W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

2020

Design And Analysis Of Memory Management Techniques For Next-Generation Gpus

Haonan Wang
William & Mary - Arts & Sciences, haonan.wang07@gmail.com

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Wang, Haonan, "Design And Analysis Of Memory Management Techniques For Next-Generation Gpus" (2020). *Dissertations, Theses, and Masters Projects.* Paper 1616444486.
http://dx.doi.org/10.21220/s2-e46y-k152

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Design and Analysis of Memory Management Techniques for Next-generation GPUs

Haonan Wang

Zibo, Shandong, China

Bachelor of Science, East China University of Science and Technology, 2013
Master of Science, College of William & Mary, 2017

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
August 2020

© Copyright by Haonan Wang 2020

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Haonan Wang

Haonan Wang

Approved by the Committee, August 2020

Adwait Jog

Committee Chair
Adwait Jog, Assistant Professor, Computer Science
College of William & Mary

Xu Liu, Assistant Professor , Computer Science
College of William & Mary

Bin Ren

Bin Ren, Assistant Professor, Computer Science
College of William & Mary

Evgenia Smirni, Professor, Computer Science
College of William & Mary

Onur Kayiran, MTS Silicon Design Engineer
AMD Research

ABSTRACT

Graphics Processing Unit (GPU)-based architectures have become the default accelerator choice for a large number of data-parallel applications because they are able to provide high compute throughput at a competitive power budget. Unlike CPUs which typically have limited multi-threading capability, GPUs execute large numbers of threads concurrently to achieve high thread-level parallelism (TLP). While the computation of each thread requires its corresponding data to be loaded from or stored to the memory, the key to supporting the high TLP of GPUs lies in the high bandwidth provided by the GPU memory system. However, with the continuous scaling of GPUs, the challenges of designing an efficient GPU memory system have become two-fold. On one hand, to keep the growing compute and memory resources highly utilized, co-locating two or more kernels in the GPU has become an inevitable trend. One of the major roadblocks in achieving the maximum benefits of multi-application execution is the difficulty to design mechanisms that can efficiently and fairly manage the application interference in the shared caches and the main memory. On the other hand, to maintain the continuous scaling of GPU performance, the increasing energy consumption of the memory system has become a major problem because of its limited power budget. This limitation of the GPU memory energy restricts its maximum theoretical bandwidth and in turn limits the overall throughput.

To address the aforementioned challenges, this dissertation proposes three different approaches. First, this dissertation shows that high efficiency and fairness can be achieved for GPU multi-programming with novel TLP management techniques. We propose a new metric, *effective bandwidth (EB)*, to accurately estimate the shared resources in the GPU memory hierarchy. Meanwhile, we propose *pattern-based searching scheme (PBS)* that can quickly and accurately achieve efficiency or fairness via managing the TLP of each application. Second, to reduce data movement and improve GPU throughput, this dissertation develops *Address-Stride Assisted Approximate Value Predictor (ASAP)* for GPUs. We show that by utilizing address stride and value stride correlation present in GPGPU applications, significant data movement reduction and throughput improvement can be achieved at a much lower application quality loss and hardware overhead. ASAP achieves this by predicting load values if it detects strides in their corresponding addresses. Third, this dissertation shows that GPU memory energy can be significantly reduced by utilizing novel memory scheduling techniques. We propose a lazy memory scheduler which significantly improves the row buffer locality of GPU memory by leveraging the latency and error tolerance of GPGPU applications. Finally, our new work targets data movement reduction with flexible data precisions. We present initial results to motivate novel data types and architectural support to dynamically reduce the data size transferred per each memory operation. Altogether, this dissertation develops several innovative techniques to improve the GPU memory system efficiency, which are necessary for enabling the development of next-generation GPUs.

TABLE OF CONTENTS

Acknowledgments vi

Dedication vii

List of Tables viii

List of Figures ix

1 Introduction 2

1.1 Problem Statement 3

1.2 Contributions 3

1.2.1 Achieving Efficiency and Fairness in GPU Multi-programming via Effective Bandwidth Management 4

1.2.2 Reducing GPU Data Movement by Efficient Approximate Value Prediction 5

1.2.3 Improving GPU Memory Energy Efficiency with Lazy Memory Scheduling 5

2 A General Background on Graphics Processing Units (GPUs) 8

2.1 Baseline GPU Architecture 8

2.2 GPU Memory Organization 9

2.3 GPU Memory Operations and Scheduling Techniques 10

3 Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management 12
3.1 Introduction 12
3.2 Background and Evaluation Methodology 15
3.2.1 Evaluation Methodology 16
3.3 Analyzing Application Resource Consumption 19
3.3.1 Understanding Effects of TLP on Resource Consumption 19
3.3.2 Quantifying Resource Consumption 20
3.4 Motivation and Goals 22
3.5 Pattern-based Searching (PBS) 26
3.5.1 Overview 26
3.5.2 Optimizing WS via PBS-WS 27
3.5.3 Optimizing Fairness via PBS-FI 29
3.5.4 Optimizing HS via PBS-HS 30
3.5.5 Implementation Details and Overheads 31
3.6 Experimental Evaluation 33
3.6.1 Effect on Weighted Speedup 34
3.6.2 Effect on Fairness Index 36
3.6.3 Effect on Harmonic Weighted Speedup 37
3.6.4 Case Studies 38
3.7 Related Work 40
3.8 Chapter Summary 42

4 Address-Stride Assisted Approximate Load Value Prediction in GPUs 43
4.1 Introduction 43
4.2 Background 46
4.2.1 Baseline Architecture and Metrics 46

4.2.2 Baseline Value Predictors 47
4.3 Motivation and Analysis 50
4.3.1 Analysis of Address and Value Strides 50
4.3.2 Motivation 51
4.4 Design and Operation 53
4.4.1 Design of ASAP 53
4.4.2 Operation of ASAP 57
4.4.3 Use Cases of ASAP 58
4.4.4 Output Quality Control 61
4.4.5 Hardware Overhead 63
4.5 Evaluation Methodology 64
4.5.1 Application Characteristics 64
4.5.2 Choice of the Restricted Address Strides 65
4.6 Experimental Evaluation 65
4.6.1 Effect on Output Quality 65
4.6.2 Effect on Performance and Energy 68
4.7 Sensitivity Studies 69
4.8 Related Work 71
4.9 Chapter Summary 72

5 Exploiting Latency and Error Tolerance of GPGPU Applications for an Energy-efficient DRAM 73
5.1 Introduction 73
5.2 Background and Metrics 76
5.2.1 Evaluation Methodology and Metrics 77
5.3 Motivation and Analysis 79
5.3.1 Delayed Memory Scheduling (DMS) 79

