode, while ensuring that both critical and non-critical requests have substantial presence in MC, during actual issue of requests to the memory banks, the scheduler can be in any of the modes – locality or criticality. However, if, $\forall k \in \{1..8\}$, PCR(k) is greater than the Th_{SM} value, the scheduler switches to the locality mode by setting the Th_{CR} value to 8. This value of 8 will always switch the scheduler to the locality mode, because as per definition, both PCR(8) and $PCR_b(8)$ are always equal to 1, and therefore, it will be always greater than Th_{SM}. This analysis is used as a foundation for the next scheme.

Importance of $PCR_b(Th_{CR})$ and PCR(k)$\forall k$: For calculating appropriate Th_{CR}, Semi-Dyn-CLAMS consults PCR(k), $\forall k$ information, but the scheduler makes actual decisions on the working modes based on the current set of requests to be issued to the bank, i.e., by examining $PCR_b(Th_{CR})$. This has two advantages. First, the actual mode decision is aware of the current state of the requests destined to the bank. Second, this decision is also aware of the status of all the requests in the MC, which in turn is also aware of PCC information.

5.5.5 Design of Dyn-CLAMS Memory Scheduler

We find Semi-Dyn-CLAMS as an aggressive design in taking advantage of criticality because of two reasons: First, Semi-Dyn-CLAMS always strives to find opportunities to work in the criticality mode. Second, Semi-Dyn-CLAMS goes to locality mode only when there are too many critical requests at Th_{CR}=1 (PCR(1) $> Th_{SM}$) in the MC queue. Because of these two reasons, we observe significant loss in locality and performance in some applications (e.g., `SCP` and `RAY`).

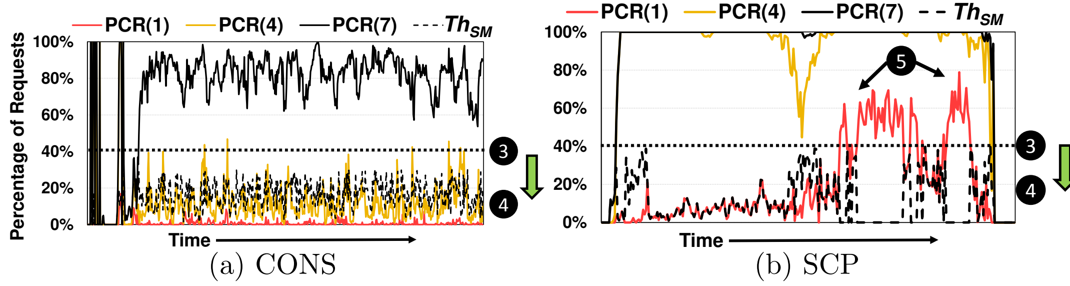
Even though Semi-Dyn-CLAMS calculates an Th_{CR} that facilitates the

Table 5.1: Pseudo code for the proposed schemes

procedure [P1]	**procedure [P2]**
Semi-Dyn-CLAMS	**Dyn-CLAMS**
$Th_{CR} = 8$.	$Th_{SM} = Th_{SMinit}$; $Th_{CR} = 8$.
for $k \in \{1...7\}$ **do**	**for** $k \in \{1...7\}$ **do**
if ($0 <$ PCR(k) $\leq Th_{SM} <$ PCR(k+1))	**if** ($0 <$ PCR(k) $\leq Th_{SM} <$ PCR(k+1)) **then**
then	Th_{CR} = k; Th_{SM} = PCR(Th_{CR}).
Th_{CR} = k.	**end if**; **end for**
end if; end for	**if** $Th_{CR} = 8$ **then** $Th_{SM} = 0\%$. **end if**
return Th_{CR}.	return (Th_{CR}, Th_{SM}).

scheduler to be in the criticality mode, when it actually issues requests to the bank, it might prefer locality mode based on $PCR_b(Th_{CR})$ value (Section 5.5.4). The goal of Dyn-CLAMS is to improve locality by increasing such opportunities. To do so, we exploit one of the important limitations of Semi-Dyn-CLAMS that it still uses a fixed value of Th_{SM}. In other words, in Semi-Dyn-CLAMS, Th_{CR}-Th_{SM} dependence is only *one way*, and Th_{SM} is not updated based on the calculated Th_{CR} value. Therefore, the *key idea* of Dyn-CLAMS is to first gauge the negative effect of the loss in row-locality on the latency tolerance (i.e., performance) of the cores by dynamically examining Th_{CR}, and then restoring the loss by lowering the value of Th_{SM} as much as possible while maintaining the same value of Th_{CR} calculated using Semi-Dyn-CLAMS. This is because, with a lower value of Th_{SM}, the scheduler will work in the locality mode (see Eq. 5.1). Dyn-CLAMS uses exactly the same procedure as adopted by Semi-Dyn-CLAMS to calculate Th_{CR}, but in addition, also lowers Th_{SM}. At the beginning of every window, we start with a fixed Th_{SM} value (Th_{SMinit}) to determine Th_{CR} using Semi-Dyn-CLAMS. Value of Th_{SMinit} is equal to 40%, which we calculated based on extensive experimental evaluation. After calculating Th_{CR}, we update (lower) the value of Th_{SM} to PCR(Th_{CR}). By doing so, Th_{CR} remains the same as per Semi-Dyn-CLAMS scheme, but Th_{SM} is reduced. Thus, both Th_{CR} and Th_{SM} values are updated dynamically.

Figure 5.9 illustrates this scheme. For `CONS`, we observed in Semi-Dyn-CLAMS (Figure 5.8) that Th_{CR} is usually 4, but in Dyn-CLAMS, while maintaining the same Th_{CR}, the value of Th_{SM} is lowered (pointed by ❸ $\rightarrow$ ❹) such that it closely resembles the PCR(4) curve. Similarly, in `SCP`, the value of Th_{SM} is lowered to



(a) CONS (b) SCP

Figure 5.9: Execution of CONS and SCP to illustrate the working of Dyn-CLAMS. Th_{SM} is dynamically updated based on Th_{CR}.

match closely with PCR(1). However, in cases when SCP prefers locality mode (❺), Semi-Dyn-CLAMS sets Th_{CR} to 8, and Dyn-CLAMS sets Th_{SM} to 0%, making the scheduler work in locality mode. Algo. 5.1 formally describes the procedures adopted by the schemes.

5.5.6 Hardware Overheads

(I) Tagging memory request with core criticality: Each core is assigned with a rank depending on its current state of the latency tolerance. As the maximum possible number of warps on a core is 48, this scheduler needs two 11-bit counters to store the windowed-average of short-latency and issued warps over 32 cycles. The proposal calculates the rank using one 11-bit divider and eight comparators. This rank is stored in a 3-bit register. At the time when memory request is issued, the proposal tag the memory request with the corresponding core's rank.

(II) Static-CLAMS: Two 8-bit up-down counters per-bank (max. MC buffer size is 256) are required to track the number of critical and pending memory requests. For mode selection, this proposal compare the value of $PCR_b(Th_{CR})$ to a fixed value of Th_{SM} with the help of comparator. Th_{CR} (3 bits) and Th_{SM} (7 bits) values are stored as fixed thresholds in registers within MC.

(III) Semi-Dyn-CLAMS: PCR(k)$\forall k$ per MC is calculated over a window of 512 cycles by keeping track of the critical requests and the number of total requests in MC. This proposal needs one 9-bit counter per rank and one 9-bit counter to keep track of the pending memory requests. The scheduler takes a snapshot of these

Table 5.2: Key configuration parameters of the simulated GPU configuration.

Core Features	1400MHz core clock, 32 cores (streaming multi-processors), SIMT width = 32, Greedy-then-oldest first (GTO) dual warp scheduler [4], Thread-blocks are scheduled on SMs in load-balanced fashion
Resources / Core	48KB shared memory, 32KB register file, Max. 1536 threads
Private Caches / Core	16KB 4-way L1 data cache, 12KB 24-way texture cache 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
Shared L2 Cache	16-way 128 KB/memory channel (768KB in total), 128B cache block size
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Memory Model	6 GDDR5 Memory Controllers (MC), FR-FCFS scheduling, 256 max. common request buffer for all 8 banks per MC, 4 bank-groups/MC, 924 MHz memory clock, Global address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing, t_{CL} = 12, t_{RP} = 12, t_{RC} = 40, t_{RAS} = 28, t_{CCD} = 2, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12
Interconnect	1 crossbar/direction (32 cores, 6 MCs), 1400MHz interconnect clock

Table 5.3: Evaluated applications. Table also shows: 1) Average occupancy (occ) in terms of warps, 2) Average Th_{CR} and Th_{SM} calculated using Semi-Dyn-CLAMS and Dyn-CLAMS, respectively, and 3) % of critical requests (%-cri) served in the criticality-mode.

Application	Abbr.	Class	occ	Th_{CR}	Th_{SM}	%-cri
Lulesh [90]	LUH	A	18	4	13%	46
Reduction [91]	RED	A	15	1	16%	38
Scan [91]	SCAN	A	14	4	16%	42
Laplace 3D [23]	LPS	A	11	4	23%	32
Ray Tracing [23]	RAY	A	16	6	11%	40
Convolution [23]	CONS	A	15	5	21%	45
Scalar Product [23]	SCP	B	30	1	10%	66
BlackScholes [23]	BLK	B	32	2	8%	70
Hotspot [42]	HS	B	23	6	17%	17
CFD Solver [42]	CFD	B	45	2	4%	10
Gaussian [42]	GAUSS	B	8	6	6%	2

counters in an extra storage, and then flush the counters. The scheduler then calculates PCR(k)$\forall k$ based on the snapshot values, and store them in 8 PCR(k) registers. To calculate Th_{CR}, this scheduler compares fixed Th_{SM} value with 8 PCR(k) registers.

(IV) Dyn-CLAMS: This scheme updates Th_{SM} with the value of PCR(Th_{CR}); thus does not require extra overhead. The information for all schemes is computed locally at the MCs. The total storage required for one of the MCs is 43B.

5.6 Evaluation Methodology

The baseline architecture is simulated described in Table 5.2 using GPGPU-Sim v3.2.1 [31], a cycle-accurate GPU simulator. The architecture has similar compute to memory bandwidth ratio as that of GTX Titan. This work studied 25 applications from various suites such as SDK [23], Rodinia [42], LLNL [90], and SHOC [91], and report results for 11 of them that have more than 5% average COV in IPCs and latencies (Table 5.3). None of the studied applications exhibit performance degradation, because CLAMS is able to dynamically adjust its thresholds. This work classifes 11 applications into two classes. Class-A applications show high to moderate scope for criticality-aware scheduling because of the presence of variation in criticality across cores (see Figure 5.1). The other applications are classified as Class-B because of low scope for criticality-aware scheduling (see Figure 5.6 and Figure 5.1). All the applications are executed until completion, except for `LUH`, `CONS`, and `CFD`, where this work executes 500 million instructions [7, 27].

5.7 Experimental Results

This section analyzes the performance of three CLAMS designs, along with two more memory schedulers: FR-FCFS-Cap-Best and Static-CLAMS-Best. FR-FCFS-Cap (streak control) scheduler enforces a *cap* on the younger row-hit requests that can be serviced before an older row access to the same bank. When the cap is reached, FCFS policy is applied. While such a cap alleviates the starvation problem for waiting requests, it is not aware of the criticality of requests it is servicing. The results of FR-FCFS-Cap-Best are shown that picks the best performing *cap* threshold profiled separately for *each* application. Evaluated choices for *cap* values are 2, 4, 6, 8, 12 and 16. Static-Best-CLAMS is the static best scheduler that uses the best performing combination of Th_{CR} and Th_{SM} profiled separately for *each* application. In contrast to Static-Best-CLAMS, Static-CLAMS uses a *single* set of thresholds (Th_{CR}=4 and Th_{SM}=20%) that provides best average performance across all applications. The

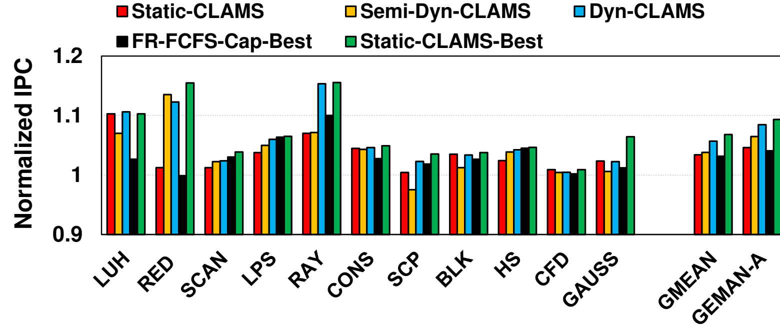
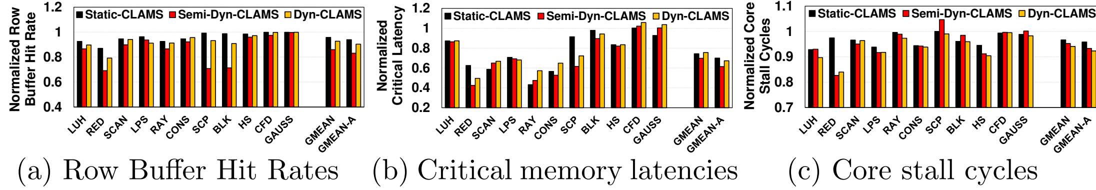


Figure 5.10: Performance results normalized to FR-FCFS.

values of these thresholds are chosen from a pool of 42 (7 × 6) different combinations formed using fixed values of Th_{CR} (1 through 7) and Th_{SM} (0% through 100% in steps of 20%). **Note that both FR-FCFS-Cap-Best and Static-Best-CLAMS are hard to implement as they require an exhaustive search across many threshold combinations.** It is observed that the FRFCFS-Cap results are very sensitive to thresholds and a single threshold does not work well for all the applications. Dynamic adaptation of thresholds is non-trivial and that is why CLAMS is proposed.

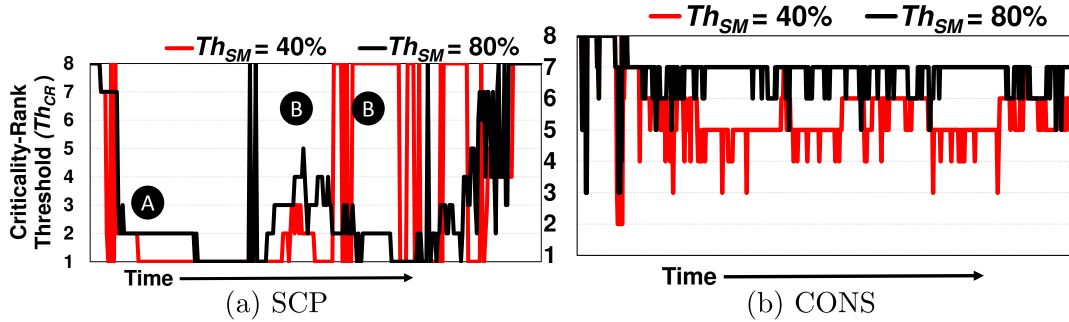
Figure 5.10 shows the IPC improvement for proposed memory schedulers. Two averages are provided: GMEAN for all applications, and GMEAN-A only for Class-A applications. Auxiliary metrics are also presented related to the DRAM and core in Figure 5.11. Figure 5.11a shows the Row Buffer Hit Rates (RBHR) to measure the locality metric. Figure 5.11b depicts the latency reduction of critical memory requests (with respective values of Th_{CR}), and Figure 5.11c shows the reduction in core cycle stalls during which GPU cores are not able to issue any warps. This reduction in stall cycles is attributed to the prioritization schemes for critical requests. All results are normalized to the baseline FR-FCFS scheduler.

(A) Analysis of Static-CLAMS: Using a single set of thresholds (Th_{CR}=4 and Th_{SM}=20%) for all the applications do not lead to significant improvements over FR-FCFS in `SCP` and `RED`. It is expected because with Th_{CR}=4, high percentage of requests are treated as critical, and using Th_{SM}=20% along with it pushes the scheduler to mostly work in locality mode. However, for `LUH`, 10% IPC improvement is observed, because these thresholds address both the



(a) Row Buffer Hit Rates (b) Critical memory latencies (c) Core stall cycles

Figure 5.11: Effect on (a) DRAM page hit rates, (b) memory latencies for critical requests, (c) core stall cycles. Results are normalized to FR-FCFS.

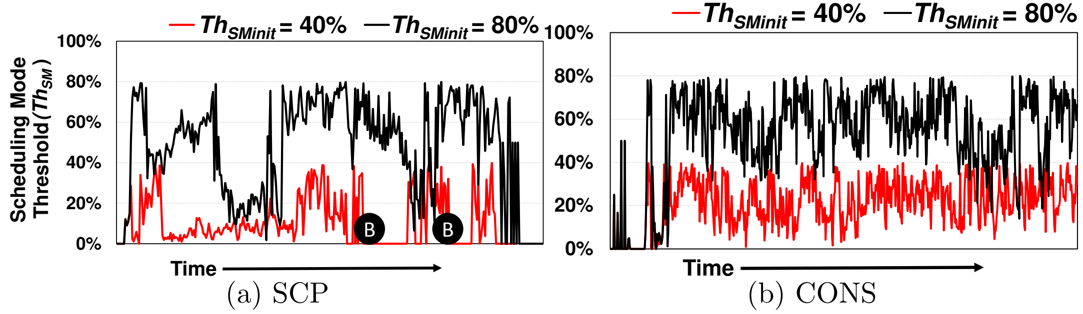


(a) SCP (b) CONS

Figure 5.12: Changes in Th_{CR} are observed when Semi-Dyn-CLAMS is employed. When Th_{CR}=8, scheduler is in locality mode.

challenges reasonably (Section 5.5.1). On average, Static-CLAMS provides 3% IPC improvement for all 11 applications. Although none of the applications experience performance degradation, this scheme is still far from Static-Best-CLAMS.