5.3.2 Approximate Memory Scheduling (AMS) 82
5.3.3 Delayed and Approximate Scheduling 84
5.3.3.1 How can approximate memory scheduling help delayed memory scheduling 84
5.3.3.2 How can delayed memory scheduling help approximate memory scheduling 85
5.4 Design and Operation 87
5.4.1 Overview 87
5.4.2 Delayed Memory Scheduling Schemes 88
5.4.3 Approximate Memory Scheduling Schemes 90
5.4.4 Value Prediction Unit 93
5.4.5 Hardware Overhead 94
5.5 Experimental Results 95
5.6 Related Work 101
5.7 Chapter Summary 102

6 Towards Architectural Support for Flexible Data Precisions 103
6.1 Background and Metrics 104
6.1.1 Floating-Point Data Storage Formats 104
6.1.2 Value Dependency for Data Movement Energy 106
6.1.3 Evaluation Methodology and Metrics 107
6.2 Motivation and Analysis 108
6.2.1 Analysis of Floating-point Formats 108
6.2.2 Value Truncation for Floating-point Formats 110
6.2.3 Symmetrical Floating-point (SFP) Format 112
6.2.4 A Flexible Memory System 114
6.3 Conclusions 115

7 Conclusion and Future Work 116

7.1 Summary of Dissertation Contributions 116

7.2 Future Work 118

Bibliography 118

Vita 137

ACKNOWLEDGMENTS

I sincerely thank my advisor, Adwait Jog, for his thoughtful and diligent mentoring.

I also thank my labmates, Fan Luo, Gurunath Kadam, Hongyuan Liu and Mohamed Ibrahim for their help and companionship in my Ph.D. life.

I extend my gratitude to my dissertation committee members, Xu Liu, Bin Ren, Evgenia Smirni, and Onur Kayiran for their generous support and attentive feedback.

Finally, I would like to thank my parents, who have always been the firmest support on the road of my education. I would like to thank all my family members for their care and encouragement.

To my dear parents.

LIST OF TABLES

3.1 Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.2 [34] for the complete list. 16
3.2 List of evaluated TLP configurations. 17
3.3 List of evaluated metrics. 17
3.4 GPGPU application characteristics: *(A) IPC@bestTLP: The value of IPC when the application executes with bestTLP, (B) EB@bestTLP: The value of the effective bandwidth when the application executes with bestTLP, (C) Group information: Each application is categorized into one of the four groups (G1-G4) based on their individual EB values.* . 18

4.1 Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.2 [34] for the full list. 47
4.2 List of evaluated GPGPU applications. 63

5.1 Key configuration parameters of the simulated GPU. 76
5.2 List of evaluated GPGPU applications. See Table 5.3 for more details. 95
5.3 Application features and intensity classifications. The thresholds are used only to facilitate the discussion in Section 5.5. 96

6.1 Configurations of common floating-point data formats. 106
6.2 Configurations of symmetrical floating-point data formats. 112

LIST OF FIGURES

2.1 Overview of GPU Architecture. 9

3.1 Weighted Speedup (WS) and Fairness Index (FI) for `BFS2_FFT`. Evaluation methodology is described in Section 3.2. 14

3.2 Effect of TLP on performance and other metrics for `BFS2`. 20

3.3 Effective Bandwidth at different levels of the hierarchy. For brevity, we show only one core (with attached L1 cache) and one L2 cache partition. 21

3.4 Effect of different TLP combinations on application: a) slowdown and b) effective bandwidth. 22

3.5 IPC_{AR} vs. EB_{AR} 24

3.6 Illustrating the patterns observed in `BLK_TRD`. 28

3.7 Illustrating the working of PBS-FI (a & b) and PBS-HS (c & d) schemes for `BLK_TRD`. 30

3.8 Proposed hardware organization. Additional hardware is shown via shaded components and dashed arrows. 31

3.9 Impact of our schemes on Weighted Speedup. Results are normalized to ++bestTLP. 35

3.10 Impact of our schemes on Fairness. Results are normalized to ++bestTLP. 35

3.11 Effect of changes in TLP over time for BLK_BFS2 with: a) PBS-WS and b) PBS-FI. 36

3.12 Impact of our schemes on HS. 37
3.13 Effect of PBS over ++bestTLP with: a) core partitioning, b) cache partitioning, c) 3-application scaling. 38

4.1 Pixel values of consecutive row and column positions. 45
4.2 Baseline GPU Architecture with a value predictor. 46
4.3 Design of the baseline value predictors. 48
4.4 Illustrating the relationship between average value stride of data with different address strides for a variety of inputs. 50
4.5 Illustrative example showing the importance of request order on value strides and the ease of value predictability. 51
4.6 Normalized Stride Difference (in log scale) between *consecutively* observed value strides. Considering address-stride-based (2nd and 3rd bar) improves the value predictability over traditional PC-based approach (1st bar). 53
4.7 Design of the Address-Stride assisted value predictor. 55
4.8 Operation of ASAP and its advantages over OSP. The matched addresses, predicted values, relevant strides are shaded. 57
4.9 Working steps of ASAP in Scenario I: Regular Address Pattern. The address stream considered is: 0, 1, 2, 3, 10, 11, 12, 13. The matched addresses and relevant strides are shaded. 59
4.10 Working steps of ASAP in Scenario II: Interleaving Address Pattern. The addresses considered are: 1, 2, 4, 5, 7, 8, 10, 11. The matched addresses and relevant strides are shaded. 60
4.11 Working steps of ASAP in Scenario III: Missing Intermediate Address Pattern. The addresses considered are: 0, 1, 2, 3, 5. The matched addresses and relevant strides are shaded. 61

4.12 Application Error for different value predictors at (a) 10% (b) 20% coverage. 66
4.13 `EMBOSS(2DCONV)` outputs at 10% coverage. 67
4.14 GPU performance and total energy consumption at different coverages. 69
4.15 Effect of AddressStrideLong on Miss Match Rate. 69
4.16 Miss Match Rate with different entry numbers. 70
4.17 Miss Match Rate with GTO and RR Scheduler. 71

5.1 Effect of pending queue size on the number of row activations (Act.). Results are normalized to the case of pending queue size 128. 77
5.2 An example illustrating the benefits of delayed memory scheduling due to increased visibility to the memory controller. Eight requests are shown in total destined to four DRAM rows (R1, R2, R3, R4). . 79
5.3 Effect of delayed memory scheduling on the number of activations and performance. Results are normalized to the baseline architecture (Section 5.2), which does not employ delayed or approximate scheduling. 80
5.4 Effect of delayed memory scheduling on activation proportions of each RBL. x-axis indicates delay. y-axis indicates each component's proportion to the total number of activations. 81
5.5 The cumulative distribution of total row activations for requests associated with different RBLs. x-axis is the proportion of requests sorted by their RBLs. 83
5.6 Examples illustrating how approximate memory scheduling can help delayed memory scheduling. 84
5.7 Example illustrating how delayed memory scheduling (DMS) can help approximate memory scheduling (AMS) by comparing different schemes. 86