(B) Analysis of Semi-Dyn-CLAMS: The dynamic changes are first analyzed in Th_{CR} calculated by Semi-Dyn-CLAMS scheme for two applications – `SCP` and `CONS`. Figure 5.12 shows these results for two fixed values (40%, 80%) of Th_{SM}. We first start when Th_{SM}=40%. In `SCP`, Semi-Dyn-CLAMS chooses Th_{CR}=1 in the first half of the execution (Ⓐ) (as expected from the discussion in Section 5.5). In the second half of the execution (Ⓑ), we observe many switches to Th_{CR}=8 as it detects that there are too many critical requests and hence it switches to locality mode. In `CONS`, the scheme chooses Th_{CR} between 4 and 5 and mostly remains in the criticality mode. However, in the case with Th_{SM} equals to 80%, we observe increase in Th_{CR} values for SCP in the first half of the execution (Ⓐ), and also it switches to the locality mode less often (less Th_{CR}=8 values are observed). It is expected because at higher value of Th_{SM}, the scheduler will go more often to



(a) SCP (b) CONS

Figure 5.13: Changes in Th_{SM} are observed with Dyn-CLAMS.

criticality mode.

On average, Semi-Dyn-CLAMS provides 5% IPC improvement over FR-FCFS. RED, RAY, and LPS achieve 13%, 7%, and 5% improvement, respectively. As desired, this scheme attempts to push the scheduler mostly to criticality mode, which helps in reducing the latency of critical requests further by 6% (at the cost of 10% reduction in RBHR) compared to Static-CLAMS. This in turn reduces the core stall cycles further by 2% compared to Static-CLAMS. In SCP, RBHR is hampered the most (30%) leading to 5% performance degradation compared to FR-FCFS. On the other hand, in RED, even though RBHR is reduced by 20%, performance improves significantly (by 12%) due to the reduction in critical request latency and core stall cycles. The detailed analysis shows that, in RED, the impact of locality on performance is much lower than that of criticality, and vice versa in SCP. The next scheme, Dyn-CLAMS is expected to recover the loss in locality and performance by reducing the Th_{SM} value dynamically.

(C) Analysis of Dyn-CLAMS: We first analyse the dynamic changes in Th_{SM} calculated by Dyn-CLAMS scheme for two applications – SCP and CONS. Figure 5.13 shows these changes when $Th_{SMinit} = 40\%$ and 80%. We start with analysing Th_{SM} curves when Th_{SMinit}=40%. In SCP and CONS, we find that the value of Th_{SM} is less than or equal to 40%. This helps the scheduler to go to the locality mode more frequently as discussed in Section 5.5.5. During the phases when Th_{CR}=8 (as pointed in Figure 5.12 by **B**), Th_{SM} value is 0% (**B**), pushing the scheduler to be always in the locality mode. For $Th_{SMinit} = 80\%$ curves, we observe the value of Th_{SM} is much higher because of the increase in Th_{CR} values.

On average, Dyn-CLAMS performs better than all three memory scheduling schemes. `RED`, `RAY`, and `LUH` are the best performers with 15%, 15%, and 10% improvement over FR-FCFS, respectively. This scheme is especially useful for applications in which locality is also important. For example, in `SCP` and `RAY`, RBHR is improved by 5% and 7%, respectively, leading to additional benefits over Semi-Dyn-CLAMS. We also observe reduction in PCC(1) for these applications (22%, 25%, and 6%, respectively), as expected from the discussion in Section 5.3.2. It is further observed from Figure 5.10 that the gap between Dyn-CLAMS and Static-CLAMS-Best is not significant for many applications, suggesting that Dyn-CLAMS is able to dynamically calculate the best static combinations of thresholds for each application, as shown in Table 5.3, without the need of any offline application profiling. This work also reports percentage of critical requests that are served in the criticality mode (%-cri) in Table 5.3. For `SCP` and `BLK`, even though %-cri is very high, Dyn-CLAMS does not show benefit because the *number* of critical requests is itself small. For Class-A applications, %-cri is significant, which shows that Dyn-CLAMS is able to improve performance by prioritizing critical requests. In summary, Dyn-CLAMS achieves 9% IPC improvement over FR-FCFS, 5% over FR-FCFS-Cap-Best, and also is within 1% of the Static-CLAMS-Best for Class-A applications. For Class-B applications, none of the applications degrades with the final scheme.

To get a deeper understanding into performance results, Figure 5.14 shows the break down of the memory bandwidth for FR-FCFS and Dyn-CLAMS schemes to the following components: (A) App-BW: the percentage of DRAM cycles during which the application moves data (reads and writes) over the DRAM interface, (B) Waste-BW: the percentage of DRAM cycles during which no data is transferred over the DRAM interface, but there are pending memory requests in MC queue due to DRAM timing constraints; improving RBHR, and bank-level parallelism (BLP) can reduce this Wasted-BW, and (C) Idle-BW: the percentage of DRAM cycles during which there are no requests pending in the MC queues, and hence DRAM is idle. It is observed that IPC and App-BW are highly correlated, which is consistent with the findings shown by Guz et al. [84, 92]. It is further observed that Waste-BW increases in `LUH` and `RAY` because of loss in RBHR. It is expected as the loss in locality causes more row conflicts. However, in `SCAN`, in spite of the

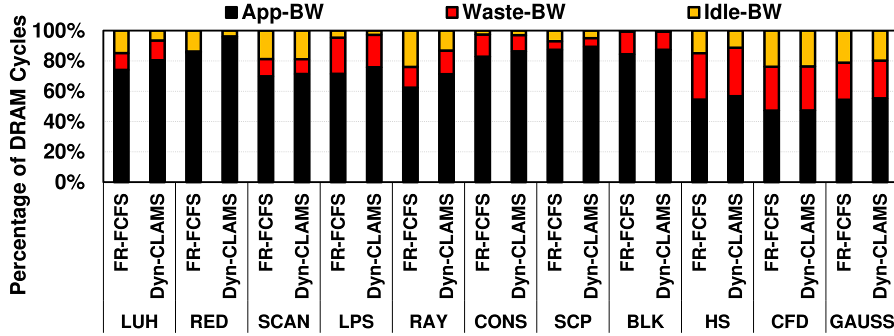


Figure 5.14: Effect of Dyn-CLAMS on DRAM bandwidth distribution.

reduction in RBHR, minimal or no reduction in Waste-BW is observed. This is because Dyn-CLAMS facilities more cores to be active at a time, also allowing requests from more cores to take advantage of BLP. This increase in BLP helps in masking the ill-effects of the loss in locality.

(D) Brief Summary of Sensitivity Studies: For Semi-Dyn-CLAMS, increasing Th_{SM} value from 20% to 40% improves the performance of applications (e.g., `CONS`, `RED`) that prefer criticality mode. However, beyond 40%, performance of applications (except `RED`) saturate, where performance starts dipping after Th_{SM}=60% because of the steep decrease in RBHR. Similar trends for Th_{SMinit} are observed in Dyn-CLAMS. On average, 40% for both Th_{SM} and Th_{SMinit} leads to best average performance results across all applications.

5.8 Related Work

This section summarizes the prior work related to this chapter.

CPU Memory Scheduling: Ebrahimi et al. [93] propose parallel application memory scheduling, where they explicitly manage the inter-thread memory interference for improving the performance of the critical section of a program. Ghose et al. [94] used load-criticality information for effective memory scheduling. In contrast to their static memory controller policy that always utilizes criticality, this dissertation work develop memory scheduling mechanisms that can *dynamically* switch between criticality and locality modes. By exploiting

inter-core criticality, a novel concept this dissertation work developed especially for GPUs, this work determines the duration of time for which the scheduler should be in one of those modes. Ausavarungnirun et al. [51] propose a staged memory scheduler to improve row-buffer locality in CPU+GPU architectures. Other memory scheduling works include ATLAS [46], PAR-BS [37], STFM [47], TCM [45], and BLISS [95]. These memory schedulers concentrated only on single-threaded or modestly multi-threaded/multi-programmed workloads, while this dissertation work demonstrates the benefits of the schemes for massively-threaded applications. Scaling the conventional CPU memory scheduling policies for thousands of threads in a GPU would be challenging in GDDR5 MC that has to support multi-gigabit command issue rates. Further, many of these schedulers need to track *each thread's* MLP, bank-level parallelism (BLP), and row-buffer locality (RBL), and requires an expensive insertion sort-like procedure to shuffle the ranks of high-MLP applications. In contrast, CLAMS only requires a few counters, comparators, and moderate storage (Section 5.5.6). Moreover, this dissertation work does not employ coordination across MCs, as they may cause significant performance/power overheads in GPUs.

Criticality Related Studies in CPUs: Srinivasan et al. [96] contrasts criticality and locality by performing a limit study that gives the maximum potential of exploiting critical memory requests. A follow-up work [97] gives a practical algorithm for identifying critical memory requests. Other works which take advantage of criticality information include [94, 98–102]. In comparison, the criticality definition is calculated as a function of the latency tolerance of the core, which is very different from prior studies.

Memory Scheduling in GPUs: Lakshminarayana et al. [50] explored a DRAM scheduling policy that chooses between Shortest Job First (SJF) and FR-FCFS. Their scheme uses a statically determined parameter that needs to be uniquely calculated for each application. As this scheduler does not adapt to dynamic needs of criticality and locality in the application, it can even degrade the performance compared to FR-FCFS [50]. On the other hand, CLAMS *dynamically* detects the preferred mode (locality or criticality) for applications at

runtime for better performance while making sure that performance of none of the applications degrades. Yuan et al. [49] propose an arbitration mechanism in the interconnection network to restore the lost row buffer locality caused by the interleaving of requests. They showed that performance of in-order DRAM is competitive to FR-FCFS. This chapter shows qualitatively and quantitatively that CLAMS outperforms FR-FCFS.

Warp Scheduling: Narasiman et al. [8] and Gebhart et al. [48] proposed two-level warp schedulers to improve latency tolerance and energy consumption in GPUs, respectively. Rogers et al. [67] and Jog et al. [7] proposed warp schedulers to reduce contention in caches. Lee et al. [89] proposed criticality-aware warp scheduler that prefers critical warps over others for better latency tolerance. None of these works specifically coordinate with the underlying memory schedulers for orchestrated warp and memory scheduling decisions. CLAMS provides a substrate to foster such research, as it incorporates the core-criticality information while making memory scheduling decisions.

5.9 Chapter Summary

This chapter introduces a new GPU memory scheduler, called CLAMS. The unique feature of this scheduler compared to all other existing techniques is that it uses both criticality and locality of pending memory requests to service the next request, while prior schemes use only the locality metric. The rationale for using this dual parameter based scheduling is that not all memory requests for an application exhibit similar latency criticality, and servicing them ahead of the other requests improves application performance. The evaluations show that CLAMS can provide significant performance benefits for the class of applications that exhibit high variance in criticality across cores, without hurting the performance of other applications. Considering that GPUs applications are more latency tolerant than their CPU counterparts due to high TLP, enhancing performance benefits through memory scheduling is non-trivial. This chapter shows that considering core criticality is a promising way of improving GPU performance and can be exploited in GPU and CPU-GPU memory systems.

Chapter 6

Concurrent Kernel Execution in GPUs: Problems and Some Solutions

The available computing resources in modern GPUs are growing with each new generation. However, as many general purpose applications with limited thread-scalability are tuned to take advantage of GPUs, available compute resources might not be optimally utilized. To address this, modern GPUs will need to execute multiple kernels simultaneously. As current generations of GPUs (e.g., NVIDIA Kepler, AMD Radeon) already enable concurrent execution of kernels from the same application, this chapter addresses the next logical step: executing multiple concurrent applications in GPUs. This chapter shows that while this paradigm has a potential to improve the overall system performance, negative interactions among concurrently executing applications in the memory system can severely hamper the performance and fairness among applications. It is shown that current application agnostic GPU memory system design can (1) lead to sub-optimal GPU performance; and (2) create significant imbalance in performance slowdowns across kernels. Thus, this chapter argues that GPU memory system should be augmented with application awareness. As one example to the applicability of this concept, the memory system hardware is augmented with the application awareness such

that requests from different applications can be scheduled in a round robin (RR) fashion while still preserving the benefits of the current first-ready FCFS (FR-FCFS) memory scheduling policy. Evaluations with different multi-application workloads demonstrate that the proposed memory scheduling policy, first-ready round-robin FCFS (FR-RR-FCFS), improves fairness and delivers better system performance compared to the existing FR-FCFS memory scheduling scheme.

6.1 Introduction

Traditionally, GPUs were designed to execute only a *single* kernel at a time; it was expected that a single kernel would have enough threads to keep all GPU resources busy. However, as GPU resources are growing with each new generation (for example, the state-of-the-art Kepler Architecture model GTX 780 Ti has 2880 cores [103], and AMD Radeon R9 290X has 2816 cores [104]), and as more irregular and general-purpose applications are ported to GPUs, many kernels will not be able to proportionally scale [105] and effectively utilize the growing compute resources. To address this problem, a new GPU computing paradigm was recently introduced, where multiple kernels can be executed concurrently on the same GPU platform. This paradigm has two primary advantages. First, it significantly improves the GPU efficiency, as shown in Fermi White Paper [4] and Pai et al. [105]. Second, it facilitates the consolidation of jobs from multiple independent users on to the same GPU substrate, as demonstrated by NVIDIA GRID technology [106].

Figure 6.1 shows this new computing paradigm pictorially. Figure 6.1 (A) shows the traditional GPU architecture, where all cores are executing a single kernel. In Figure 6.1 (B), the same GPU architecture is concurrently executing multiple kernels. Broadly speaking, these multiple kernels can originate from: (i) a single application, or/and (ii) multiple independent applications (contexts). Although this new computing paradigm is an effective way to increase GPU performance and resource utilization, many architectural challenges need to be addressed to unlock its full potential. Open issues that have not been sufficiently explored include: efficient hardware support for execution of multiple applications[1], finding

[1]This chapter assumes that concurrently executing kernels originate from separate

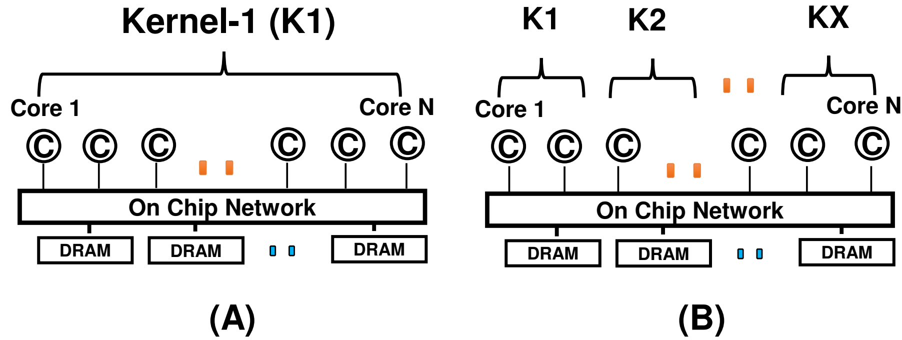


Figure 6.1: Baseline GPU architecture executing: (A) Single kernel, (B) Multiple kernels concurrently.

the optimal number of cores for a particular kernel, designing a QoS-aware on-chip network fabric and memory hierarchy, augmenting the memory hierarchy to include resource sharing policies, and concurrency management for effectively utilizing the on-chip shared resources. Given the wide scope of the problem, this chapter focuses on one aspect of the design space: management and allocation of shared memory system resources in GPUs executing multiple applications concurrently.

The goal is to provide a better understanding of the interactions of concurrent applications in the GPU memory subsystem. This work observes that the traditional GPU memory system is application agnostic, and the widely used First-ready FCFS (FR-FCFS) [32–34] memory scheduler is unaware of the individual demands of concurrent multiple applications. It primarily focuses on improving DRAM page hit rates, and might hurt overall system performance and fairness. This problem becomes more pronounced when the memory demands of concurrently executing applications have wide variation, implying that an application with high memory demand attempts to monopolize the resource usage over an application with low memory demands. To address this, this work augments application-awareness to the memory-system logic and propose the First-ready Round-robin FCFS (FR-RR-FCFS) memory scheduler, which schedules memory requests originating from different applications in a round-robin (RR) fashion, while preserving the benefits of FR-FCFS scheduling. This chapter shows that this simple change to the existing FR-FCFS memory

applications. Hence, the terms kernels and applications interchangeably.

scheduler is a better alternative for concurrent execution of multiple kernels in terms of equitable memory bandwidth sharing and overall system performance.

This chapter makes the following **contributions**:

• This work provides a detailed analysis of the interactions of multiple applications in the GPU memory system.

• This work manifests the fact that a naive coupling and execution of different applications concurrently on modern GPUs with application-agnostic shared resource allocation do not lead to desired results.

• This work shows that one of the primary reasons for sub-optimal performance and unfairness is the application agnostic management of shared resources. The popular FR-FCFS memory scheduler fails to distinguish memory requests originating from multiple applications.

• In this context, this chapter proposes to propagate application awareness to the memory system scheduling logic. As one possible implementation, this chapter suggests a memory controller scheduling policy which not only preserves the benefits of FR-FCFS, but also improves fairness and overall system performance by serving memory requests of different applications in an round-robin fashion. The proposed memory scheduler is evaluated across 14 diverse 2-application workloads on a 60-core GPU simulated platform using GPGPU-Sim [31]. On average, the proposed scheduler provides 7% improvement (up to 49%) in fairness, 10% improvement (up to 64%) improvement in instruction throughput performance, and up to 7% improvement in weighted system speedup.

6.2 Background and Experimental Methodology

This section provides a brief description of the baseline GPU architecture and experimental methodology.

Table 6.1: Simulated baseline GPU configuration

SM Configuration	1400MHz, SIMT width = 16 × 2
Resources / SM	Max. 1536 threads (48 warps, 32 threads/warp) 48KB shared memory, 32684 registers
Caches / SM	16KB 4-way L1 data cache, 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
Default Warp Scheduling	Greedy-then-oldest (GTO) [67]
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Interconnect	1 crossbar/direction (60 SMs, 6 Memory Controllers), 1400MHz
Memory Model	6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling 16 DRAM-banks/MC, 924 MHz memory clock
Hynix GDDR5 Timing [107]	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$

6.2.1 Baseline GPU Architecture

The baseline GPU (see Figure 6.1) consists of multiple cores, named Streaming Multiprocessors (SMs). Each SM is associated with a private L1 data cache, a read-only texture cache and a constant cache. A software managed scratchpad memory is also associated with each SM. SMs are connected to memory channels (partitions) via a crossbar, and memory requests to each partition are handled by a GDDR5 memory controller. The baseline architecture described in Table 6.1 is simulated using GPGPU-Sim v3.2.1 [31], a cycle-accurate GPU simulator.