5.8 Design overview of the lazy memory scheduler and associated components. 88
5.9 Illustrating the relationship between IPC and BWUTIL. 89
5.10 Effect of reducing Th_{RBL}. 92
5.11 Comparison of different schemes with different metrics for applications with Medium or High Error Tolerance. Row Energy and IPC results are normalized to the baseline that does not adopt DMS or AMS. . . 97
5.12 Comparison between the accurate and the approximate output (which has 17% Application Error and is generated when the Dyn-DMS and Dyn-AMS schemes are applied together) for application `laplacian`. . 99
5.13 Effect of pending queue size on the number of activations (normalized to the baseline) with DMS(2048). 99
5.14 Comparison of different schemes in the delay-only mode for applications with Low Error Tolerance. 100

6.1 FP32 layout. 105
6.2 Number of ones and bit toggling for FP32 and SFP32. 108
6.3 Average relative error distribution for LSTM and MNIST when using short formats. 109
6.4 Layouts of the symmetrical floating-point (SFP) formats. 111
6.5 Flit mapping strategy enabled by SFP. 114

Design and Analysis of Memory Management Techniques for Next-generation GPUs

Chapter 1

Introduction

Graphics Processing Unit (GPU)-based architectures have become the default accelerator choice for a large number of data-parallel applications because GPUs are able to provide high compute throughput at a competitive power budget. Nowadays, GPU accelerated application are widely used in fields like machine learning [5, 89, 114], image and video processing [117, 27, 85], physical simulation [100, 13, 136], gene sequencing [105, 124, 109, 123] and even cryptography [18, 70, 31, 126]. On the other hand, GPUs are being employed into almost all kinds of computing systems, including many machines on Top500 [4] and Green500 lists [3].

Unlike CPUs which typically have limited multi-threading capability, GPUs execute large numbers of threads concurrently to achieve high thread level parallelism (i.e., TLP). While the computation of each thread requires its corresponding data to be loaded from or stored to the memory, the key to supporting the high TLP of GPUs lies in the high bandwidth provided by the GPU memory system. The GPU memory system consists of two levels of cache, L1 cache and L2 cache, and the L2 cache is further connected to the the memory channel. With the state of the art high-bandwidth memory technology (i.e., HBM, HBM2), each memory channel is able to provide 16-32 GB/Sec bandwidth [82, 68]. Meanwhile, the GPU is usually equipped with multiple memory channels, which all work independently of each other. Therefore, a peak bandwidth of 900 GB/Sec can be

achieved in the latest GPU model [81]. With each new generation of GPUs, peak memory bandwidth and throughput are growing at a steady pace [2, 128], and it is expected to continue as technology scales and new emerging high-bandwidth memories become mainstream.

1.1 Problem Statement

With the continuous scaling of GPU, the challenges of designing the GPU memory system have become two-fold. On one hand, to keep the growing compute and memory resources highly utilized, co-locating two or more kernels (originating from the same or different applications) in the GPU has become an inevitable trend [84, 46, 8, 35, 132, 47, 118, 135, 66, 86, 63, 72, 125, 87]. However, one of the major roadblocks in achieving the maximum benefit of multi-application execution is the difficulty to design mechanisms that can efficiently and fairly manage the application interference in the shared caches and the main memory. On the other hand, to maintain the continuous scaling of GPU performance, the energy efficiency of the memory system has become a major problem because of its high energy consumption and the limited GPU total power budget [16, 83, 57]. This limitation of the GPU memory can either restrict the GPU's maximum theoretical throughput directly or prevent the GPU memory from reaching its peak bandwidth, which in return reduces the overall throughput of the GPU. Therefore, in this dissertation, we focus to answer the following three questions: 1. How can we efficiently and fairly mange the memory resources for multiple co-running applications in the GPU? 2. How can we reduce the data movement in the memory hierarchy and improve the throughput of the GPU? 3. How can we improve the energy efficiency of the GPU memory?

1.2 Contributions

This dissertation addresses the three questions above by three different approaches. First, this dissertation shows that efficiency and fairness can be achieved in the GPU multi-

programming environment with more accurate metrics to measure the memory resources and efficient TLP management policies. Second, this dissertation shows that by utilizing address stride and value stride correlation, memory throughput can be improved through data movement reduction with low application quality loss and small hardware overhead. Third, this dissertation shows that GPU memory energy efficiency can be significantly improved by utilizing novel memory scheduling techniques. In the rest of this chapter, we discuss theses three contributions in details.

1.2.1 Achieving Efficiency and Fairness in GPU Multi-programming via Effective Bandwidth Management

We perform a detailed analysis of the TLP management techniques in the context of multi-application execution in GPUs and show that new TLP management techniques, if developed carefully, can significantly boost the system throughput and fairness. In this context, our *goal* is to develop techniques that can find the optimal TLP combination that allows a judicious and good use of all the available shared memory resources. To measure such use, we propose a new metric, *effective bandwidth (EB)*, which calculates the effective shared resource usage for each application considering its private and shared cache miss rates and memory bandwidth consumption. We find that a TLP combination that maximizes the total effective bandwidth across all co-located applications while providing a good balance of individual applications' effective bandwidth leads to high system throughput and fairness. Instead of incurring the high overheads of an exhaustive search across all the different combinations of TLP configurations that achieve these goals, we propose pattern-based searching (PBS) that cuts down a significant amount of overheads by taking advantage of the trends (which we call *patterns*) in the way application's effective bandwidth changes with different TLP combinations.

1.2.2 Reducing GPU Data Movement by Efficient Approximate Value Prediction

We propose a novel value approximation technique to reduce data movement for GPUs with low application quality loss and hardware overhead. One of the major challenges in achieving this goal is to identify the value stride pattern(s) in a highly multi-threaded environment where thousands of memory requests can be on-the-fly and their access order is highly dependent on GPU-specific features such as warp scheduling and coalescing. Previous works for CPUs used large per-thread prediction tables to achieve high accuracy [120, 75, 106]. However, it can become prohibitively expensive to apply those approaches directly to the highly multi-threaded environment in GPUs [139]. To address this problem, we take advantage of our key new observation that consideration of memory addresses and the relationship with their value strides is effective for providing high value prediction accuracy. Specifically, we find that for many realistic inputs used by GPGPU applications, particular address strides have linear correlations with their value strides. Based on this new observation, we propose an Address-Stride Assisted Approximate Value Predictor (ASAP), which predicts the values only if it detects strides in their corresponding addresses. Each entry in the ASAP prediction table carefully keeps track of one type of address stride and their corresponding value stride. We find that as the number of address stride patterns in typical GPGPU applications is usually limited, the number of prediction table entries is significantly reduced, thereby making it area and power-efficient. We also show that ASAP remains effective even under different address patterns, which can be influenced by warp scheduling and coalescing.