Single Application Scheduling: A typical CUDA application consists of multiple kernels (or grids). Each kernel is further divided into groups of threads, called cooperative thread arrays (CTAs). Traditionally, GPUs execute all kernels of an application sequentially, i.e. one kernel at a time. In this scenario, when a kernel is launched on the GPU, the CTA scheduler picks available CTAs associated with that kernel and distributes them to the SMs as evenly as possible [31]. The maximum number of CTAs per SM is limited by various resources associated with each SM, and by the resources required by a given kernel [1, 31]. Hence, if a kernel requires less resources, the maximum number of CTAs per SM will be larger than that of another kernel whose CTAs need more resources.

Multiple Application Scheduling: This chapter considers the case where multiple *kernels* from multiple *applications* are executed concurrently, i.e., the kernels from *different* applications are simultaneously executed. Since the focus

of the chapter is the memory system, a simple kernel-to-SM assignment scheme is used: in a two-application scenario where two kernels of different applications are executed concurrently, half of the SMs are assigned to the first application, and the second half to the other application. The sophisticated SM-partitioning techniques are left as a part of the future work. The CTA assignment for each kernel follows the same load-balanced distribution strategy as described before; the only difference is that each kernel is now assigned to only half of the SMs of the baseline GPU architecture.

Memory Scheduling in GPUs: First-ready FCFS (FR-FCFS) [32–34] is the commonly employed memory scheduling technique in GPUs. This scheme is targeted at improving DRAM row hit rates, so request prioritization order is as follows: 1) row-hit requests are prioritized over other requests; then 2) older requests are prioritized over younger requests. Among row-hit requests, older requests are prioritized over younger requests.

6.2.2 Evaluation Methodology

The new generation of GPUs allows concurrent execution of *streams*, where a *stream* is defined as a set of commands required to be executed serially. This mechanism is exploited to issue commands from different applications (kernels) to separate streams, thus allowing them to execute concurrently. Six different CUDA applications are studied; Table 6.2 lists the applications along with their DRAM bandwidth utilization. Note that the considered applications have diverse memory demands – while `GUPS` has the highest memory bandwidth utilization of 93%, `HIST` and `DGEMM` have considerably lower memory bandwidth utilization (50% and 34%, respectively). Section 6.3 shows that memory intensive applications like `GUPS` significantly interfere with co-scheduled applications, leading to poor overall performance and fairness.

From these 6 CUDA applications, all possible two-application workloads are formed and simulated on the GPGPU-Sim simulator. The results of one workload (`bfs_dgemm`) are omitted as the infrastructure could not simulate it faithfully. Table 6.3 lists all 14 two-application workloads. The statistics are

collected at the point when both applications execute to completion at least once. To do so, if one of the applications finishes execution earlier than the other, the the faster running application is again relaunched. This process continues until the slower running application completes.

Table 6.2: Evaluated applications, along with their DRAM bandwidth utilization when they are executed alone on the entire baseline GPU architecture.

Application	Abbr.	Bandwidth Utilization
Histogram	HIST	50%
Gaussian	GAUSS	70%
Random Access	GUPS	93%
Breadth First Search	BFS	79%
3D Stencil	3DS	85%
Matrix Multiplication	DGEMM	34%

6.2.3 Evaluation Metrics

This chapter reports on: (I) Weighted Speedup, for measuring application throughput, (II) Instruction Throughput, for measuring raw machine throughput, and (III) Fairness Index, for measuring fairness in the system. For weighted speedup (WS), the slowdown experienced by each application relative to the case where it runs alone on the entire GPU is measured (Eq.(1)). Note that when an application is running alone, it can use all SMs in the system. The sum of slowdowns of all the concurrent applications is defined as weighted speedup (Eq.(2)). WS indicates how many jobs are executed per unit time. Assuming there is no constructive interference among applications, the maximum value of WS is equal to the number of applications. Thus, in a 2-application mix, the optimal (maximum) value of WS is 2. In the worst case, if both the applications stall the progress of each other indefinitely, WS will be equal to 0. Instruction Throughput (IT) is defined as the total number of instructions committed per cycle in the entire chip. (Eq.(3)). It basically depicts the raw machine throughput. The Fairness Index (FI) is used to express the imbalance of performance slowdowns across applications. Eq.(4) shows the FI equation for a system that executes two application concurrently. When FI is equal to 1 the system is completely fair, because all applications are slowed down equally when they execute concurrently and share the same resources.

Summary of Evaluation Metrics

(A) IPC_i is the number of committed instructions per cycle (IPC) of ith application, **(B)** IPC_i^{shared} is IPC of ith application when it is co-scheduled with other applications, **(C)** IPC_i^{alone} is IPC of ith app. when it is the only application executing on the entire GPU,

Eq.(1) $Slowdown(APP_i) = \frac{IPC_i^{shared}}{IPC_i^{alone}}$

Eq.(2) $Weighted\ Speedup = \sum_i Slowdown(APP_i)$

Eq.(3) $Instruction\ Throughput = \sum_i IPC_i$

Eq.(4) $Fairness\ Index = MAX(\frac{slowdown(APP_1)}{slowdown(APP_2)}, \frac{slowdown(APP_2)}{slowdown(APP_1)})$

Table 6.3: Evaluated 2-application GPU workloads.

Workload #	1st APP	2nd APP	Abbr.
1	HIST	GAUSS	hist_gauss
2	HIST	GUPS	hist_gups
3	HIST	BFS	hist_bfs
4	HIST	BFS	hist_bfs
5	HIST	3DS	hist_dgemm
6	GAUSS	GUPS	hist_gups
7	GAUSS	BFS	gauss_bfs
8	GAUSS	3DS	gauss_3ds
9	GAUSS	DGEMM	gauss_dgemm
10	GUPS	BFS	gups_bfs
11	GUPS	3DS	gups_3ds
12	GUPS	DGEMM	gups_dgemm
13	BFS	3DS	bfs_3ds
14	3DS	DGEMM	3ds_dgemm

6.3 Concurrent Kernel Execution Challenges

This section focuses on the memory system and highlights the main challenges associated with concurrent execution of applications. This section shows that while concurrent execution of applications can increase overall system performance, negative interactions among applications in the memory system can hamper application performance, and can introduce severe fairness problems.

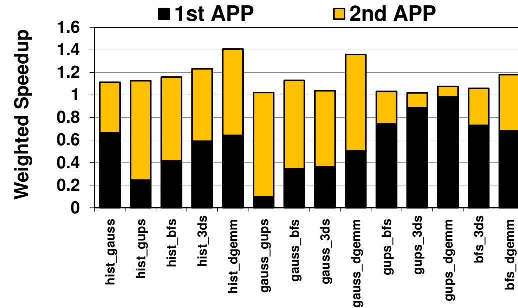


Figure 6.2: Weighted speedup (Application throughput) for the evaluated workloads. The `1st APP` and `2nd APP` are the first and second applications in the workload, respectively, as mentioned in Table 6.3.

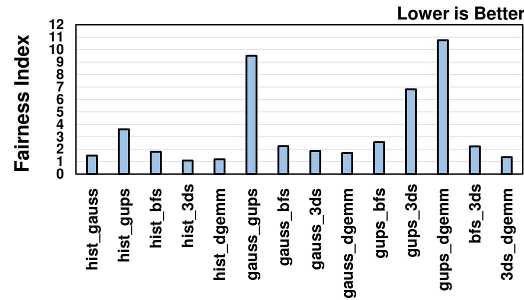


Figure 6.3: Fairness Index for the evaluated workloads when the memory scheduler adopts the baseline FR-FCFS scheduling policy.

6.3.1 Fairness considerations

Explicitly addressing fairness in the memory system is essential whenever multiple applications (potentially from different users) share the same resources. An unfair or uncoordinated resource allocation can lead to imbalance in performance degradation across applications. This phenomenon is demonstrated in Figure 6.2, where the weighted speedups obtained for all 14 evaluated workloads are plotted. The slowdown of each application in every workload is shown. `1st APP` shows the slowdown of the first application when co-scheduled with the `2nd APP`, and vice versa. In a completely fair system, the values of slowdowns (performance degradation) for each one of the two applications would be the same. Instead, it is hardly ever the case, and the slowdowns are considerably different between the two applications. For example, consider the case of `gups_dgemm`, where there is significant difference between the slowdowns

of GUPS, 1st APP, and DGEMM, 2nd APP. This means that while GUPS performance is hardly affected by sharing the GPU resource, DGEMM performance is considerably degraded. In fact, most of the contribution to WS in this case is associated with GUPS.

Figure 6.3 shows the FI metric for the evaluated workloads. While slowdowns in some workloads are relatively balanced (e.g., hist_dgemm) the FI value for other workloads is very high (e.g. gups_3ds and gauss_gups). Particularly, the FI for gups_dgemm is the worst (maximal) among all the workloads (10.75). The reason for such high FI value is the fact that GUPS has very high memory bandwidth demands compared to other co-scheduled applications. Its memory bandwidth utilization reaches 93% (see Table 6.2), which significantly degrades performance of other co-scheduled applications. As shown next, the presence of such memory bandwidth demanding applications causes imbalance in performance degradation, leading to very high FI values.

To get a deeper insight into the mechanics of the cross-application interference in the memory system, Figure 6.4 shows a break down of the memory bandwidth to the following components: (A) 1st and 2nd APP: the relative portion of DRAM cycles during which the 1st and 2nd applications in the workload move useful data over the DRAM interface, (B) Wasted-BW: the relative portion of DRAM cycles during which no data is transferred over the DRAM interface, but there are pending memory requests in memory controller. This is because of DRAM timing constraints; improving DRAM page hit rates, and bank-level parallelism can reduce this Wasted-BW, and (C) Idle-BW: the relative portion of DRAM cycles during which there are no requests pending in the memory controller queues, and hence DRAM is idle. Besides the concurrent execution configuration, the figure also plots alone_60 – where an application executes in a stand-alone mode over the entire 60 SM GPU system, and alone_30 – where an application executes in a stand-alone mode over half of the compute resources (up to 30 SMs in this case).

It is evident from Figure 6.4 that the memory intensive applications monopolize the memory scheduler while the lighter applications are unable to get a fair share of the bandwidth. For example, when GUPS is co-scheduled with other applications (HIST, GAUSS, BFS, 3DS, and DGEMM), the majority of the

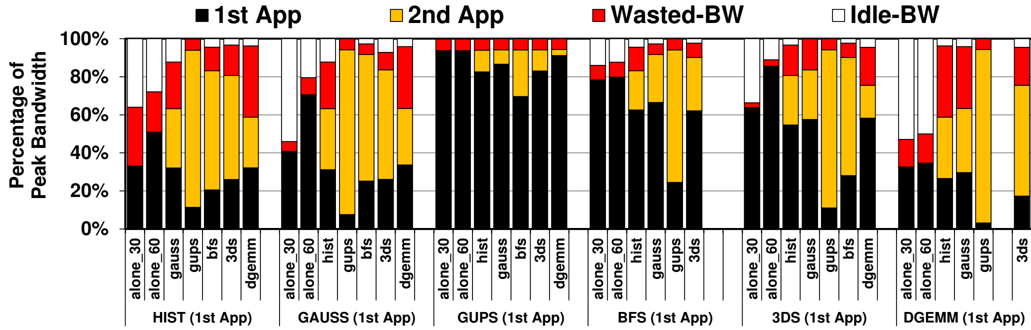


Figure 6.4: DRAM bandwidth utilization distribution across various workloads when memory scheduler adopts the baseline FR-FCFS memory scheduling policy.

bandwidth is consumed by `GUPS`, while the other application gets a very small share of the bandwidth. In the case of `gups_dgemm`, the memory bandwidth obtained by `GUPS` reduces only marginally (by 6% over alone_60 configuration), but `dgemm` achieves only 3% of the memory bandwidth (31% lower than its alone_60 configuration). This imbalanced allocation of memory resources translates to imbalance in performance degradation – GUPS slows downs by only 2% while `DGEMM` slows down by 90%, when `GUPS` and `DGEMM` are coupled together. Overall, these observations indicate that one of the main reasons for poor fairness is the interference caused by applications with intensive bandwidth requirements.

6.3.2 Throughput considerations

One of the primary motivations for preferring concurrent execution of multiple applications over time division multiplexing of the GPU hardware is to increase the machine utilization and thus improve application throughput. Applications throughput is reflected by weighted speedup and is shown in Figure 6.2. Indeed, the achieved WS for each one of the workloads is above one, indicating that concurrent execution performs either as good or better than a time division scheme. Some workloads present significant speedups - up to 41% for `hist_dgemm`. This is because these two applications have the lowest memory bandwidth demands in the workload suite (see Table 6.2), and hence do not interfere significantly in the memory system when co-scheduled together. However, other workloads present modest to minimal gains: in the case of

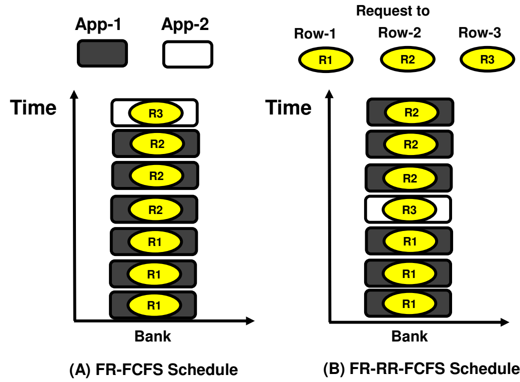


Figure 6.5: Conceptual example showing the working of (A) baseline FR-FCFS memory scheduling, (B) proposed FR-RR-FCFS memory scheduling.

`gauss_gups`, the interference is significant and is mostly caused by `GUPS`, leading to only 2% improvement in WS. In fact, most applications that are co-scheduled with `GUPS` exhibit poor WS.

The main reasons for this sub-optimal weighted-speedup behavior is in fact a manifestation of the previously discussed fairness problem, where the bandwidth-intensive application significantly degrades the performance of other applications leading to lower overall weighted speedup. For example, consider the case of `gups_dgemm` where the weighted-speedup is only 7%. `GUPS` interferes with the progress of `GAUSS` leading to its sub-optimal performance resulting in lower overall WS.

6.4 Application-Aware Memory Scheduling

The analysis in Section 6.3 shows that application-agnostic approach of the underlying memory-system scheduling policy leads to sub-optimal results both in terms of overall application throughput and fairness. Therefore, it is imperative to develop an application-aware memory scheduling approach to address these issues. To this end, this section proposes and discusses the details of an example design of a simple application-aware memory scheduler that improves fairness and performance.

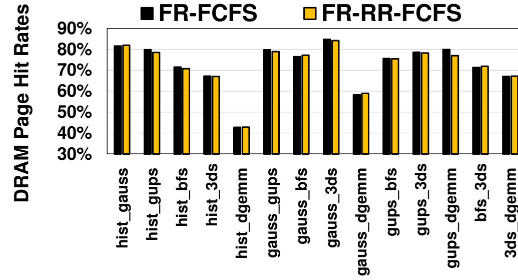


Figure 6.6: Effect on DRAM page hit rates. The proposed scheduler FR-RR-FCFS preserves the DRAM page hit rates obtained by the baseline FR-FCFS memory scheduler.

6.4.1 Designing Application-aware Memory Scheduler

The bandwidth-intensive applications (e.g. `GUPS`) can severely degrade the performance of its co-scheduled applications, leading to sub-optimal application performance and fairness. To address these issues, this work proposes an example implementation of a simple memory scheduler. This work proposes to equip the widely known FR-FCFS memory scheduler with application awareness by choosing requests from different applications in round-robin (RR) fashion. The advantage of this memory scheduling approach is that the bandwidth-intensive application would not be able to starve its co-scheduled applications for a long period time. Note that this scheme still prioritizes the row-hit requests over other requests for optimizing DRAM page hit rates. The only difference in this memory scheduler is that the requests are not picked in FCFS fashion, but in RR fashion across applications. If the pending memory controller queue has only requests from either one of the applications, the RR automatically performs FCFS.

Formally, this work proposes First-ready Round-robin FCFS (FR-RR-FCFS) memory scheduling method for handling memory requests from multiple GPU applications. The request prioritization order of FR-RR-FCFS is: 1) row-buffer-hit requests over all other requests, 2) requests from the application next in the round-robin scheduling order, 3) older requests over younger ones. Among row-hit requests, older requests are prioritized over younger requests.

Figure 6.5 shows the mechanics of the proposed FR-RR-FCFS memory scheduling function, for multiple concurrent GPU applications. Figure 6.5 (A)

shows the baseline scheduler with FR-FCFS memory scheduling function. Without the loss of generality, in this example, let us assume one memory bank and one-memory controller memory system. Furthermore, let us assume that two applications, App-1 and App-2, are concurrently executing on the same GPU platform. The memory requests are tagged with their application-id (in this example, they are color coded – App-1 (gray) and App-2 (white)). In Figure 6.5 (A), the first memory request arriving to the memory controller is originating from App-1, and destined to row-1 (R1). Similarly, the next two requests also belong to R1, and originate from App-1. Because of the FR-FCFS policy, the baseline memory scheduler will schedule all these three requests back-to-back. The next three requests are also originated from App-1 (also gray coded), but are destined to R2. As the memory scheduler is application-agnostic, it will keep on scheduling those requests in their arrival order, and the request from App-2 (the last request – white coded, destined to R3) would be delayed significantly in that case. Alternatively, FR-RR-FCFS memory scheduler in Figure 6.5 (B), would service App-2 request after servicing first three App-1 requests destined to R1. In the proposed policy the waiting time of memory requests from App-2 is substantially shorter, and thus it will not be starved by requests from App-1.