1.2.3 Improving GPU Memory Energy Efficiency with Lazy Memory Scheduling

We observe that several GPGPU applications suffer from poor row buffer reuse (also referred to as *row thrashing*). To address this problem, we performed a detailed charac-

terization of row buffer locality in GPUs and revealed two key insights. First, the current GPU memory scheduling policies are too aggressive in reducing latencies of requests: requests in the pending queue are issued to their destined DRAM banks as soon as these DRAM banks finish serving the previous requests. Second, the current memory scheduling policies are too strict in terms of fetching only the *exact* values from the DRAM banks. Therefore, an entire DRAM row has to be fetched into the row buffer even if it is poorly reused. We argue that these aggressive and strict policies are sub-optimal towards improving row buffer locality. To this end, we propose the lazy memory scheduler which relaxes the aforementioned constraints by leveraging the fact that several GPGPU applications are latency and error tolerant [134, 50]. First, we demonstrate that delaying the scheduling of memory requests can significantly improve the overall row buffer locality because the memory controller can find more requests that can be scheduled back to back to the same row. Given that several GPGPU applications are latency tolerant, we do not observe notable performance reduction in such applications. To control the performance loss caused by delays, we devise a low-overhead dynamic mechanism that limits the delay by ensuring that utilization of DRAM stays above a certain threshold. Second, we demonstrate that a small fraction of memory requests can cause a large fraction of row activations (i.e., there is non-uniform reuse of row buffers). Therefore, approximating a limited number of requests (bounded by the prediction coverage) can significantly reduce the row energy, without notably degrading the output quality of error-tolerant GPGPU applications. To improve the row buffer locality more effectively under a limited prediction coverage, we devise a low-overhead dynamic mechanism that is able to prioritize the approximation of requests with relatively low row buffer localities.

The rest of this dissertation is organized as follows. Chapter 2 provides a general background for GPU and GPU memory architectures. In Chapter 3, we present PBS for multi-programming in GPUs. In Chapter 4, we present the novel value approximation technique ASAP. In Chapter 5, we present the lazy memory scheduler. In Chapter 6, we present motivational results in order to develop architectural support for flexible data

precisions. Finally, in Chapter 7, we conclude this dissertation and discuss about future work.

Chapter 2

A General Background on Graphics Processing Units (GPUs)

In this chapter, we provide a general background of Graphics Processing Units (GPUs). Specifically, we focus on the GPU organization and operations of GPU memory.

2.1 Baseline GPU Architecture

GPUs achieve high throughput because it is capable of executing a large number of threads concurrently. To facilitate this, GPUs consists of a large number of processing elements (PEs), which are organized in a hierarchical fashion. As shown in Figure 2.1, a group of PEs are clustered together in a core, also known as Streaming Multi-processors (SM) in NVIDIA terminology. The threads from a GPGPU application are uniformly distributed on the SMs at the granularity of a thread-block. Each SM is capable of handling threads from multiple thread-blocks and executes them at the granularity of a warp (or wave-front in AMD terminology). A warp is essentially a collection of (usually 32) individual threads that execute in a lock-step manner on the PEs of the same SM. These threads from the same warp execute a single instruction at a time on different data (i.e., implement SIMD). Multiple warps residing on the same SM facilitate in hiding long memory

latencies via executing in a pipelined and multiplexed manner and hence improving the utilization/throughput of the SM.

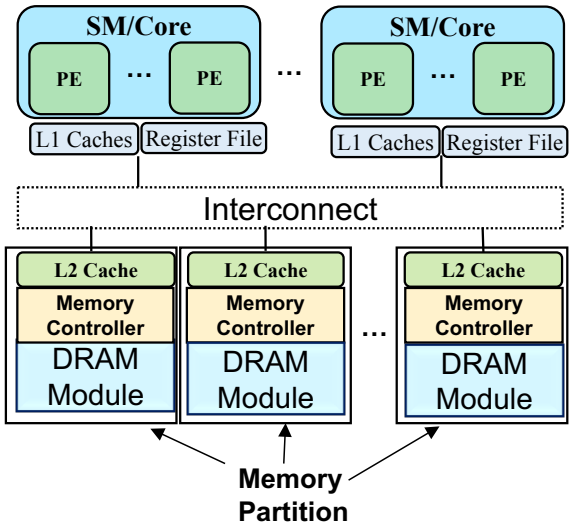


Figure 2.1: Overview of GPU Architecture.

2.2 GPU Memory Organization

We consider the memory hierarchy under a generic GPU architecture consisting of several cores, which are connected to memory partitions via an interconnect as shown in Figure 2.1. In order to support large amount of thread-level parallelism in GPUs, each SM consists of several processing elements (PEs), supported by a large register file (for saving context of a large number of concurrent threads so as to minimize context switch overhead) and all memory partitions manage high bandwidth memories (for fast data access to large number of concurrent threads). Each SM also has a private L1 cache and each memory partition is attached to a shared L2 cache. Each memory partition also has a memory controller that is responsible to schedule L2 cache misses (i.e., memory requests sent from the L2 cache) to the GPU memory.

The data is spread across multiple channels (partitions) for achieving high memory bandwidth. For each channel, the memory operations are performed at the granularity of

DRAM banks. Each bank consists of the cell arrays and a row buffer (sense amplifier) to read data from or write data to the cell arrays [110]. The cell arrays are where the data is stored and consist of many rows (pages) and columns (bits). Each memory channel is also associated with a memory controller, which buffers the pending memory requests in a request pending queue and determines the order to serve them in their destined banks.

2.3 GPU Memory Operations and Scheduling Techniques

GPU memory operations. In order to serve a read or write request to a bank, a whole row in the cell array must first be activated (i.e., opened) to fetch its data into the row buffer. After the pending accesses to the current row are served and before the pending accesses for other rows can be served, the data present in the row buffer must be restored back to the cell arrays to safely keep the correct data values of the row. Finally, a precharge operation also needs to be performed in order to ensure that the next activation operation of a row can be performed successfully. The access energy is dependent on the access type. The row buffer hit request consumes less energy compared to the row buffer miss request. It is because serving the row buffer miss request involves costly operations such as activation, restore and precharge. The energy consumed by these operations (referred to as row energy in this dissertation) contributes significantly to the total DRAM energy consumption [16], especially when the row buffer locality is low (i.e., the ratio of row buffer miss requests is high among all DRAM requests). Note that although we use GDDR5 DRAM model as our example, the row locality concerns are pervasive across all GPU memory technologies (e.g., HBM, HBM2) [16, 83].