One might argue that after servicing the first memory request of App-1, memory scheduler should shift to App-2 for the natural round-robin sequence. However, by doing so, the scheduler would have to switch the memory-row (from R1 to R3) and back to R1 (R3 to R1). These row-switches would have degraded DRAM page hit-rates and throughput. In order to preserve DRAM page-hit rate, this scheme first services all the memory requests to the same page, and then moves to the next application in round-robin fashion. Figure 6.6 shows that the DRAM page hit-rates for FR-FCFS and the proposed scheduler are roughly the same (average reduction is less than 1%).

6.4.2 Hardware Complexity

The proposed method is relatively simple to implement in hardware, and that it would require a very low additional hardware cost compared to an existing

scheduling logic.

In order to propagate application-related information throughout the memory-system, the memory request need to be tagged with the application-id information. The tagging is performed at the SM-level. For a limited number of concurrent applications on a single GPU, we assume several bits per memory request. For the example discussed in this chapter, of up to 2 applications per GPU, single bit is needed for application-id extension of the request meta-data fields.

The additional hardware required for the RR function in the memory controller is minimal compared to an existing FR-FCFS logic. It requires a duplication of the find-first masking logic according to the application ID, similar to what is done for finding the first ready request for an open-row in the memory controller already. In addition, it is required to compute the next-application ID in a RR fashion, which can be implemented by a simple rotating function. Note that all the required information is computed locally at the memory controller, and no communication/coordination across memory controllers, and banks within memory controller is required.

6.5 Experimental Results

This section provides a comparative analysis of the evaluated schedulers in terms of fairness and performance.

6.5.1 Fairness Results

Figure 6.7 shows the fairness index (FI) for all the evaluated workloads, both for the baseline memory scheduler (FR-FCFS), and for the proposed FR-RR-FCFS policy. This work observes significant improvements in fairness (decrease in FI) with FR-RR-FCFS: 49% for `hist_gups`, 47% for `gups_3ds`, 14% for `gauss_bfs`, and 11% for `hist_bfs`. On average, this work observes 7% improvement in fairness over the baseline FR-FCFS policy. To understand these benefits better, Figure 6.4 is re-plotted in Figure 6.8, but the bandwidth distribution of the baseline FR-

FCFS is compared with the distribution achieved when using proposed FR-RR-FCFS scheduling policy. For clarity, those workloads are omitted that do not have significant difference in these distributions. It is observed that improvement in FI has originated from a fairer distribution of the overall memory bandwidth across the concurrently executing kernels. FR-RR-FCFS provides more memory bandwidth to the *lighter* applications (compared to the baseline), and thus limits their performance degradation. For example, when `HIST` is coupled with `GUPS`, the bandwidth obtained by `HIST` is increased from 10% (fr-fcfs-gups) to 20% (fr-rr-fcfs-gups). Note that, ideally `HIST` should reach up to 33% and 50%, when it executes alone on 30 and 60 SMs system, respectively. The proposed scheduler has facilitated in bridging this gap.

6.5.2 Performance Results

Figure 6.9 and Figure 6.10 show the improvement in instruction throughput (IT) and weighted-speedup (WS) when using the FR-RR-FCFS memory scheduler, respectively. Results are normalized to the baseline FR-FCFS scheduler. This work observes significant improvement in IT and WS for workloads `hist_gups`, `hist_bfs`, `gauss_gups`, and `gups_3ds`. The maximum improvement is observed in `hist_gups`: 64% in instruction throughput and 7% in weighted speedup. These improvements in performance are primarily due to the fairer allocation of memory bandwidth (see Figure 6.8). FR-RR-FCFS facilitates the lighter applications and thus reduces their performance degradation. It is evident that the high memory demanding applications `GUPS` and `BFS` are now comparatively less dominant, thereby improving performance.

It is noted that in two cases (`hist_gauss` and `hist_3ds`), there is a small decrease (2-3%) in WS when using FR-RR-FCFS. Clearly, in these workloads, the FR-RR-FCFS scheduler is unable to intelligently allocate memory bandwidth among the applications. Indeed, while "round-robin" gives better opportunity to all concurrent applications in taking service from memory, it is still unaware of individual application's *characteristics*. A more sophisticated scheme might use application characterization to influence the priority settings in attaining service

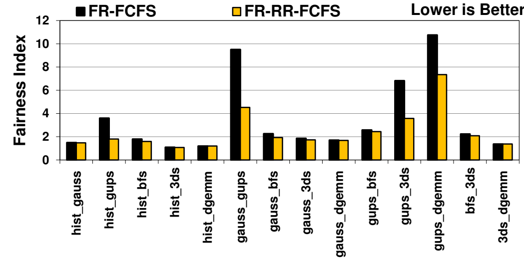


Figure 6.7: Fairness index (FI) of the evaluated workloads when memory scheduler adopts FR-FCFS (baseline, 1st bar) and FR-RR-FCFS (proposed, 2nd bar) scheduling techniques.

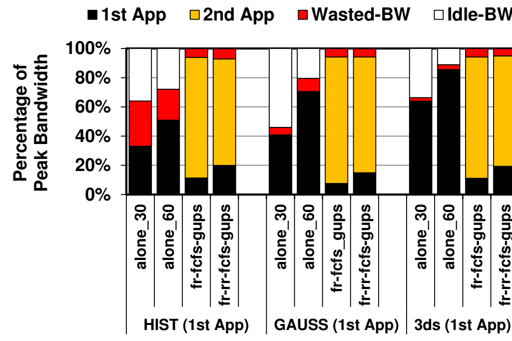


Figure 6.8: DRAM bandwidth utilization distribution across selected workloads when memory scheduler adopts FR-FCFS (baseline, 3rd bar) and FR-RR-FCFS (proposed, 4th bar) scheduling techniques.

from memory.

6.6 Related Work

This work provides a detailed analysis on the interactions of multiple applications in GPU memory system, and proposes a memory scheduler to improve both fairness and overall performance.

Memory scheduling techniques: There is a large body of work on memory scheduling techniques in the context of multi-cores. Thread cluster memory scheduling (TCM) [45] classified applications on the basis of their sensitivity to memory bandwidth and latency. They further proposed various memory request prioritization schemes for improving fairness and throughput. However, their

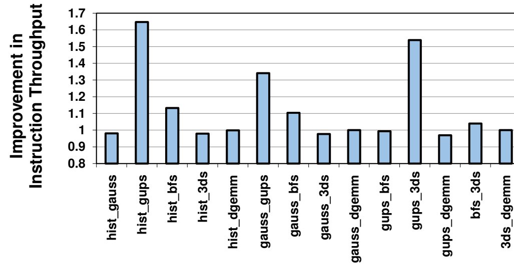


Figure 6.9: Improvement in instruction throughput (IT) across the evaluated workloads. Results are normalized to the case when memory scheduler adopts the baseline FR-FCFS scheduling policy.

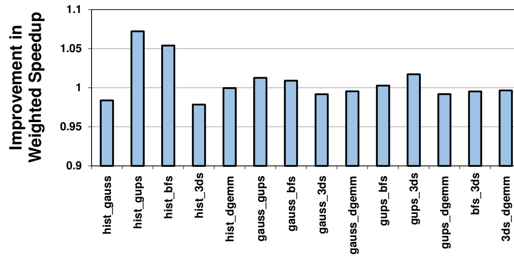


Figure 6.10: Improvement in weighted speedup (WS) across the evaluated workloads. Results are normalized to the case when memory scheduler adopts the baseline FR-FCFS scheduling policy.

work only considers multiple single-programmed applications. On the other hand, this work focuses on multiple massively threaded applications, and proposes a simple but effective memory scheduling technique to handle their memory requests. Ebrahimi et al. [93] proposed parallel application memory scheduling, where they explicitly managed inter-thread memory interference for improving performance, especially in critical sections of the program. However, their work only considers a single multi-threaded application, while this work deals with scheduling of multiple multi-threaded applications.

In the context of GPUs, Lakshminarayana et al. [50] explored a DRAM scheduling policy that essentially chooses between Shortest Job First (SJF) and FR-FCFS [33, 34]. Yuan et al. [49] proposed an arbitration mechanism in the interconnection network to restore the lost row buffer locality caused by the interleaving of requests. They showed that performance of in-order DRAM memory scheduler can be competitive to FR-FCFS, if interconnect is aware of the requests destined to the same row. Ausavarungnirun et al. [51] proposed a

staged memory scheduler for CPU-GPU architectures. Their primary goal was to improve row-buffer locality in heterogeneous architectures. All these works only focus on improving the performance of single GPU application and do not focus on the scenarios when multiple applications are scheduled concurrently, as done in this work.

Concurrent execution of multiple kernels on GPUs: Pai et al. [105] proposed elastic kernels that allow a fine-grained control over their resource usage. Further, they proposed elastic-kernel aware concurrency management policies for improving GPU performance. Adriaens et al. [108] proposed spatial partitioning of SM resources across concurrent applications. They presented a variety of heuristics for dividing the SM resources across applications. This work assumes an even partitioning technique, according to which SMs are distributed evenly among concurrent applications. Gregg et al. [109] presented KernelMerge, a runtime framework to understand and investigate concurrency issues for OpenCL applications. Wang et al. [110] proposed context funneling, which allows kernels from different programs to execute concurrently. None of these works directly addressed the problem of contention caused by multiple concurrently executing kernels in the memory system.

Warp scheduling in GPUs: Narasiman et al. [8] proposed two-level warp scheduler that splits the concurrently executing warps into groups to improve memory latency tolerance. Rogers et al. [67] proposed cache-conscious wavefront scheduling to improve the caching efficiency in GPUs. Gebhart and Johnson et al. [48] proposed a two-level warp scheduling technique that focuses on reducing the energy consumption in GPUs. Jog et al. [7] proposed a series of CTA-aware warp scheduling techniques to reduce cache and memory contention. Kayiran et al. [9] modulated the available thread-level parallelism by intelligent CTA scheduling. Jog et al. [27] proposed prefetch-aware warp scheduling techniques for enhancing GPGPU performance. All these warp scheduling schemes are developed for the scenario when only one kernel is executing at a time. It is not clear how these techniques will perform when multiple kernels are scheduled concurrently. However, this work does not design smart warp scheduling techniques for such scenarios, but do believe that it is an open research issue.

6.7 Chapter Summary

GPUs are expected to support concurrent execution of multiple kernels – either from the same application or from multiple applications. While this computing paradigm can improve machine utilization when executing applications with limited scalability, the complexity of marshaling multiple kernels introduces key architectural challenges. This chapter zoomed in on the memory system and showed that the interactions among memory streams of concurrently executing applications can lead to severe unfairness and sub-optimal performance. Furthermore, this chapter showed that the primary reason for these problems is the application-agnostic management of shared resources. For example, the memory scheduler refers to all memory requests as a single request stream and focuses solely on improving the overall DRAM page hit rates.

This chapter argues that in order to overcome these problems, application awareness must be propagated to the memory system. To this end, this chapter proposed a simple augmentation to the current memory system scheduler that schedules memory requests from different applications in a round-robin manner that not only preserves DRAM page hit rates, but also makes sure that co-scheduled memory-intensive applications do not starve other applications for long intervals. Detailed simulation results show that the proposed scheduler delivers superior performance and improves fairness across a wide set of workloads.

Chapter 7

Anatomy of Multi-Application Execution in GPUs

As GPUs make headway in the computing landscape spanning mobile platforms, supercomputers, cloud and virtual desktop platforms, supporting concurrent execution of multiple applications in GPUs becomes essential for unlocking their full potential. However, unlike CPUs, multi-application execution in GPUs is little explored. This chapter studies the memory system of GPUs in a concurrently executing multi-application environment. An analytical performance model is first presented for many-threaded architectures to show that the common use of misses-per-kilo-instruction (MPKI) as a proxy for performance is not accurate without considering the bandwidth usage of applications. Second, this chapter characterizes the memory interference of applications and discuss the limitations of existing memory schedulers in mitigating this interference. Third, the analytical model is extended for multiple applications to identify the key metrics for controlling the various performance metrics. Extensive simulation results using an enhanced version of GPGPU-Sim targeted for concurrently executing multiple applications show that memory scheduling decisions based on MPKI and bandwidth information are more effective in enhancing system throughput compared to the traditional FR-FCFS and the previously proposed RR FR-FCFS policies.

7.1 Introduction

The computing trajectory of GPUs has evolved from traditional graphics rendering to accelerating general purpose and high performance computing applications, and of late to cloud and virtual desktop computing. Two trends have driven this trajectory. First, GPU resources are rapidly growing with each technology generation [103, 104, 111] to provide the workhorse for efficient computation. Second, advances in software and virtualization technology for GPUs such as NVIDIA GRID [106], and OpenStack IaaS framework have made the transition possible.

GPU virtualization is required for concurrent access to the GPU resources by multiple applications, potentially originating from different users. This can be facilitated by spatial as well as temporal allocation of GPU resources. The current NVIDIA GRID [106] and other cloud providers support virtualization by time multiplexing. Spatial resource sharing has yet to evolve because unlike CPUs, traditionally, GPUs were designed to execute only a *single* application at a time. However, it has been shown recently that only executing a single application at a time may not effectively utilize the available computing resources in state-of-the-art GPUs [105], thereby making a compelling case for multi-application execution [28, 105]. Thus, supporting multi-application execution is essential both from performance and utility (adoption in cloud environments) perspectives.

Unlike CPU-based architectures, where resource allocation and scheduling for multiple application execution has been studied extensively, only a few prior works (e.g., [28, 105, 108–110]) have scratched the issues related to multiple application execution in the context of GPUs. Among them only a few works [28] have addressed the problem of multi-application interference in the GPU memory system. Therefore, many of the design issues are still little understood. This chapter focuses on the interactions of multiple applications in GPU memory system, and specifically attempt to answer the following questions: (i) *How do we characterize the interactions between multiple applications in the GPU memory system?* (ii) *What are the limitations of traditional application-agnostic*

FR-FCFS [32–34] and RR FR-FCFS scheduling [28] in the context of throughput oriented GPU platforms?, (iii) *Is it possible to push the performance envelope further with an efficient scheduling mechanism?*, and (iv) *How do we explore the design space and develop analytical performance models to find appropriate knobs for guiding the scheduling decisions?* In this context, this chapter makes the following **contributions**:

• Contrary to the common use of misses-per-kilo-instruction (MPKI) as a metric for gauging the memory intensity, and as a proxy for application performance, this work shows that a model based on **both** MPKI and the achieved DRAM bandwidth information is able to gauge the memory intensity and estimate the performance of GPU applications more accurately.

• This work performs a detailed analysis of application characteristics and interactions among multiple applications in GPUs to classify applications and understand the scheduling design space based on this classification.

• This work develops a simple analytical model to demonstrate that L2-MPKI and bandwidth utilization can be used as two control knobs to optimize performance metrics: instruction throughput (IT) and weighted speedup (WS), respectively. Based on this analytical model, this work develops two memory scheduling schemes, ITS and WEIS, which are customized to improve IT and WS, respectively. This work shows that the proposed solutions are still effective with different core partitioning configurations and scalable for running up to three applications concurrently.

• This work qualitatively and quantitatively compares the proposed schemes with the traditional FR-FCFS and the recently proposed round-robin RR FR-FCFS [28] schedulers. Across 25 representative workloads, ITS improves IT by 34% and 8% over FR-FCFS and RR FR-FCFS, respectively; and WEIS improves WS by 10% and 5% over FR-FCFS and RR FR-FCFS, respectively.

• This work conducts an in-depth evaluation on GPU memory systems in multi-application environment. In this context, this work develops a GPU Concurrent Application suite (GCA) and a parallel workload simulation framework by extending GPGPU-Sim [31], a cycle accurate GPU simulator.

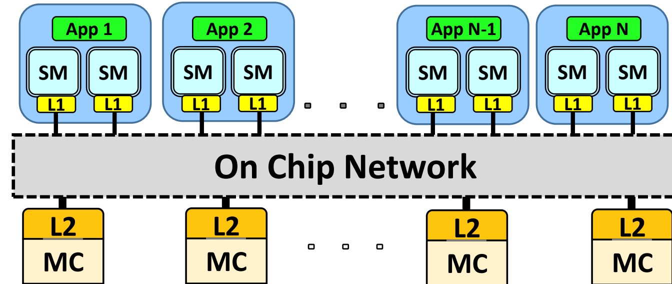


Figure 7.1: Overview of the baseline architecture capable of executing multiple applications.

7.2 Background

7.2.1 Baseline Architecture

This chapter considers a generic NVIDIA-like GPU as the baseline, where multiple cores, also called as streaming multi-processors (SMs)[1], are connected to multiple memory controllers (MCs) via an on-chip network as shown in Figure 7.1. Each MC is a part of the memory partition that also contains a slice of L2 cache for faster data access. The details of the baseline configuration are shown later in Table 7.2.

Single Application Scheduling: CUDA uses computational kernels to take advantage of parallel regions in the application. Each application may include multiple kernels. GPUs execute all kernels of an application sequentially, i.e. one kernel at a time. Each kernel is organized as blocks of cooperative thread arrays (CTAs) that are executed in a parallel fashion on the whole GPU. During kernel launch, the CTA scheduler initiates scheduling of the CTAs related to that kernel, and tries to distribute them evenly [31].

Multiple Application Scheduling: This work simultaneously executes kernels from *different* applications. This work uses a kernel-to-SMs allocation scheme where the SMs are distributed evenly in a spatial manner based on the number of applications as shown in Figure 7.1. For example, if the GPU consists of 30 SMs

[1]This work uses "core" and "SM" terms interchangeably.

that need to be partitioned among two applications, the first 15 SMs are assigned to the first application, and the rest of the SMs to the second application. As the focus of this work is memory (not caches), unless otherwise specified, SMs and L2 cache are equally partitioned across concurrently executing applications. This work also evaluates the schemes with two different SM-partitioning techniques (i.e. 10-20 and 20-10) to demonstrate the robustness of the model with respect to core partitioning in Section 7.8.4.

Memory Scheduling: The most widely used memory scheduler in GPUs is first-ready FCFS (FR-FCFS) [32–34], which is implemented in the hardware. This scheme is targeted at improving DRAM row hit rates, so request prioritization order is: 1) row-hit requests are prioritized over other requests; then 2) older requests are prioritized over younger ones.