GPU Memory Scheduling Techniques. For the purpose of improving the row buffer locality of the GPU memory, several techniques can be applied. First-Row First-Come-First-Serve (FR-FCFS) [97, 146, 98, 12] is one of the commonly employed memory scheduling technique that optimizes for row buffer locality in GPUs. Specifically, FR-FCFS prioritizes row buffer hit requests over other requests, including older ones. If no request is a

row buffer hit, then FR-FCFS prioritizes older requests over younger ones. The open-row policy is often used together with the FR-FCFS scheduler to minimize the row activations. The open-row policy leaves the row open (i.e., does not proceed to restore and precharge operations) as long as no other request is destined to the same bank. Therefore, if the next access is also requesting the same row, the restore, precharge, and activation operations are not necessary. On the contrary, a closed-row policy closes the row (i.e., restores the row and precharges) immediately after the first access to the row. This can be harmful to the row buffer locality, but can also reduce the overall access latency for applications that have low row buffer locality. Also, a large re-order pending request queue can potentially help in reducing the number of row activations by making more requests visible to the FR-FCFS memory scheduler.

Chapter 3

Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management

3.1 Introduction

The idea of co-locating two or more kernels [1] (originating from the same or different applications) has been shown to be beneficial in terms of both GPU resource utilization and throughput [84, 46, 47]. One of the major roadblocks in achieving the maximum benefits of multi-application execution is the difficulty to design mechanisms that can efficiently and effectively manage the application interference in the shared caches and the main memory. Several researchers have proposed different architectural mechanisms (e.g., novel resource allocation [23, 22, 21], cache replacement [95, 94], and memory scheduling [46, 47]), both in CPU and GPU domains, to address the negative effects of the shared-resource application interference on the system throughput and fairness. In fact, a recent surge of works [54, 99, 50, 49, 55] considered the problem of managing inter-thread cache/memory interference even for the single GPU application execution. In particular, the techniques that find

[1] In this chapter, we evaluate workloads where concurrently executing kernels originate from separate applications. Hence, we use the terms kernels and applications interchangeably.

the optimal thread-level parallelism (TLP) of a GPGPU application were found to be very effective in improving the GPU performance both for cache and memory-intensive applications [54, 99]. The key idea behind these techniques is to exploit the observation that executing a GPGPU application with the maximum possible TLP does not necessarily result in the highest performance. This is because as the number of concurrently executing threads increases, the contention for cache space and memory bandwidth also increases, which can lead to sub-optimal performance. Therefore, many of the TLP management techniques proposed to limit the TLP via limiting the number of concurrently executing warps (or wavefronts) to a particular value.

Inspired by the benefits of such TLP management techniques, this chapter delves into the design of new TLP management techniques that can significantly improve the system throughput and fairness by modulating the TLP of each concurrently executing application. To understand the scope of TLP modulation in the context of multi-application execution, we analyzed three different scenarios. First, both applications are executed with their respective best-performing TLP (bestTLP), which are found by statically profiling each application separately by running it alone on the GPU. Note that these individual best TLP configurations can also be effectively calculated using previously proposed run-time mechanisms (e.g., DynCTA [54], CCWS [99]). We call this multi-application TLP combination as ++bestTLP. Second, both applications are executed with their respective maximum possible TLP (maxTLP). We call this multi-application TLP combination as ++maxTLP. Third, both applications are executed with TLP such that they collectively achieve the highest Weighted Speedup (WS) (or Fairness Index (FI))[2] and defined as optWS (or optFI).

Figure 3.1 shows the WS and FI when `BFS2` and `FFT` are executed concurrently under these three aforementioned scenarios[3]. The results are normalized to ++bestTLP. We

[2]We find this combination by profiling 64 different combinations of TLP and picking the one that provides the best WS (or FI).

[3]BFS2_FFT is one of the representative workloads, which demonstrates the scope of the problem. Other workloads are discussed in Section 3.6.

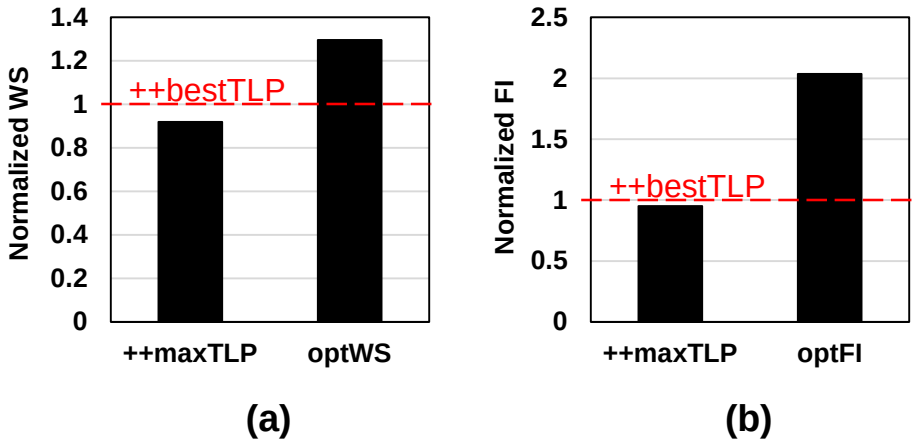


Figure 3.1: Weighted Speedup (WS) and Fairness Index (FI) for `BFS2_FFT`. Evaluation methodology is described in Section 3.2.

find that there is a significant difference in WS and FI between optimal (optWS and optFI) and ++bestTLP combinations, which suggests that blindly using the bestTLP configuration for each application in the context of multi-application execution is a sub-optimal choice. It is because each application in the ++bestTLP or ++maxTLP scenario consumes disproportionate amounts of shared resources as it assumes no other application is co-scheduled. That leads to high cache and memory contention.

Contributions. To our knowledge, this is the first work that performs a detailed analysis of the TLP management techniques in the context of multi-application execution in GPUs and shows that new TLP management techniques, if developed carefully, can significantly boost the system throughput and fairness. In this context, our *goal* is to develop techniques that can find the optimal TLP combination that allows a judicious and good use of all the available shared resources (thereby reducing cache and memory bandwidth contention, and improving WS and FI). To measure such use, we propose a new metric, *effective bandwidth (EB)*, which calculates the effective shared resource usage for each application considering its private and shared cache miss rates and memory bandwidth consumption. We find that a TLP combination that maximizes the total effective bandwidth across all co-located applications while providing a good balance of individual applications' effective bandwidth leads to high system throughput (WS) and fairness (FI). Instead of incurring

the high overheads of an exhaustive search across all the different combinations of TLP configurations that achieve these goals, we propose pattern-based searching (PBS) that cuts down a significant amount of overheads by taking advantage of the trends (which we call *patterns*) in the way application's effective bandwidth changes with different TLP combinations.