7.2.2 Evaluation Metrics and Application Suite

Typically, performance of single application is captured by its *instruction throughput (IT)*. When multiple applications execute concurrently, instruction throughput measures the raw machine throughput and is given by $IT = \sum_{i=1}^{N} IPC_i$, where there are N co-running applications, and IPC_i is the number of committed instructions per cycle of the ith application. Note that this metric only considers IPC throughput, without taking fairness into account. This work focuses on this metric to evaluate pure machine performance of GPUs without considering the fairness aspect.

For evaluating system throughput, this work uses *Weighted speedup (WS)*, which indicates how many jobs are executed per unit time: $WS = \sum_{i=1}^{N} SD_i$, where SD_i is the slowdown of ith application given by $SD_i = \frac{IPC_i}{IPC_i^{alone}}$, where IPC_i^{alone} is IPC of ith application when running alone. Assuming there is no constructive interference among applications, the maximum value of WS is equal to the number of applications.

This work also shows the impact of the proposed schemes on *Harmonic Speedup (HS)* that not only measures system performance, but also has a notion

of fairness [45] and is given by, $HS = 1/(\sum_{i=1}^{N} \frac{1}{SD_i})$. Also, the Average Normalized Turn-around Time (ANTT) metric is the reciprocal of HS.

Application Suite: For experimental evaluations, this work uses a wide range of GPGPU applications implemented in CUDA. These applications are chosen from Rodinia [42], Parboil [43], CUDA SDK [31], and SHOC [91], and are listed in Table 7.1. In total, 25 applications are studied.

7.3 Performance Characterization of Many-threaded Architectures

This section revisits a performance model for many-threaded architectures proposed by Guz et al. [84], and classify the applications based on this model.

7.3.1 A Model for Many-threaded Architectures

Recent works have shown that bandwidth is usually the critical bottleneck in many-threaded architectures like GPUs [7, 9, 27]. The considered model [84] shows that performance of many-threaded architectures is directly proportional to the bandwidth that the application receives from DRAM (attained DRAM bandwidth). In this model, application performance is expressed as,

$$P = \frac{BW}{b_{reg} \times r_m \times L_{miss}} \tag{7.1}$$

where

P = performance [Operations/second (Ops)]

BW= attained DRAM bandwidth [Bytes/second (Bps)]

r_m = the ratio of the number of memory instructions to the number of total instructions

b_{reg} = the operand size [Bytes]

L_{miss}= cache miss rate

This generic model assumes a single memory space, a single-level cache and DRAM. As a result, the miss rate is used for quantifying DRAM accesses. In addition, it ignores the case of multiple memory spaces and the case of scratch-pad usage; e.g. the case where scratch-pad can provide an additional source of bandwidth. Operand size represents the amount of data fetched to/from DRAM on each cache miss, which is equal to the cache line size.

In this model, as the numerator in (7.1) is BW, performance is directly proportional to the bandwidth attained by the application. The denominator is expressed as,

$$\begin{aligned} b_{reg} \times r_m \times L_{miss} &= b_{reg} \times \tfrac{i_{mem}}{i_{tot}} \times \tfrac{c_{miss}}{c_{tot}} \\ &= b_{reg} \times \underbrace{\frac{c_{miss}}{i_{tot}}}_{\text{MPI}} \times \underbrace{\frac{i_{mem}}{c_{tot}}}_{1} \\ &= \underbrace{b_{reg} \times MPI}_{\substack{\text{Bytes required to transfer from DRAM} \\ \text{for commiting an instruction}}} \end{aligned} \tag{7.2}$$

where

i_{mem} = the number of memory instructions

i_{tot} = the number of total instructions

c_{miss} = the number of cache misses

c_{tot} = the number of cache accesses

MPI = the number of cache misses per instruction

Thus, from (7.1) and (7.2), this work notes that performance is directly proportional to the achieved DRAM bandwidth, and is inversely proportional to the size of datum that needs to be fetched to/from DRAM in order to commit an instruction. Since this work considers a system with two levels of cache with a constant cache-line size, the overall performance becomes inversely proportional to the number of L2 misses that need to be served for committing one thousand instructions (L2 misses per kilo-instruction (L2MPKI)), as given in (7.3). This chapter uses the term MPKI to represent L2MPKI.

$$P \propto \frac{BW}{MPKI} \tag{7.3}$$

This work validates this model by simulating applications from Table 7.1 using GPGPU-Sim simulator. Figure 7.2 shows the observed IPC from the simulator and the calculated IPC values by using Equation (7.3) and simulated BW and $MPKI$ values. For all 25 applications, the mean absolute relative error (MARE) is 10.3%. This work observes that for many (21 out of 25) applications, the MARE is only 4.2%. However, for other 4 applications: `MM`, `HS`, `BP`, and `SAD`, MARE is higher (42.1%), because these applications make extensive use of the software-managed scratchpad memory. As a result, their performance is not only dependent on the achieved DRAM bandwidth, but is also driven by the additional scratchpad bandwidth, which is not captured in the proposed model. For cases that involve usage of a scratch-pad memory, the model underestimates the actual IPC obtained by the application. This work conducted a similar analysis on real GPU hardware[2], and Figure 7.3 shows the absolute relative error for each application. The results for `GUPS`, `MUM` and `QTC` are omitted because they could not execute them faithfully on real hardware. For 22 applications, the observed value of MARE to be 9.5%.

7.3.2 Application Characterization

Several previous works (e.g., [45, 46, 112–114]) have 1) characterized the memory intensity of applications primarily based on their last-level cache MPKI values, and 2) used MPKI as a proxy for performance. This work shows that considering only MPKI is not enough for the both cases in GPUs.

Table 7.1 summarizes $MPKI$ and the ratio between attained DRAM bandwidth and peak bandwidth (BW/C) for the applications, where C is the peak memory bandwidth. They are listed in the descending order of their $MPKI$. First, Table 7.1 shows that only considering $MPKI$ values is not sufficient for estimating the memory intensity of a particular application. It is evident that high $MPKI$ levels may not necessarily lead to a very high DRAM bandwidth utilization, as it is also a function of the inherent compute to

[2]NVIDIA K20m, CUDA capability 3.5, Driver 6.0

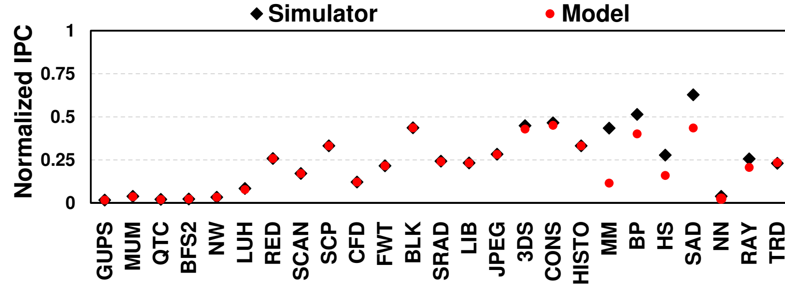


Figure 7.2: Application performance obtained via simulation and the model. IPC is normalized with respect to the maximum achievable IPC supported by the architecture.

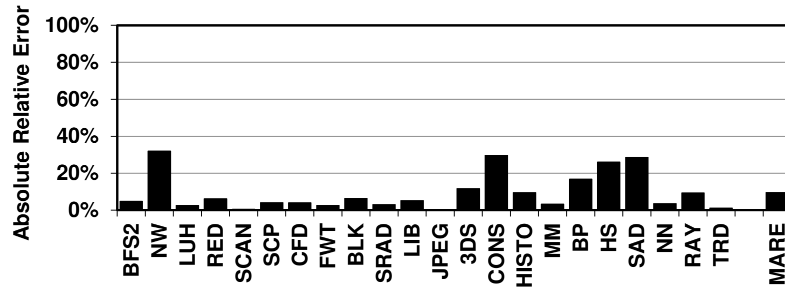


Figure 7.3: Absolute relative error between IPCs obtained from real hardware (NVIDIA Kepler K20m) and the model.

bandwidth ratio of a particular application. For example, although `QTC` has the third highest $MPKI$ among the applications, there are 15 other applications in the suite that have higher DRAM bandwidth utilization than `QTC`.

Second, Section 7.3.1 showed that performance is not only dependent on $MPKI$, but also BW. For the applications shown in Table 7.1, the correlation between $MPKI$ and IPC is only -44.4%. While $MPKI$ is the number of misses that needs to be served to commit 1000 instructions, it lacks the information about the *rate* at which these misses are served by DRAM. In other words, L2-MPKI is a good measure of the bandwidth demand of the application, but performance is a function of *both the demanded and the achieved bandwidth.*

7.4 Analyzing Memory System Interference

This section builds a foundation and draw key observations towards designing a more efficient, application-aware memory scheduler for GPUs. We start with

Table 7.1: Application characteristics: *(A) MPKI*: L2 cache misses per kilo-instructions. *(B) BW/C*: The ratio of attained bandwidth to the peak bandwidth of the system.

GPGPU Application	Abbr.	MPKI	BW/C (in %)
Random Access	GUPS	57.1	93.0
MUMmerGPU [23]	MUM	22.6	79.7
Quality Threshold Clustering [91]	QTC	7.9	15.3
Breadth-First Search [23]	BFS2	5.3	11.6
Needleman-Wunsch [42]	NW	5.1	16.7
Lulesh [90]	LUH	4.7	35.6
Reduction [91]	RED	2.8	68.8
Exclusive [91] Parallel Prefix Sum	SCAN	2.7	45.0
Scalar Product [23]	SCP	2.7	86.3
Fluid Dynamics [42]	CFD	2.3	27.2
Fast Walsh Transform [23]	FWT	2.2	45.1
BlackScholes	BLK	1.6	67.6
Speckle Reducing Anisotropic Diffusion [42]	SRAD	1.6	36.4
LIBOR Monte Carlo [23]	LIB	1.1	25.4
JPEG Decoding	JPEG	1.1	29.8
3D Stencil	3DS	1.0	42.3
Convolution Separable [23]	CONS	1.0	43.5
2D Histogram [43]	HISTO	0.6	18.8
Matrix Multiplication [43]	MM	0.5	5.1
Backpropogation [42]	BP	0.4	16.5
Hotspot [42]	HS	0.4	6.1
Sum of Absolute Differences [43]	SAD	0.1	5.6
Neural Networks [23]	NN	0.1	0.3
Ray Tracing [23]	RAY	0.1	1.8
Stream Triad [91]	TRD	0.1	1.4

presenting the nature of interference among concurrent applications in GPU memory system, then we discuss the inefficiencies of existing memory schedulers, and finally we draw initial insights for designing a better memory scheduling technique.

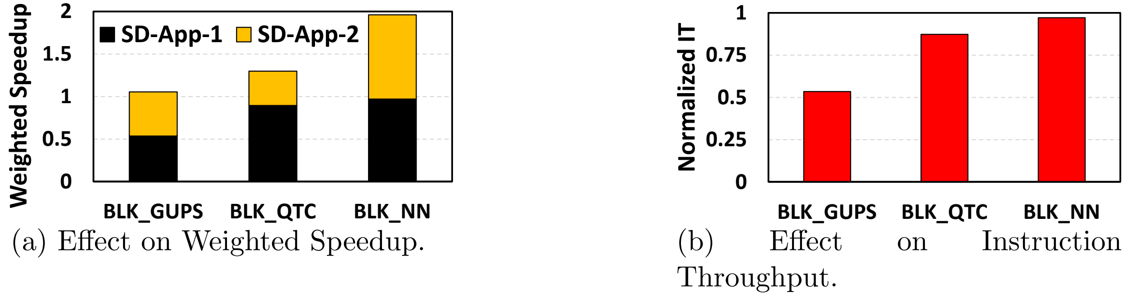
7.4.1 The Problem: Application Interference

When multiple applications are co-scheduled on the same GPU hardware, they interfere at various levels of the memory hierarchy, such as interconnect, caches and memory. As memory system is the critical bottleneck for a large number of GPU applications, explicitly addressing contention issues in the memory system is essential. We find that an uncoordinated allocation of GPU resources, especially memory system resources, can lead to significant performance degradations both in terms of IT and WS. To demonstrate this, consider Figure 7.4, which shows the

impact on WS and IT, when BLK is co-scheduled with three different applications (GUPS, QTC, NN). The workloads formed are denoted by BLK_GUPS, BLK_QTC and BLK_NN, respectively. Let us first consider the impact on WS in Figure 7.4a, where we also show the breakdown of WS in terms of slowdowns (SD) experienced by individual applications. The slowdowns of the applications in the workload are denoted by SD-App-1 and SD-App-2 for the first and the second applications, respectively. Note that when the applications do not interfere with each other, both SD-App-1 and SD-App-2 are equal to 1, leading to a weighted speedup of 2. This figure demonstrates three different memory interference scenarios: (1) in BLK_GUPS, both applications slow down each other significantly, (2) in BLK_QTC, slowdowns of BLK and QTC are very different – slowdown of QTC being much higher than that of BLK, and (3) in BLK_NN, slowdown in both applications is negligible. In these different scenarios, the degradation in WS is also very different. This work observes significant degradation in WS in the first (90%) and second (54%) cases, whereas, it is negligible (2%) in the third case.

Figure 7.4b shows IT degradation when BLK is co-scheduled with the same three applications vs. the case where BLK and other application in the workload are executed sequentially. This work observes similar interference trends in BLK_GUPS and BLK_NN, where in the former case, we observe significant degradation in IT, while the latter exhibits negligible degradation. Interestingly, the IT degradation in BLK_QTC is not as significant as compared to its WS degradation. From this discussion, we conclude that *concurrently executing applications are not only susceptible to significant levels of destructive interference in memory, but also the degree with which they impact each other can be strongly dependent on the considered performance metric.*

Analysis: As shown in Section 7.3, performance of a GPU application is a function of both BW and $MPKI$. As we seek insights on the impact of each metric on performance, let us first discuss the memory bandwidth component. For example, since both BLK and GUPS have very high bandwidth demands (Table 7.1), when they are co-scheduled, performance of both applications degrade significantly. We observe exactly the same behaviour in Figure 7.4a, where both applications do not receive their bandwidth shares for achieving



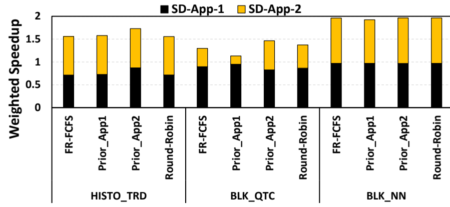
(a) Effect on Weighted Speedup.

(b) Effect on Instruction Throughput.

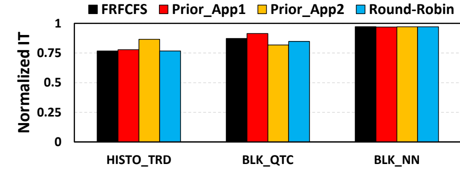
Figure 7.4: Different performance slowdowns obtained when BLK is co-scheduled with three different applications: `GUPS`, `QTC`, and `NN`. Memory scheduling policy is FR-FCFS.

stand-alone performance levels. Similarly, the available DRAM bandwidth is not sufficient for the `BLK_QTC` case, and `BLK` hurts `QTC` performance quite significantly by getting most of the available bandwidth. In the third case with `BLK_NN`, the bandwidth is sufficient for both applications, resulting in negligible performance slowdowns for both. This infers the fact that limited DRAM bandwidth is one of the important reasons of application interference and different application slowdowns.

However, considering only BW of each application would not explain why `BLK_QTC` experiences only a slight degradation in IT. We also need to consider the second parameter that affects performance, $MPKI$, whose values are listed in Table 7.1. In this example, there is significant difference in $MPKI$ of `BLK` and `QTC` (`BLK` $MPKI$ ¡ `QTC` $MPKI$). From (7.3), we know that the application with low BW and high $MPKI$ will achieve lower performance levels than the applications with high BW and low $MPKI$. Therefore, in `BLK_QTC`, `BLK` contributes much more towards higher IT than `QTC`. As `BLK` does not slow down significantly (Figure 7.4a), the total IT reduction is low. This discussion confirms that *both BW and $MPKI$ are key parameters for better understanding of the performance characteristics and scheduling considerations for concurrently executing applications in GPUs.*



(a) Effect on Weighted Speedup.



(b) Effect on Instruction Throughput

Figure 7.5: Different performance slowdowns experienced when different memory scheduling schemes are employed.

7.4.2 Limitations of Existing Memory Schedulers

This section discusses the limitations of three memory schedulers. This work considers the baseline FR-FCFS [32–34] scheduler that targets improving DRAM row hit rates, and prioritizes row-hit requests over any other request. In addition, this work explores two other schemes a) Prior_App scheduler that *statically* prioritizes requests of only one of the co-scheduled applications, and b) the recently proposed round-robin (RR) FR-FCFS GPU memory scheduler [28] that gives equal priority to all concurrently executed applications in the system without considering their properties. None of these schedulers sacrifice locality, but instead of picking memory requests in FCFS order after servicing row-hit requests (as done in FR-FCFS), Prior_App_i always prefers memory request from i^{th} application, and Round-Robin arbitrates between applications in the round-robin order. Figure 7.5 shows the effect of these three memory schedulers on weighted speedup and instruction throughput for two of the workloads already shown in Figure 7.4a (`BLK_QTC` and `BLK_NN`), and an additional workload `HISTO_TRD`.

Limitations of FR-FCFS: When multiple applications are co-scheduled, FR-FCFS still optimizes for row-hit locality and does not consider the individual application properties while making scheduling decisions. Because of such application-unawareness, FCFS nature of the scheduler would allow a high memory demanding application to get a larger bandwidth share, as that application would introduce more requests in memory controller queue. Therefore, as shown in Figure 7.5, in `BLK_QTC`, this work observes that `BLK` gets

higher bandwidth share, causing large slowdowns in QTC. Moreover, Figure 7.5 demonstrates that the best performing scheduling strategy for improving either of the the performance metrics is prioritizing one of the applications throughout the entire execution.

Limitations of Prior_App_i: This work observes in HISTO_TRD that prioritizing TRD over HISTO provides the best IT and WS among all the considered scheduling strategies. However, in BLK_QTC, prioritizing one application over another does not improve both the performance metrics. In BLK_NN, since both the applications attain their uncontested bandwidth demands, prioritizing one over another does not impact performance. Even though prioritizing one application over another provides the best result, the challenge is to determine which application to prioritize. One way of doing so is to profile the workload and employ a static priority mechanism throughout the execution. However, such strategy is often hard to realize. Another mechanism can be to switch priorities between applications during run-time, which is similar to RR FR-FCFS [28].

Limitations of RR FR-FCFS: As discussed above, in order to optimize WS and IT in BLK_QTC, different applications are prioritized differently. Since RR switches priorities between these two applications, it achieves a balance between improving both metrics. However, in HISTO_TRD, although employing RR mechanism leads to a slightly better IT and WS over FR-FCFS, it is far from the case where TRD is prioritized.

Based on the above discussion, this work makes two key observations that guide us in developing an application-conscious scheduling strategy.

Observation 1: Prioritizing lower $MPKI$ ***applications improves*** IT***:*** In the workloads where significant slowdowns are observed in either of the applications, prioritizing the application with lower $MPKI$ improves IT. For example, in HISTO_TRD, TRD has lower $MPKI$ than HISTO and therefore, Prior_App_2 yields better IT. Similarly, in BLK_QTC, BLK has lower $MPKI$ than QTC, thus Prior_App_1 provides better IT.

Observation 2: Prioritizing lower BW ***applications improves*** WS***:*** In the workloads where significant slowdowns are observed in either of the

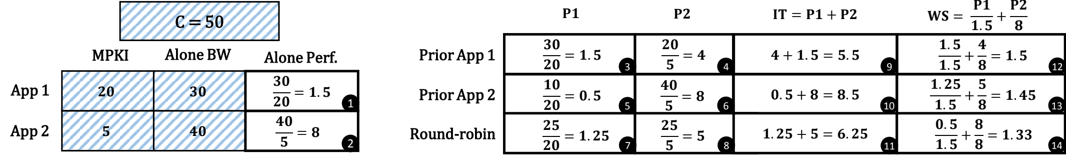


Figure 7.6: An illustrative example showing IT and WS for two applications running together. The shaded boxes represent system and application properties. The peak memory bandwidth is 50 units. Application 1 and 2 use 30 and 40 units bandwidth, respectively, when they execute alone. Their $MPKI$s are 20 and 5, respectively.

applications, prioritizing the application with lower BW improves WS. For example, in `HISTO_TRD`, `TRD` has lower BW than `HISTO` and therefore Prior_App_2 yields better WS. Similarly, in `BLK_QTC`, `QTC` has lower BW than `BLK`, thus Prior_App_2 provides better WS.

7.5 A Performance Model for Concurrently Executing Applications

In order to provide a theoretical background for the discussed observations for optimizing IT and WS in Section 7.4.2, model is extended for two applications, and it is analyzed to understand the effect of concurrent execution on the performance metrics *when memory bandwidth is the system bottleneck*. This work shows how this model guides us in developing different memory scheduling algorithms targeted for optimizing each metric separately, and explain the findings based on the model by examples.

The notations used in the model are given below:

P_i^{alone} = performance of application i when it runs alone

BW_i^{alone} = bandwidth attained by application i when it runs alone

P_i = performance of application i

BW_i = bandwidth attained by application i

C = peak bandwidth of the system

ϵ = infinitesimal bandwidth given to an application

$MPKI_i$ = $MPKI$ of application i

When applications run alone, they cannot attain more bandwidth than that is available in the system, thus $BW_i^{alone} \leq C \ \forall i$; and similarly for concurrent execution, the cumulative bandwidth consumption cannot exceed the peak bandwidth, thus $\sum_{i=1}^{N} BW_i \leq C$. Also, this work uses $P_i, BW_i, MPKI_i \geq 0 \ \forall i$.

Let us analyze the case when two applications are executed concurrently[3]. Assuming that performance is limited by the memory bandwidth, an application cannot consume more bandwidth without getting more share from the other application's bandwidth. Thus, in a two-application scenario, $BW_1 + BW_2 = C$, assuming no wastage of bandwidth is incurred.

This work assumes that, at time $t = t_0$, we have $P_i \propto \frac{BW_i}{MPKI_i} \ \forall i$; and at $t = t_1$ where $t_1 > t_0$, we give an additional ϵ bandwidth to the first application by taking it from the other. Thus, at $t = t_1$, we have $P_1' \propto \frac{BW_1+\epsilon}{MPKI_1}$, and $P_2' \propto \frac{BW_2-\epsilon}{MPKI_2}$, where P_i' is the performance of application i at $t = t_1$.

7.5.1 Analyzing Instruction Throughput

In order to have higher IT at $t = t_1$ compared to $t = t_0$,

$$P_1' + P_2' > P_1 + P_2 \tag{7.4}$$

$$\frac{BW_1 + \epsilon}{MPKI_1} + \frac{BW_2 - \epsilon}{MPKI_2} > \frac{BW_1}{MPKI_1} + \frac{BW_2}{MPKI_2} \tag{7.5}$$

Simplifying (7.5) yields,

$$\epsilon(MPKI_2 - MPKI_1) > 0 \tag{7.6}$$

$$\implies MPKI_2 > MPKI_1, \text{if } \epsilon > 0 \tag{7.7}$$

$$MPKI_1 > MPKI_2, \text{if } \epsilon < 0 \tag{7.8}$$

From (7.7) and (7.8), this work finds that *Application with lower $MPKI$ in order to optimize IT should be prioritized*

[3]The model can be easily extended to analyze more than two applications. This work shows the scalability of the model for three applications in Section 7.8.4.

Illustrative Example: Figure 7.6 shows an example of the optimal scheduling strategy for maximizing IT. This example considers three different scheduling strategies: 1) prioritizing the first application, 2) prioritizing the second application, and 3) the RR scheduler. When these applications run alone, based on their BW and $MPKI$ values, it can be said that the first and the second applications achieve performances of 1.5 (❶) and 8 units (❷), respectively, based on (7.3). If these applications are co-scheduled on the GPU, and if the first application is prioritized throughout the execution, that application will get 30 units of bandwidth, since it demands 30 units when it runs alone. This work assumes that $MPKI$ does not change significantly during different executions, as this work does not do any cache-related optimization in this work for preserving simplicity. Because the first application's $MPKI$ is 20, its performance will be 1.5 units (❸). The remaining 20 units of bandwidth will be used by the second application, as the peak bandwidth is 50 units ($C = 50$). Because its $MPKI$ is 5, its performance will be 4 units (❹). Similarly, if we prioritize the second application, it will get 40 units of bandwidth as it demands 40 units when running alone, and the first application will use the remaining 10 units. Thus, the performances of the first and the second applications will be 0.5 (❺) and 8 units (❻), respectively. Also, if the round-robin scheduler is employed, assuming both applications have similar row-buffer localities, they get the same share from the available bandwidth, which is 25 units, leading to 1.25 (❼) and 5 (❽) units of performance for the first and the second applications, respectively. Based on these individual application performance values calculated using the model, this work shows that IT (the sum of individual IPCs) of this workload is 5.5 (❾), 8.5 (❿), and 6.25 units (⓫), when we prioritize the first application, prioritize the second application, and employ RR scheduler, respectively. These results are consistent with the model, which suggests that prioritizing the application that has lower $MPKI$ would provide the best IT. Intuitively, as per (7.3), the same BW provided for the application with lower $MPKI$, which is the second application in this example, translates to higher IPC. Thus, prioritizing the application with lower $MPKI$ provides higher IT for the system.

Figure 7.5b observes that prioritizing the application with lower $MPKI$ results in higher IT for `HISTO_TRD` and `BLK_QTC`. Since the application interference in

BLK_NN is not significant, it does not benefit from prioritization.

7.5.2 Analyzing Weighted Speedup

In order to have higher WS at $t = t_1$ compared to $t = t_0$,

$$\frac{P_1'}{P_1^{alone}} + \frac{P_2'}{P_2^{alone}} > \frac{P_1}{P_1^{alone}} + \frac{P_2}{P_2^{alone}} \tag{7.9}$$

$$\frac{\frac{BW_1+\epsilon}{MPKI_1}}{\frac{BW_1^{alone}}{MPKI_1}} + \frac{\frac{BW_2-\epsilon}{MPKI_2}}{\frac{BW_2^{alone}}{MPKI_2}} > \frac{\frac{BW_1}{MPKI_1}}{\frac{BW_1^{alone}}{MPKI_1}} + \frac{\frac{BW_2}{MPKI_2}}{\frac{BW_2^{alone}}{MPKI_2}} \tag{7.10}$$

Simplifying (7.10) yields,

$$\epsilon(BW_2^{alone} - BW_1^{alone}) > 0 \tag{7.11}$$

$$\implies BW_2^{alone} > BW_1^{alone}, \text{if } \epsilon > 0 \tag{7.12}$$

$$BW_1^{alone} > BW_2^{alone}, \text{if } \epsilon < 0 \tag{7.13}$$

From (7.12) and (7.13), this work finds that *application with lower BW^{alone} should be prioritized to optimize for weighted speedup.*

Illustrative Example: We continue the example shown in Figure 7.6 with the optimal scheduling strategy for maximizing WS. We obtain $WS = 1.5$ (12), 1.33 (13), and 1.45 units (14) if we prioritize the first application, the second application, and employ RR scheduler, respectively. As discussed above, prioritizing the second application provides the best IT. However, we also observe that prioritizing the application with the lowest BW^{alone}, which is the first application in this example, yields the best WS, which is consistent with the model. Intuitively, the same amount of bandwidth provided for the application with lower BW^{alone}, the first application in this example, translates to lower performance degradation over its uncontested performance for that application. Since WS is the sum of slowdowns, prioritizing the application with lower BW^{alone} provides higher WS for the system.

We observe in Figure 7.5a that prioritizing the application with lower BW^{alone} results in higher WS for HISTO_TRD and BLK_QTC. BLK_NN is not a bandwidth limited

workload, thus, does not benefit from prioritization.

However, the problem with the approach that prioritizes the application with lower BW^{alone} is that, it is difficult to obtain BW^{alone} of an application without offline profiling, as pointed out by Subramanian et al. [115]. They also propose an approximate method to obtain alone performance of an application in a multiple-application environment during run-time. However, doing so requires halting the execution of one of the applications to approximately calculate the alone performance of the other application, which might cause drop in WS. Also, it does not completely eliminate the application interference. Furthermore, this sampling has to be done frequently to capture execution phases with completely different behaviours. Thus, instead of approximating P_i^{alone} or BW_i^{alone}, we slightly change the WS optimization condition, which leads to a solution that is much easier to implement and employ during run-time. The approximation does not use alone performance of an application; instead, compares P_i' with P_i. Mathematically, we have,

$$\frac{P_1'}{P_1} + \frac{P_2'}{P_2} > \frac{P_1}{P_1} + \frac{P_2}{P_2} \tag{7.14}$$

$$\frac{BW_1 + \epsilon}{MPKI_1}\frac{MPKI_1}{BW_1} + \frac{BW_2 - \epsilon}{MPKI_2}\frac{MPKI_2}{BW_2} > 2 \tag{7.15}$$

Simplifying (7.15) yields,

$$\epsilon(BW_2 - BW_1) > 0 \tag{7.16}$$

$$\implies BW_2 > BW_1, \text{if } \epsilon > 0 \tag{7.17}$$

$$BW_1 > BW_2, \text{if } \epsilon < 0 \tag{7.18}$$

From (7.17) and (7.18), we find that *we can prioritize the application with lower BW to improve relative weighted speedup.*

Section 7.8 later demonstrates that such approximation leads to better weighted speedup. This is also consistent with the observation made by Kim et al. [46] that preferring application with the least attained bandwidth can improve weighted speedup.

Discussion: Figure 7.6 showed that optimizing both IT and WS using the same

memory scheduling strategy might not be possible. A similar scenario showed in Figure 7.5 where `BLK_QTC` prefers a different application to be prioritized in order to achieve the best IT or WS. The key reason behind this is the properties of the applications that form the workload. In workloads, where the same application has the lower $MPKI$ and the lower BW^{alone}, that application can be prioritized to optimize both IT and WS. However, in workloads, where one application has lower $MPKI$ but the other demands less bandwidth, then the optimal scheduling strategy for improving the performance metrics is different. In such scenarios, RR achieves a good balance between IT and WS, because it gives equal priorities to both applications (assuming both applications have similar row-buffer localities). However, in scenarios where prioritizing only one application is optimal for both IT and WS, RR would be far from optimal in terms of performance.

7.6 Mechanism and Implementation Details

This work provided two key observations in Section 7.4.2, and presented a theoretical background for them in Section 7.5. Based on these observations, this work proposes two memory scheduling schemes: a) Instruction Throughput Targeted Scheme (ITS), and b) Weighted Speedup Targeted Scheme (WEIS).

1) Instruction Throughput Targeted Scheme (ITS): This scheme aims to improve IT based on the observation that the application having lower $MPKI$ should be prioritized. In order to determine the application with lower $MPKI$, this scheduler first periodically (every 1024 cycles[4]) calculates two metrics during run-time: 1) the number of L2 misses for each application locally at each memory partition, and 2) the number of instructions committed by each application. Then, the information regarding the committed instructions is propagated to the MCs. The proposed scheduler calculates $MPKI$ of all the applications locally at each MC, using an arithmetic unit. Then, by using a comparator, the proposed scheduler determines the application with the lowest $MPKI$, and prioritize the

[4]This work also used three other sampling size windows (256, 1024, 2048) cycles. The difference in overall average performance is less than 1%, implying that sampling window size does not have a significant impact on the design.

Table 7.2: Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.1 [116] for full list.

Core Features	1400MHz core clock, 30 SMs, SIMT width = 32, Greedy-then-oldest first (GTO) dual warp scheduler [4]
Resources / Core [38, 67, 116]	16KB shared memory, 16KB register file, Max. 1536 threads
Private Caches / Core [38, 67, 116]	16KB 4-way L1 data cache 12KB 24-way texture cache 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
Shared L2 Cache	16-way 128 KB/memory channel (768KB in total)
Features	Memory coalescing and inter-warp merging enabled, immediate post dominator based branch divergence handling
Memory Model	6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling , 8 DRAM-banks/MC, 4 bank-groups/MC, 924 MHz memory clock Hynix GDDR5 Timing [107], $t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$
Interconnect [116]	1 crossbar/direction (30 SMs, 6 MCs), 1400MHz interconnect clock

memory requests generated by that application in the MC.

2) Weighted Speedup Targeted Scheme (WEIS): This scheme aims to improve WS based on the observation that the application having lower BW should be prioritized. In order to determine the application with lower BW, the proposed scheduler periodically calculates the amount of data transferred over the DRAM bus for each application locally at each memory partition. Then, by using a comparator, the proposed scheduler determines the application with the lowest BW, and prioritize the memory requests generated by that application in the MC.

Implementation of the Priority Mechanism: The priority mechanism takes advantage of the already existing FR-FCFS memory scheduler implementation. However, after serving the requests that generate DRAM row buffer hits, instead of picking request in the FCFS order, the proposed scheduler picks the oldest request from the highest priority application. When there is no request from the highest priority application in the MC queue, the proposed scheduler picks the oldest request originating from the application with the next highest priority.

7.7 Infrastructure and Evaluation Methodology

Infrastructure: Most of the prior GPU research (e.g. [7, 9, 27, 36, 51, 83]) is focused on improving the performance of a single GPU application, and is evaluated based on the benchmarks originating from different application suites

(Section 7.3.2). However, to investigate the research issues in the context of multiple applications, these applications need to be concurrently executed on the same GPU platform. This is a non-trivial task because it involves building a framework that can launch existing CUDA applications in parallel without significant changes to the source code. In order to do so, a new framework is developed, called GPU concurrent application framework (GCA). This framework takes advantage of CUDA *streams*. A stream is defined as a series of memory operations and kernel launches that are required to execute sequentially, however, different streams can be executed in parallel. The GCA framework creates a separate stream for each application, and issues all its associated commands to the stream. As many of the legacy CUDA codes use synchronous (e.g., *cudaMemcpy()*) memory transfer operations, the framework does source code modifications to change them to asynchronous CUDA API calls (e.g., *cudaMemcpyAsync()*). To ensure correct execution of multiple streams, the framework also adds appropriate synchronization constructs (e.g., *cudaStreamSynchronize()*) to the source code at correct places. After these steps are performed, GCA is ready to execute multiple streams/applications either on real GPU hardware or on the simulator. This chapter uses GCA to concurrently execute workloads on GPGPU-Sim, which is already capable of concurrently running multiple CUDA streams. As a part of initial package, the suite contains 300 ($\binom{25}{2}$) 2-application workloads and 2300 ($\binom{25}{3}$) 3-application workloads. Also, the initial GCA package includes guidelines to add a new CUDA code to the suite.

Evaluation Methodology: This work simulates both two- and three-application workloads on GPGPU-Sim [31], a cycle-accurate GPU simulator. Table 7.2 provides the details of simulation configuration. GCA framework launches CUDA applications on GPGPU-Sim and executes until the point where all the applications complete at least once. To achieve this, GCA framework relaunches the faster running application(s) until the slowest application completes its execution. This work collects the statistics of individual applications when they finish their respective executions such that amount of work done by individual applications across different runs is consistent. This methodology is consistent with the prior works [105]. This work only simulates

application kernels, and do not have a performance simulation for the data transfer between CPU and GPU. The DRAM contention model is the same that comes with the default GPGPU-Sim distribution, which is validated across many workloads [31].

Workload Classification: This work evaluates 100 two-application workloads and classify them based on two criteria. The first classification is based on the $MPKI$ difference between the applications in the workload. If this difference is greater than 10, the workload belongs to Class-A-MPKI. If it is less than 1, the workload belongs to Class-C-MPKI, otherwise it belongs to Class-B-MPKI. The second classification is based on the BW/C (bandwidth-utilization) difference between the applications in the workload. If this difference is greater than 50%, the workload belongs to Class-A-BW. If it is less than 25%, the workload belongs to Class-C-BW, otherwise it belongs to Class-B-BW. The intuition behind this classification method is that the workloads with high $MPKI$ and high BW difference are more likely to benefit from ITS and WEIS, respectively.