Results. Our newly proposed PBS schemes improve the system throughput (WS) and fairness (FI) by 20% and 2×, respectively over ++bestTLP, and 10% and 1.44×, respectively over the recently proposed TLP modulation and cache bypassing scheme (Mod+Bypass [66]) for multi-application execution for GPUs. Also, our PBS schemes are within 3% and 6% of the *optimal* TLP combinations: optWS and optFI, respectively for the evaluated 50 two-application workloads.

3.2 Background and Evaluation Methodology

Multi-application execution. Recent GPUs by AMD [9] and NVIDIA [80] support the execution of multiple tasks/kernels. This advancement led to a large body of work in GPU multiprogramming [84, 46, 8, 35, 132, 47, 118, 135, 66, 86, 63, 72]. Execution of multiple tasks can potentially increase GPU utilization and throughput [84, 46, 8, 131, 119]. While it is possible to execute different independent kernels from the same application concurrently, in this work, we execute different applications simultaneously. To understand how these applications interfere in the shared memory system, each application is mapped to an exclusive set of cores and allowed to use resources beyond the cores (e.g., L2, DRAM). We allocate an equal number of cores to each concurrently executing application. Sensitivity to different core and L2 cache partitioning techniques is discussed in Section 3.6.4.

TLP configurations. We assume that different TLP configurations are implemented at the warp granularity via statically or dynamically limiting the number of actively executing warps [99]. Table 3.2 lists different TLP configurations that are evaluated in the work.

The maximum value of TLP is 24 as the total number of possible warps on a core is 48 and there are two warp schedulers per core. The baseline GPU uses the best-performing TLP (bestTLP) when it executes only one application at a time.

Table 3.1: Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.2 [34] for the complete list.

Core Features	1400MHz core clock, 30 cores, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32684 registers Max. 1536 threads (48 warps, 32 threads/warp)
L1 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
L2 Cache	16-way 256 KB/memory channel (1536 KB 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 (MCs), FR-FCFS scheduling 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock Global linear address space is interleaved among partitions in chunks of 256 bytes [33] Hynix GDDR5 Timing [43], $t_{CL} = 12$, $t_{RP} = 12$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$
Interconnect	1 crossbar/direction (30 cores, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

3.2.1 Evaluation Methodology

We evaluate our proposed techniques on MAFIA [47], a GPGPU-Sim [12] based framework that can execute two or more applications concurrently. The memory performance model is validated across several GPGPU workloads on an NVIDIA K20m GPU [12, 47]. The key parameters of the GPU (Table 3.1) are faithfully simulated. All evaluated metrics are summarized in Table 3.3.

Performance- and Fairness-related Metrics. We report *Weighted Speedup* (WS) [113, 84] and Fairness Index (FI) [47] to measure system throughput and fairness (imbalance of performance slowdowns), respectively. Both metrics are based on individual application slowdowns (SDs) in the workload, where SD is defined as the ratio of performance (IPC) achieved in the multi-programmed environment (IPC-Shared) to the case when it runs alone on the same set of cores with bestTLP (IPC-Alone). The maximum value of WS is equal to the number of applications in the workload assuming there is no constructive

Table 3.2: List of evaluated TLP configurations.

Acronym	Description
maxTLP	Single application is executed with the maximum possible value of TLP.
++maxTLP	Two or more applications are executed concurrently with their own respective maxTLP configurations.
bestTLP	Single application is executed with the best-performing TLP.
++bestTLP	Two or more applications are executed concurrently with their own respective bestTLP configurations.
DynCTA	Single application is executed with DynCTA.
++DynCTA	Two or more applications are executed concurrently with each one using DynCTA.
optWS	Two or more applications are executed concurrently with their own TLP configurations such that Weighted-speedup (WS) is maximized.
optFI	Two or more applications are executed concurrently with their own TLP configurations such that Fairness Index (FI) is maximized.
optHS	Two or more applications are executed concurrently with their own TLP configurations such that Harmonic Weighted-speedup (HS) is maximized.

Table 3.3: List of evaluated metrics.

Acronym	Description
SD	Slowdown. SD = IPC-Shared/IPC-bestTLP.
WS	Weighted Speedup. WS = SD-1 + SD-2 .
FI	Fairness Index. FI = Min(SD-1/SD-2, SD-2/SD-1)
HS	Harmonic Speedup. HS = 1/(1/SD-1 + 1/SD-2).
BW	Attained Bandwidth from Main Memory.
CMR	Combined Miss Rate (MR). CMR = L1MR × L2MR.
EB	Effective Bandwidth. EB = BW/CMR.
EB-WS	EB-based Weighted Speedup. EB-WS = EB-1 + EB-2 .
EB-FI	EB-based Fairness Index. EB-FI = Min(EB-1/EB-2, EB-2/EB-1).
EB-HS	EB-based Harmonic Speedup. EB-HS = 1/(1/EB-1 + 1/EB-2).

interference among applications. An FI value of 1 indicates a completely fair system. We also report Harmonic Weighted Speedup (HS), which provides a balanced notion of both system throughput and fairness in the system [59]. In this work, we refer to all these metrics as SD-based metrics.

Auxiliary Metrics. We consider *Attained Bandwidth* (BW), which is defined as the amount of DRAM bandwidth that is useful for the application (i.e., the useful data transferred over the DRAM interface) normalized to the theoretical peak value of the DRAM bandwidth. We also consider *Combined Miss Rate* (CMR), which is defined as the product of L1 and L2 miss rates. Note that BW and L1/L2 miss rates are separately calculated for each application even in the multi-application scenario. The *Effective Bandwidth* (EB) of an application is the ratio of BW to CMR. It gauges the rate of data delivery to cores by considering how the bandwidth achieved from DRAM is amplified by the caches (e.g., a

Table 3.4: GPGPU application characteristics: *(A) IPC@bestTLP: The value of IPC when the application executes with bestTLP, (B) EB@bestTLP: The value of the effective bandwidth when the application executes with bestTLP, (C) Group information: Each application is categorized into one of the four groups (G1-G4) based on their individual EB values.*