7.8 Experimental Results

This section evaluates five memory scheduling mechanisms: 1) the baseline FR-FCFS, 2) the recently proposed RR FR-FCFS [28], 3) ITS, 4) WEIS, 5) a static mechanism that always prioritizes the lowest $MPKI$ application in the workload, Prior_App_Low_MPKI, and 6) a static mechanism that always prioritizes the application in the workload with the lowest BW^{alone}, Prior_App_Low_BW. Note that the above static mechanisms require offline profiling that are difficult to employ during run-time. This work uses these static schemes as comparison points for ITS and WEIS, respectively. This work uses equal core partitioning across SMs for the applications in a workload, and show sensitivity to different core partitioning schemes in Section 7.8.4. This work uses geometric mean (GM) to report average performance results. For each proposed scheme, this work first demonstrates how effective the algorithm is in modulating the bandwidth given to each application, and then this work shows the

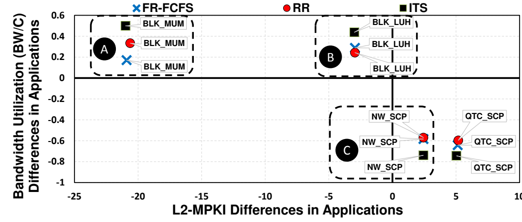


Figure 7.7: The effect of FR-FCFS, RR, and ITS on $BW_1 - BW_2$ and $MPKI_1 - MPKI_2$

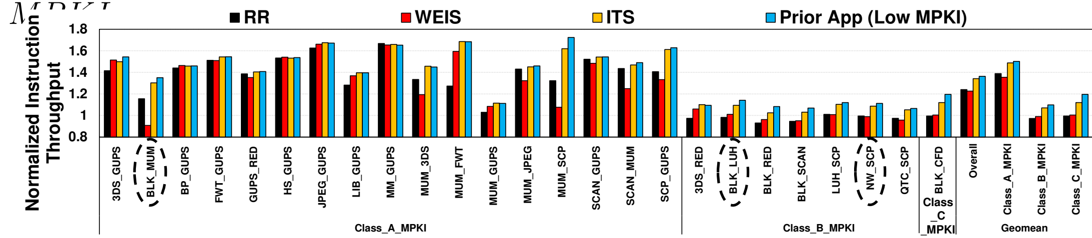


Figure 7.8: IT results normalized with respect to FR-FCFS for 25 representative workloads.

performance results.

7.8.1 Evaluation of ITS

How ITS Works: Figure 7.7 shows the effect of FR-FCFS, RR FR-FCFS, and ITS on four representative workloads. The x-axis shows $MPKI_1 - MPKI_2$, and the y-axis shows $BW_1 - BW_2$.

In BLK_MUM which is shown in **A**, since MUM has higher $MPKI$ than BLK, and BLK attains higher BW than MUM, all the points inside **A** are in the second quadrant. ITS prioritizes BLK due to its relatively lower $MPKI$, thus, the difference between BW attained by BLK and MUM increases with respect to FR-FCFS. This work observes an interesting case in RR. Since BLK already attains higher bandwidth with FR-FCFS, we would expect MUM to find more opportunity to utilize DRAM with RR. However, the RR mechanism employed by Jog et al. [28] preserves row-locality while scheduling requests. Therefore, this mechanism, although expected to give equal share of bandwidth to both applications, provides more opportunity to the application with higher row-locality for scheduling its requests. In other words, RR provides the applications with an equal opportunity to activate rows, resulting in the application with higher row-locality to schedule more requests due to their differences between the number of requests served per active row-buffer.

The exact phenomenon is observed in Ⓐ with RR, where BLK has 4× higher row-locality than that of MUM, leading to RR giving BLK even higher opportunity to schedule its requests compared to FR-FCFS. However, ITS unilaterally prefers BLK due to its consistently lower $MPKI$.

In BLK_LUH, which is shown in Ⓑ, we observe very similar trends as in Ⓐ. However, as opposed to Ⓐ, RR reduces the gap between BW achieved by LUH and BLK, since both applications have similar row-localities. In NW_SCP which is shown in Ⓒ, since NW has higher $MPKI$ than SCP, and SCP attains higher bandwidth than NW, all the points inside Ⓒ are in the fourth quadrant. ITS prioritizes SCP due to its relatively lower $MPKI$, thus, SCP achieves even more BW compared to FR-FCFS. We observe almost the same behavior with QTC_SCP as well. Note that, $MPKI$ values of the applications in the workload do not change across schemes, as it is an application property and each application has its own L2 cache partition.

ITS Performance: Figure 7.8 shows the instruction throughput of RR, WEIS, ITS, and Prior_App_Low_MPKI normalized with respect to FR-FCFS, using 25 representative workloads that span across $MPKI$-based workload classes, chosen from the pool of 100 workloads. We also show the average IT of these workloads including the individual GM for each workload class. As expected, in BLK_MUM previously shown in Ⓐ (Figure 7.7), IT improves by 15% with RR, and by 30% with ITS. We observe that, employing WEIS provides improvements over FR-FCFS, but results in slightly lower average performance than RR. It is expected, because WEIS is not targeted to optimize IT. With ITS, we observe 34% and 8% average IT improvements over FR-FCFS and RR, respectively, across 25 workloads. These numbers are 49% and 7% for Class-A-MPKI applications, because the workloads that have applications with strikingly different $MPKI$s are more likely to benefit with ITS over FR-FCFS. Class-B-MPKI and Class-C-MPKI also gain moderate performance improvements, by 7% and 12% over FR-FCFS, respectively. As we have shown in Figure 7.7, $MPKI$ does not change significantly. Thus, dynamism of ITS does not provide extra IT benefits over Prior_App_Low_MPKI.

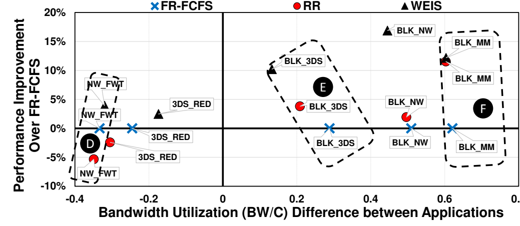


Figure 7.9: Effect of FR-FCFS, RR, and WEIS on WS and $BW_1 - BW_2$.

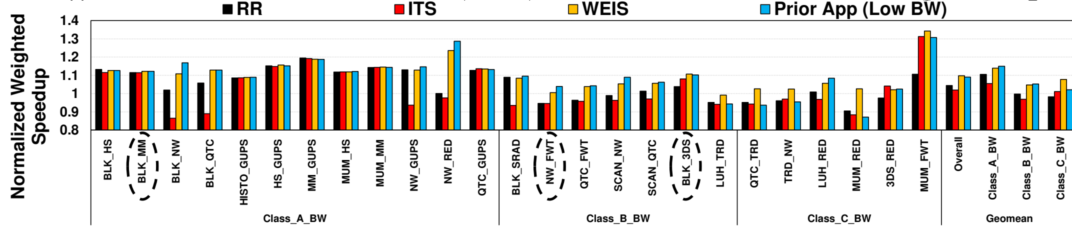


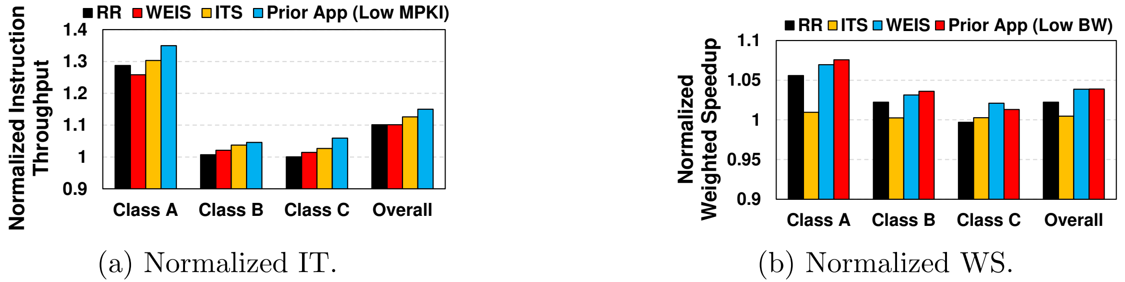
Figure 7.10: WS results normalized with respect to FR-FCFS for 25 representative workloads.

7.8.2 Evaluation of WEIS

How WEIS Works: Figure 7.9 shows the effect of FR-FCFS, RR, and ITS on five representative workloads. The x-axis shows $BW_1 - BW_2$, and the y-axis shows the normalized WS improvement over FR-FCFS. WEIS attempts to reduce the difference between BW attained by the applications, and therefore in the figure, we expect WEIS to push the workloads towards the y-axis. Also, as it is expected to improve WS, it also pushes the workload upwards. In `NW_FWT` which is shown in **D**, BW of `FWT` is higher than `NW`. WEIS prefers `NW` as it attained lower BW, which pushes this workloads upwards and towards y-axis. The RR mechanism degrades WS for `NW_FWT` because of similar reasons related to row-locality as pointed earlier (`FWT` has 13× higher row-locality than `NW`).

In `BLK_3DS` (**E**) and `BLK_MM` (**F**) both RR and WEIS push the workload towards y-axis along with improving WS. We observe that the trends in both RR and WEIS are similar in `3DS_RED` and `NW_FWT`, and also in `BLK_NW` and `BLK_3DS`.

WEIS Performance: Figure 7.10 shows WS of RR, WEIS, ITS, and Prior_App_Low_BW normalized with respect to FR-FCFS, using 25 representative workloads that span across BW-based workload classes, chosen from the pool of 100 workloads. We also show the average WS of these



(a) Normalized IT. (b) Normalized WS.

Figure 7.11: Summary IT and WS results for 100 workloads, normalized with respect to FR-FCFS.

workloads including the individual GM for each workload class. We observe that, employing ITS provides improvements over FR-FCFS, but results in lower average performance than RR. This is expected, because ITS is not targeted to optimize WS. With WEIS, we observe 10% and 5% average WS improvements over FR-FCFS and RR, respectively, across 25 workloads. These numbers are 14% and 3% for Class-A-BW applications, because the workloads that have applications with strikingly different BW are more likely to benefit with WEIS, compared to FR-FCFS. Class-B-BW and Class-C-BW also gain performance improvements, by 5% and 8% over FR-FCFS, respectively. In Class-C-BW workloads, WEIS performs much better than Prior_App_Low_BW. This is because the average BW difference is not significant, and it is more likely that the same application does not achieve consistently lower bandwidth than the other application. Therefore, prioritizing an application unilaterally like Prior_App_Low_BW does may lead to sub-optimal performance.

7.8.3 Performance Summary

We evaluate ITS and WEIS for 100 workloads and we observe in Figure 7.11 that the conclusions from previous discussions hold true for a wide range of workloads.

In Figure 7.12, we report Harmonic Speedup (HS) for 100 workloads in order to gauge WEIS with a balanced metric for performance as well as fairness [45]. Across all classes of workloads, we consistently observe better HS compared of FR-FCFS and RR. On average, RR and WEIS achieve 4% and 8% higher HS over FR-FCFS.

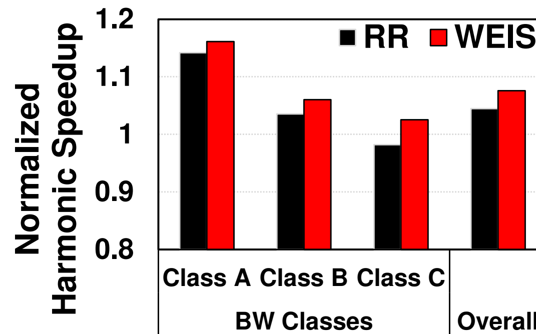


Figure 7.12: HS results for 100 workloads normalized with respect to FR-FCFS.

7.8.4 Scalability Analysis

Application Scalability: This work evaluates ITS and WEIS in the scenario when three applications are executed concurrently. This work observes that the impact of the schemes is even higher, as there is significant increase in memory interference among three applications. Figure 7.14 shows normalized IT and WS improvements with ITS and WEIS, respectively for 10 workloads. This work observes significant IT improvement (27%) in `GUPS_SCP_HISTO`, as ITS prefers `HISTO` because of its significantly lower $MPKI$ than other applications in the workload. For WEIS, this work also observes similar trends as discussed before.

Core Partitioning: This work evaluates three core partitioning configurations: (10,20), (20,10), and the baseline (15,15). Figure 7.13 shows normalized IT improvements of ITS for `JPEG_GUPS`, over FR-FCFS when it is used in their respective configurations. This work observes in all three configurations that the improvements in `JPEG_GUPS` are significant. However, if fewer cores (ITS (20,10)) are assigned to `GUPS`, which is a very high memory demanding application, the negative interference effect on `JPEG` is reduced.

Therefore, the relative IT improvements in ITS (20,10) is lower than ITS (15, 15). In the case of ITS (10, 20), the alone IPC of `JPEG` is lower as `JPEG` is assigned to fewer cores. This leads to lower scope in `JPEG` IPC improvements compared to the baseline ITS (15, 15) case. These results indicate that although core partitioning mechanisms affect the magnitude of interference, the problem still remains significant.

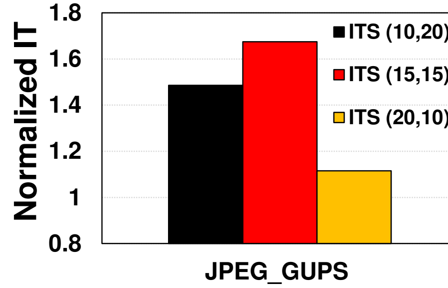
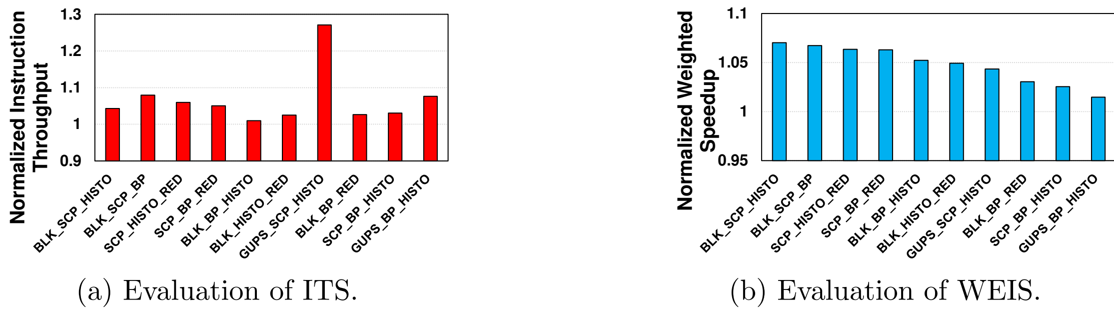


Figure 7.13: Core partitioning results.



(a) Evaluation of ITS.

(b) Evaluation of WEIS.

Figure 7.14: Evaluation of ITS and WEIS with three GPU applications.

7.9 Related Work

This work analyzes the interactions of multiple applications in GPU memory system, via both experiments as well as a mathematical model.

GPU and SoC Memory Scheduling: Prior work on memory scheduling for GPUs has dealt with a single application context only. Yuan et al. [49] proposed an arbitration mechanism in NoC to restore the lost row-buffer locality to enable a simple in-order DRAM memory scheduler. Lakshminarayana et al. [50] explored a DRAM scheduling policy that essentially chooses between Shortest Job First (SJF) and FR-FCFS [33,34]. Chatterjee et al. [83] proposed a warp-aware memory scheduling mechanism that reduces the DRAM latency divergence in a warp by avoiding the interleaved servicing of memory requests from different warps. The benefits from the above schedulers are orthogonal to the schemes and some of these mechanisms can be adapted as secondary arbitration criteria between requests for the currently prioritized application in the proposed scheduler. In the SoC space, Jeong et al. [117] proposed allowing the GPU to consume only the required bandwidth to maintain a certain real-time QoS-level for graphics applications.

Ausavarungnirun et al. [51] proposed a memory scheduling technique for CPU-GPU architectures. However, the overriding motivations for such prior work is to obtain the lowest possible latency for CPU requests without degrading the bandwidth utilization of the channel.

CPU Memory Schedulers: The impact of memory scheduling on multicore CPU systems has been a topic of significant interest in recent years [37, 45–47, 94, 118]. Ebrahimi et al. [93] proposed parallel application memory scheduling, where they explicitly managed inter-thread memory interference for improving performance. The Thread Cluster Memory Scheduler (TCM) [45] is particularly relevant because not only did it advocate prioritizing latency-sensitive applications, it identified that the main source of unfairness is the interference between different bandwidth intensive applications. To improve performance and fairness, TCM ranks the bandwidth-intensive threads based on their relative bank-level parallelism and row-buffer locality, and periodically shuffles the priority of the threads. The TCM technique is an advancement over the ATLAS [46], PARBS [37], and STFM [47] mechanisms that do not distinguish between bandwidth-intensive threads while improving performance and fairness.

This dissertation work shares the same objectives as the CPU schedulers like TCM, but the motivations and considerations behind the proposed schedulers, as well as the implementation and derived insight are significantly different. First, prior CPU memory schedulers concentrated only on single-threaded or modestly multi-threaded/multi-programmed workloads, while this dissertation work demonstrates the benefits of the proposed schemes for multiple, massively-threaded applications running on a fundamentally different architecture (SIMT). Second, the analysis in Sec. 7.3 establishes a different set of metrics from TCM, *viz.* $MPKI$ and attained bandwidth, to guide the memory scheduling at the application level (as opposed to single threads as in TCM). This is partly due to the use of TLP in GPU programs to hide memory latency as opposed to ILP and MLP in CPUs. Third, in contrast to prior works, this dissertation work demonstrates that the same scheduler can not achieve the best of both aggregate throughput and fairness. Consequently, a practical

implementation can use runtime settings to choose between high throughput and fairness-optimized scheduling based on application domains (e.g., high aggregate throughput in HPC applications vs QoS guarantees in virtualized GPUs). The final differentiator between TCM and the proposed mechanisms is complexity. TCM needs to track *each thread's* MLP, bank-level parallelism (BLP), and row-buffer locality (RBL), and requires an expensive insertion sort-like procedure to shuffle the ranks of high-MLP applications. In contrast, this dissertation work only require the L2 MPKI and currently sustained bandwidth information for *each application*, and a few simple comparisons in each time quanta. Scaling TCM's policies for the many thousands of concurrent threads in a GPU would be challenging in GDDR5 MC that has to support multi-gigabit command issue rates.