Abbr.	IPC	EB	Group	Abbr.	IPC	EB	Group
LUD [17]	40	0.13	G1	LIB [79]	211	0.93	G2
NW [17]	31	0.21	G1	LUH [53]	87	1.08	G2
HISTO [115]	471	0.29	G1	SRAD [17]	229	1.19	G3
SAD [115]	651	0.31	G1	CONS [79]	397	1.35	G3
QTC [20]	26	0.59	G2	FWT [79]	195	1.41	G3
RED [20]	180	0.70	G2	BP [17]	580	1.42	G3
SCAN [20]	151	0.72	G2	CFD [17]	95	1.49	G3
BLK [79]	457	0.79	G2	TRD [20]	238	1.67	G3
HS [17]	578	0.79	G2	FFT [115]	261	1.77	G4
SC [17]	173	0.80	G2	BFS2 [79]	18	1.78	G4
SCP [79]	307	0.85	G2	3DS	457	2.19	G4
GUPS	9	0.87	G2	LPS [79]	410	2.20	G4
JPEG [79]	330	0.92	G2	RAY [79]	328	3.12	G4

miss rate of 50% effectively doubles the bandwidth delivered.). We append the application ID (or application's abbreviation (Table 3.4)) to the end of these metrics to denote per-application metrics (e.g., CMR-1 is the combined miss rate for application 1 and EB-BLK is the effective bandwidth for CUDA Blackscholes Application).

EB-based Metrics. In addition to standard SD-based metrics that we finally report, our proposed techniques take advantage of runtime *EB-based metrics*. These metrics are calculated in a similar fashion as SD-based metrics with the difference that they use EB instead of SD. For example, EB-WS is defined as the sum of EB-1 and EB-2. More details are in Table 3.3 and in upcoming sections.

Application suites. For our evaluations, we use a wide range of GPGPU applications with diverse memory behavior in terms of cache miss rates and memory bandwidth. These applications are chosen from Rodinia [17], Parboil [115], CUDA SDK [12], and SHOC [20] based on their effective bandwidth (EB) values such that there is a good spread (from low to high – see Table 3.4). In total, we study 50 two-application workloads (spanning 26 single applications) that exhibit the problem of multi-application cache/memory interference.

3.3 Analyzing Application Resource Consumption

In this section, we first discuss the effects of TLP on various single-application metrics followed by a discussion on succinctly quantifying those effects.

3.3.1 Understanding Effects of TLP on Resource Consumption

GPU applications achieve significant speedups in performance by exploiting high TLP. Therefore, GPU memory has to serve a large number of memory requests originating from many warps concurrently executing across different GPU cores. Consequently, memory bandwidth can easily become the most critical performance bottleneck for many GPGPU applications [37, 50, 32, 96, 54, 47, 129, 128]. Many prior works have proposed to address this bottleneck by improving the bandwidth utilization and/or by effectively using both private and shared caches in GPUs via modulating the available TLP [54, 99]. These modulation techniques strive to improve performance via finding the level of TLP such that it is neither too low so as not to under-utilize the on/off-chip resources nor too high so as not to cause too much contention in caches and memory leading to poor cache miss rates and row-buffer locality, respectively. To understand this further, consider Figure 3.2 (a–c), which shows the impact of the change in TLP (i.e., different levels of TLP) for `BFS2` on IPC, BW, and CMR. These metrics are normalized to that of bestTLP (best performing TLP for `BFS2` is 4). We observe that with the initial increase in TLP, both BW and IPC start to increase rapidly. However, at higher TLP, the increase in CMR starts to negate the benefits of high BW, ultimately leading to decrease in performance. For example, when TLP limit increases from 4 to 24, BW increases by 2.7×, but CMR also increases by 2.9× leading to drop in performance. In such cases, the increase in TLP not only hampers performance but also consumes unnecessary memory bandwidth. *In summary, we conclude that the changes in performance with different TLP configurations are directly related to changes in cache miss-rate and memory bandwidth resource consumption.*

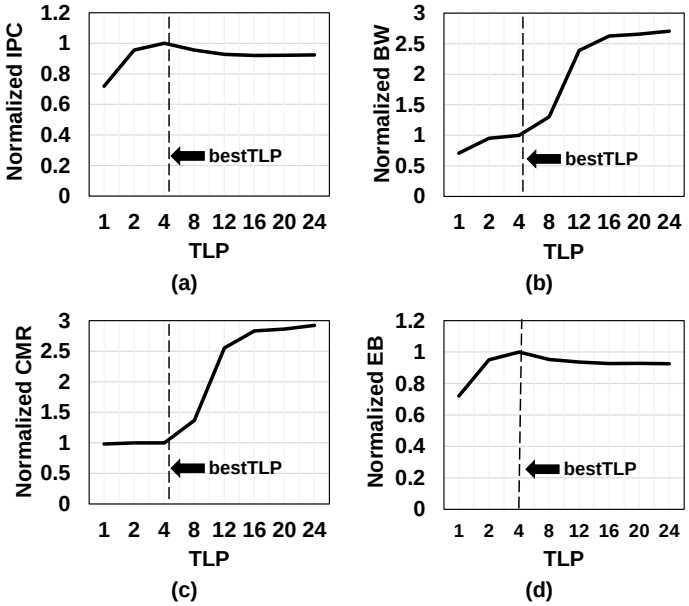


Figure 3.2: Effect of TLP on performance and other metrics for `BFS2`.

3.3.2 Quantifying Resource Consumption

To measure such resource consumption via a single combined metric, we introduce a new metric called as *effective bandwidth (EB)*. It is defined as the ratio of bandwidth to miss rate, and is calculated based on the level of hierarchy under consideration as depicted in Figure 3.3. For example, the value of EB observed by L1 (B) is defined as the ratio of BW (A) to the L2 miss rate. Similarly, the value of EB observed by the core (C) is defined as the ratio of EB observed by L1 (B) to the L1 miss rate. This value is also equivalent to the ratio of BW (A) to CMR. The EB observed by the core essentially measures how well the DRAM bandwidth is utilized. It also considers the usefulness of the caches in amplifying the performance impact of the attained DRAM bandwidth, where the amplification is based on the combined miss rate. If the CMR is 1, it implies that caches are not useful and cores will obtain the same return bandwidth that is attained from the DRAM. Therefore, EB is equal to BW for cache insensitive applications (e.g., BLK). On the other hand, a lower CMR would allow cores to obtain more return bandwidth than what is attained from the DRAM, which is the case for cache-sensitive applications (e.g., `BFS2`). In an ideal case, when combined miss rate is zero, the effective bandwidth observed

by the core would be equal to the L1 cache bandwidth of the GPU system[4].