Concurrent execution of multiple applications on GPUs: Adriaens et al. [108] proposed spatial partitioning of SM resources across concurrent applications. They presented a variety of heuristics for dividing the SM resources across applications. Pai et al. [105] proposed elastic kernels that allow a fine-grained control over their resource usage. None of these works addressed the problem of contention in the memory system. Moreover, this dissertation work shows that memory interference problem remains significant regardless of SM-partitioning mechanisms, and this dissertation work believes the proposed schemes are complementary to core resource partitioning techniques. Gregg et al. [109] presented KernelMerge, a runtime framework to understand and investigate concurrency issues for OpenCL applications. Wang et al. [110] proposed context funneling, which allows kernels from different programs to execute concurrently. This dissertation work presents a new GCA framework that consists of large number of CUDA workloads and also provides flexibility to add new CUDA codes in the framework without much effort.

7.10 Chapter Summary

This work presents an in-depth analysis of GPU memory system in a multiple-application domain. This work shows that co-scheduled applications can

significantly interfere in the GPU memory system leading to significant loss in overall performance. To address this problem, an analytical model is developed that indicates that L2-MPKI and attained bandwidth are the two knobs that can be used to drive memory scheduling decisions for achieving better performance.

Chapter 8

Conclusions and Future Research Directions

GPU-based computing is expected to grow in coming years in many areas of science and engineering as well as in consumer applications to meet the computing demand. Furthermore as heterogeneous multi-cores consisting of CPUs, GPGPUs and other accelerators are projected to be one of the most cost-effective computing paradigms and as all major chip vendors are aligning their investment along this direction, hardware innovations for GPGPUs become critical for future computing systems.

8.1 Summary of Dissertation Contributions

The research proposed in this dissertation has four major contributions.

First, a new memory-aware warp scheduling policy is developed to enhance GPGPU performance by overcoming the resource under-utilization problem caused by long latency memory operations. The key idea is to take advantage of characteristics of cooperative thread arrays (CTAs) to concurrently improve cache hit rate, latency hiding capability, and DRAM bank parallelism in GPGPUs. The proposed memory-aware warp scheduling policy achieves these benefits by 1) selecting and prioritizing a group of CTAs scheduled on a core,

thereby improving both L1 cache hit rates and latency tolerance, 2) scheduling CTA groups that likely do not access the same memory banks on different cores, thereby improving DRAM bank parallelism, and 3) employing opportunistic memory-side prefetching to take advantage of already-open DRAM rows, thereby improving both DRAM row locality and cache hit rates.

Second, a *prefetch-aware warp scheduling* technique is developed, which has the capability to orchestrate with prefetching decisions. The main idea is to form groups of thread warps such that those that have good spatial locality are in separate groups. Since warps in different thread groups are scheduled at separate times, *not* immediately after each other, this scheduling policy enables the prefetcher to have more time to hide the memory latency. This scheduling policy also better exploits memory bank-level parallelism, even when employed without prefetching, as threads in the same group are more likely to spread their memory requests across memory banks. Experimental evaluations show that the proposed prefetch-aware warp scheduling policy improves performance compared to two state-of-the-art scheduling policies, when employed with or without a hardware prefetcher that is based on spatial locality detection. This work shows that orchestrating thread scheduling and data prefetching decisions in a GPGPU architecture via prefetch-aware warp scheduling can provide a promising way to improve memory latency tolerance in GPGPU architectures.

Third, a new GPU memory scheduler is developed, called CLAMS. The unique feature of this scheduler compared to all other existing techniques is that it uses both criticality and locality of pending memory requests to service the next request, while prior schemes use only the locality metric. The rationale for using this dual parameter based scheduling is that not all memory requests for an application exhibit similar latency criticality, and servicing them ahead of the other requests improves application performance. The evaluations show that CLAMS can provide significant performance benefits for the class of applications that exhibit high variance in criticality across cores, without hurting the performance of other applications. Considering that GPUs applications are more latency tolerant than their CPU counterparts due to high TLP, enhancing performance benefits through memory scheduling is non-trivial. This work shows

that considering core criticality is a promising way of improving GPU performance and can be exploited in GPU and CPU-GPU memory systems.

Lastly, an in-depth analysis of GPU memory system in a multiple-application domain is presented. It is shown that co-scheduled applications can significantly interfere in the GPU memory system leading to significant loss in overall performance. To address this problem, an analytical model is developed that indicates that L2-MPKI and attained bandwidth are the two knobs that can be used to drive memory scheduling decisions for achieving better performance.

8.2 Future Research Directions

I believe in the concept of ubiquitous computing and envision that all types of computing systems will take advantage of GPUs by considering them as an important class of computing citizens instead of mere co-processors. In this context, there are many research opportunities for inventing novel architectures, scheduling mechanisms, hardware/software interfaces as well as investigating system and security issues related to GPU-based systems. Some of the research directions are:

(I) Futuristic GPU Computing. In order to facilitate efficient GPU computing, the community is exploring features such as 1) dynamic parallelism, where a GPU can generate new work for itself; 2) Hyper-Q, where multiple CPUs can launch computations on a GPU; and 3) hybrid architectures consisting of CPUs and GPUs on the same die. However, many of these concepts are still in their infancy, and it is interesting to have an in-depth understanding of the design space for each of these avenues and their combinations. It is also interesting to develop new execution models and architectures involving multiple GPUs and other kinds of accelerators. In future, these architectures will also concurrently execute multiple applications, potentially originating from different users. In this context of multi-application execution, there are a lot of research opportunities on 1) designing an application-aware on-chip network fabric and memory hierarchy; 2) developing shared resource management, scheduling and

concurrency management techniques; and 3) designing light-weight and efficient hardware and software support that deals with virtualization, security, and other system issues. As a part of of long-term research, it is also interesting to extend these techniques to situations where applications with different latency and bandwidth demands are executed concurrently on mobile and wearable devices under stricter energy and power constraints.

(II) Near-Data Computing in GPU-based systems. Near-data computing architectures are built on the notion that moving computation near the data is far more beneficial in terms of performance and energy efficiency than moving data near the computation. In the context of data-intensive computing, such architectures can be useful as they can significantly cut down the movement of data between memory/storage and computation units, and therefore, can conserve precious bandwidth. However, there are many questions that need to be answered before realizing such architectures. Some research questions that are worth investigating are: 1) Which parts of the CUDA/OpenCL application code triggers significant amounts of data movement?; 2) Between which components (e.g., between CPUs and GPUs, or between multiple GPUs, or between different levels of the CPU/GPU memory hierarchy) is the data movement the most expensive in terms of performance and energy; 3) Given the cost and the amount of data movement, between which components should the movement of data be minimized?; 4) Which parts of the computation can be computed near memory/storage; 5) What architectural enhancements are required to perform the computation near memory/storage?; and 6) How one can leverage my past insights from my dissertation work to design efficient techniques to co-schedule data and computation for minimizing the data movement? It is also interesting to explore the opportunities for employing approximate computing and leveraging different emerging memory/storage technologies in conjunction with near-data computing.

(III) Commonality-Aware GPU Computing. There is an increasing trend to co-host multiple applications or games from different users on the same GPU cloud platform. For example, many service providers have started to use cloud gaming technology (e.g., NVIDIA GRID) as the foundation for their on-demand

Gaming as a Service (GaaS) solution. In this context, a multi-player gaming environment might require rendering of similar scenes/objects separately for every player. Such redundant rendering computations are not only costly from performance and energy perspective but also can consume significant GPU memory space. To this end, it is interesting to pursue at least two major aspects towards designing commonality-aware GPU systems. The first aspect is to design mechanisms for detecting computation and data commonality across concurrent applications. The second aspect is to design scheduling mechanisms that can effectively co-schedule applications possessing high-commonality on the same GPU hardware. In my opinion, both aspects should be pursued synergistically for enabling inter-application optimizations for improving the overall performance and energy efficiency. As a part of long-term research, it is also interesting to address the security vulnerabilities that might arise because of commonality-aware GPU computing.

Bibliography

[1] D. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[2] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7–17, 2011.

[3] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *CC/ETAPS 2010*.

[4] NVIDIA, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," 2011.

[5] ATI, "Radeon gpus," *http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/Pages/amd-radeon-hd-6000.aspx*.

[6] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, O. Mutlu, C. Das, M. T. Kandemir, T. Mowry, and R. Ausavarungnirun, "Enabling Efficient Data Compression in GPUs," in *ISCA*, 2015.

[7] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[8] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.

[9] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *PACT*, 2013.

[10] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.

[11] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science & Engineering*, vol. 13, no. 3, pp. 92–95, 2011.

[12] nvidia, "Gpu supercomputers show exponential growth in top500 list," *http://blogs.nvidia.com/blog/2011/11/14/gpu-supercomputers-show-exponential-growth-in-top500-list/*.

[13] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the gpu-past, present and future," *Medical Image Analysis*, 2013.

[14] G. Pratx and L. Xing, "Gpu computing in medical physics: A review," *Medical physics*, vol. 38, p. 2685, 2011.

[15] S. S. Stone, J. P. Haldar, S. C. Tsao, W. mei W. Hwu, B. P. Sutton, and Z.-P. Liang, "Accelerating advanced MRI reconstructions on GPUs," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1307–1318, 2008.

[16] I. Schmerken, "Wall street accelerates options analysis with gpu technology," *2008-11-07)[2009-11-02]. http://wallstreetandtech. com/technology-risk-management/showArticle. jhtml*, 2009.

[17] NVIDIA, "Jp morgan speeds risk calculations with nvidia gpus," 2011.

[18] nvidia, "Computational finance,"

[19] nvidia, "Researchers deploy gpus to build world's largest artificial neural network," *http://nvidianews.nvidia.com/Releases/Researchers-Deploy-GPUs-to-Build-World-s-Largest-Artificial-Neural-Network-9c7.aspx*.

[20] nvidia, "How to harness big data for improving public health," *http://www.govhealthit.com/news/how-harness-big-data-improving-public-health*.

[21] P. Hofstee, "The big deal about big data - a perspective from ibm research," *http://www.nas-conference.org/NAS-2013/invited.html*.

[22] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[23] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011.

[24] K. O. W. Group *et al.*, "The opencl specification," *A. Munshi, Ed*, 2008.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, 2008.

[26] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization," in *SC*, 2011.

[27] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[28] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications," in *GPGPU*, 2014.

[29] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU Memory System for Multi-Application Execution," in *MEMSYS*, 2015.

[30] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *Proceedings of the 2010 43rd Annual IEEE/ACM international symposium on Microarchitecture*, pp. 421–432, IEEE Computer Society, 2010.

[31] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[32] S. Rixner, "Memory Controller Optimizations for Web Servers," in *MICRO*, 2004.

[33] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order," Sept. 1997.

[34] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory Access Scheduling," in *ISCA*, 2000.

[35] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in *MICRO*, 2010.

[36] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *ICS*, 2012.

[37] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[38] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *MICRO*, 2013.

[39] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs," in *DAC*, 2012.

[40] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," 2008.

[41] nvidia, "Nvidia gpus," *http://www.nvidia.com/content/global/global.php.*

[42] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[43] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Tech. Rep. IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.

[44] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.

[45] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[46] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[47] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[48] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.

[49] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures," in *MICRO*, 2009.

[50] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin, "DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function," *Computer Architecture Letters*, 2012.

[51] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[52] S. Hassan, D. Choudhary, M. Rasquinha, and S. Yalamanchili, "Regulating Locality vs. Parallelism Tradeoffs in Multiple Memory Controller Environments," in *PACT*, 2011.

[53] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, " Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems ," in *HPCA*, 2012.

[54] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: building a smarter memory controller," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, 1999.

[55] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," in *ISCA*, 1997.

[56] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *HPCA*, 2007.

[57] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," in *HPCA*, 2004.

[58] J. Doweck, "Inside Intel Core Microarchitecture and Smart Memory Access," tech. rep., Intel Corporation, 2006.

[59] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-core Systems," in *MICRO*, 2009.

[60] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM J. Res. Dev.*, Jan. 2002.

[61] T. L. Johnson, M. C. Merten, and W. W. Hwu, "Run-Time Spatial Locality Detection and Optimization," in *MICRO*, 1997.

[62] J. D. Gindele, "Buffer Block Prefetching Method," *IBM Technical Disclosure Bulletin*, vol. 20, July 1977.

[63] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *ISCA*, 1990.

[64] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *MICRO*, 2011.

[65] Synopsys Inc., *Design Compiler*.

[66] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," ISCA, 2006.

[67] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[68] J. Lee, H. Kim, and R. Vuduc, "When Prefetching Works, When It Doesn't, and Why," in *TACO*, 2012.

[69] D. Chiou, S. Devadas, J. Jacobs, P. Jain, V. Lee, E. Peserico, P. Portante, L. Rudolph, G. E. Suh, and D. Willenson, "Scheduler-Based Prefetching for Multilevel Memories," Tech. Rep. Memo 444, MIT, July 2001.

[70] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for Bandwidth-efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *HPCA*, 2009.

[71] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving Memory Bank-level Parallelism in the Presence of Prefetching," in *MICRO*, 2009.

[72] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-Aware DRAM Controllers," in *MICRO*, 2008.

[73] T. Moscibroda and O. Mutlu, "Distributed Order Scheduling and Its Application to Multi-core Dram Controllers," in *PODC*, 2008.

[74] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *ISCA*, 2010.

[75] M. Arora, S. Nath, S. Mazumdar, S. B. Baden, and D. M. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro*, pp. 4–16, 2012.

[76] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[77] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency," in *PACT*, 2013.

[78] S.-Y. Lee and C.-J. Wu, "Characterizing GPU Latency Hiding Ability," in *ISPASS*, 2014.

[79] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[80] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *ASPLOS*, 2014.

[81] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: gating aware scheduling and power gating for GPGPUs," in *MICRO*, 2013.

[82] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *HPCA*, 2013.

[83] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC*, 2014.

[84] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Comput. Archit. Lett.*

[85] N. Goswami, B. Cao, and T. Li, "Power-performance co-optimization of throughput core architecture using resistive memory," in *HPCA*, 2013.

[86] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in *MICRO*, 2013.

[87] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *IISWC*, 2010.

[88] J. Wang and Y. Sudhakar, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *IISWC*, 2014.

[89] S.-Y. Lee and C.-J. Wu, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in *PACT*, 2014.

[90] I. Karlin, A. Bhatele, J. Keasler, B. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IPDPS*, 2013.

[91] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *GPGPU*, 2010.

[92] Z. Guz, O. Itzhak, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Threads vs. caches: Modeling the behavior of parallel workloads," in *ICCD*, 2010.

[93] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel application memory scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, 2011.

[94] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *ISCA*, 2013.

[95] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.

[96] S. T. Srinivasan and A. R. Lebeck, "Load latency tolerance in dynamically scheduled processors," in *MICRO*, 1998.

[97] S. Srinivasan, R.-C. Ju, A. Lebeck, and C. Wilkerson, "Locality vs. criticality," in *ISCA*, 2001.

[98] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *DAC*, 2012.

[99] S. Subramaniam, A. Bracy, P. Wang, and G. Loh, "Criticality-based optimizations for efficient load processing," in *HPCA*, 2009.

[100] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *ISCA*, 2001.

[101] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González, "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *PACT*, 2008.

[102] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *ISCA*, 2009.

[103] "NVIDIA GTX 780-Ti." `http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/`.

[104] "AMD Radeon R9 290X." http://www.amd.com/us/press-releases/Pages/amd-radeon-r9-290x-2013oct24.aspx.

[105] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *ASPLOS*, 2013.

[106] "NVIDIA GRID." http://www.nvidia.com/object/grid-boards.html.

[107] Hynix., "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0."

[108] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The case for GPGPU spatial multitasking," in *HPCA*, 2012.

[109] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *HotPar*, 2012.

[110] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *HPCS*, 2011.

[111] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing data parallel hardware for server workloads," *SIGARCH Comput. Archit. News*.

[112] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aérgia: exploiting packet latency slack in on-chip networks," in *ACM SIGARCH Computer Architecture News*, ACM, 2010.

[113] X. Lin and R. Balasubramonian, "Refining the utility metric for utility-based cache partitioning," *Proc. WDDD*, 2011.

[114] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2006.

[115] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "Mise: Providing performance predictability and improving fairness in shared main memory systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, 2013.

[116] GPGPU-Sim v3.2.1, "GTX 480 Configuration."

[117] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 850–855, ACM, 2012.

[118] "3rd JILP Workshop on Computer Architecture Competitions (Memory Scheduling Championship)." http://www.cs.utah.edu/~rajeev/jwac12/.

Vita

Adwait Jog

Adwait Jog is a Ph.D. Candidate in the Department of Computer Science and Engineering at Penn State University. His research interests lie in the broad areas of computer architecture and systems, with an emphasis on designing high-performance and energy-efficient GPU-based platforms. He has received the Best Graduate Research Assistant Award at Penn State and was selected as one of 25 finalists for an NVIDIA Ph.D. Fellowship. His research has been published in major computer architecture conferences (ASPLOS, ISCA, MICRO, PACT). He has served as a technical reviewer for many journals and conferences including IEEE CAL, IEEE TC, ACM TECS, ACM TODAES, DAC, ICCD, ISCA, HPCA, ASPLOS, and MICRO. Adwait worked as an intern with NVIDIA Research in the summer of 2013, and with Intel in the summers of 2012 and 2011. Before joining Penn State with College of Engineering Fellowship, he completed his undergraduate studies at NIT Rourkela, India in 2009.