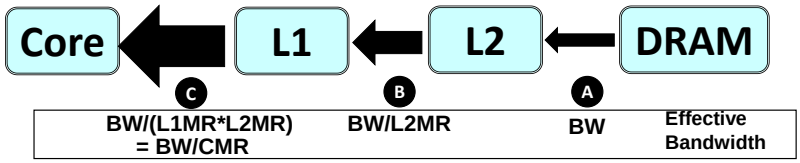


Figure 3.3: Effective Bandwidth at different levels of the hierarchy. For brevity, we show only one core (with attached L1 cache) and one L2 cache partition.

Connecting back to Figure 3.2, we observe that effective bandwidth observed by the core (C) and performance closely follow each other (Figure 3.2(d)) with changes in TLP. This concludes that the impact of changes in TLP on performance can be accurately estimated by directly measuring only the changes in EB, *without* the need to consider any architecture-independent parameters such as compute-to-memory ratio of an application. Although we demonstrate the validity of these conclusions for `BFS2`, we verified that these conclusions hold true for all the considered applications[5] listed in Table 3.4.

To substantiate the conclusions analytically, we revisit our prior work [47], which showed that GPU performance (IPC) is proportional to the ratio of BW to L2 misses per instruction (MPI),

$$IPC \propto \frac{BW}{L2MPI} \tag{3.1}$$

which in turn is proportional to the ratio of BW to $r_m \times CMR$, where r_m is the ratio of memory instructions to the total number of instructions. r_m is an application-level property[6]. Therefore, IPC is proportional to the ratio of EB to r_m, where

$$IPC \propto \frac{BW}{r_m \times L2MR \times L1MR} \propto \frac{BW}{r_m \times CMR} \propto \frac{EB}{r_m} \tag{3.2}$$

[4]It is assumed that the return packets from the memory system do not bypass the L1 cache.

[5]Applications that make heavy use of the software-managed scratchpad memory observe higher EB at the cores due to additional bandwidth from the scratchpad memory. Because the scratchpad memory is not susceptible to contention due to high TLP in our evaluation setup, our calculations do not consider the bandwidth provided by the scratchpad to the core.

[6]Arithmetic intensity (i.e., ratio between compute to memory instructions) of an application is equal to (1-r_m)/r_m.

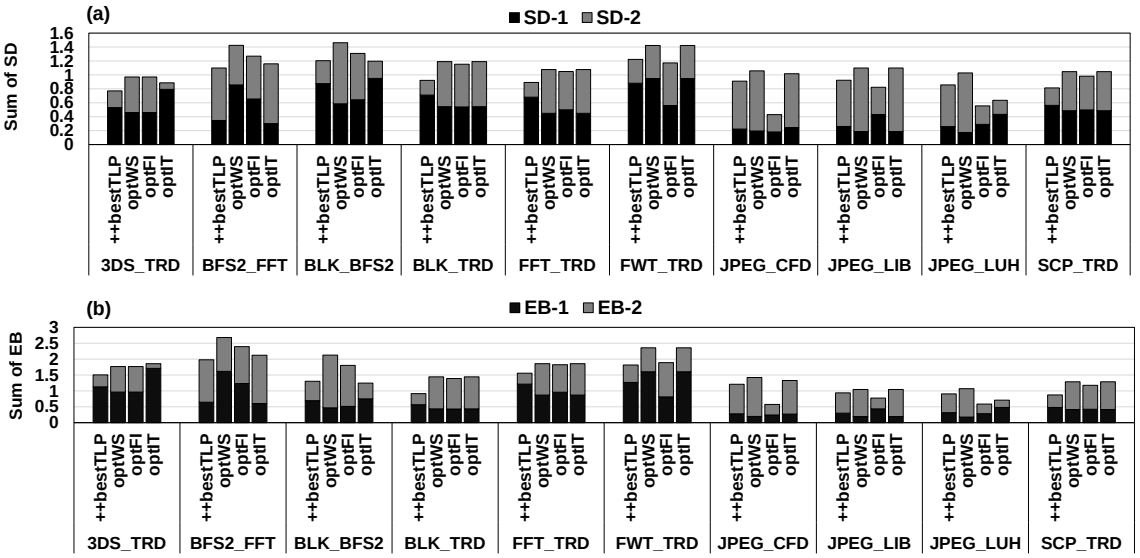


Figure 3.4: Effect of different TLP combinations on application: a) slowdown and b) effective bandwidth.

We conclude that EB is able to effectively measure the good and judicious use of cache and memory bandwidth resources and is optimal at the bestTLP configuration.

3.4 Motivation and Goals

As the total shared resources are limited, understanding how these resources should be allocated to the concurrent applications for maximizing system throughput and fairness is a challenging problem. To this end, we consider the TLP of *each* application as a knob to control its shared resource allocation. Although previously proposed TLP modulation techniques (e.g., DynCTA [54] and CCWS [99]) have been shown to be effective in optimizing TLP for single-application execution scenarios, they rely only on different kinds of *per-core heuristics* (e.g., latency tolerance, IPC, cache/memory contention) and do not consider the shared resource consumption of co-scheduled applications. In other words, each application under such TLP configurations (including bestTLP) attempts to maximize its *own* effective bandwidth, ultimately taking disproportionate amount of shared resources. This causes too much contention in caches and memory, ultimately hampering

the system-wide metrics such as system throughput and fairness.

To understand this further, consider Figure 3.4(a), which shows the WS of 10 representative workloads under ++bestTLP and opt TLP combinations[7]. WS for each two-application workload is split into its respective slowdowns for each application (SD-1 and SD-2). We find that there is a significant gap between ++bestTLP and optWS for all workloads. For example, in `BFS2_FFT` and `BLK_BFS2`, this difference is 29% and 21%, respectively. In terms of fairness, the gap is up to 2× (observed from the imbalance between SD values in ++bestTLP compared to balanced values in optFI) in `BFS2_FFT`. *We conclude that new TLP management techniques are needed to close this system throughput and fairness gap.*

Analysis of Weighted Speedup. We find that a TLP management scheme that optimizes for EB-based metrics is useful in improving system performance and fairness. To understand this analytically, we first focus on system throughput (weighted speedup) via equations 3.3, 3.4, and 3.5. First, let us define IPC alone ratio (IPC_{AR}) and EB alone ratio (EB_{AR}) of two applications (App-1 and App-2) when each of them separately execute alone on the GPU:

$$\begin{aligned} IPC_{AR} &= \frac{IPC_{Alone-1}}{IPC_{Alone-2}} \\ EB_{AR} &= \frac{EB_{Alone-1}}{EB_{Alone-2}} \end{aligned} \tag{3.3}$$

Next, as WS is the sum of slowdowns of co-scheduled applications, we derive the following Equation:

$$\begin{aligned} WS &= \frac{IPC_{Shared-1}}{IPC_{Alone-1}} + \frac{IPC_{Shared-2}}{IPC_{Alone-2}} \\ WS &\propto IPC_{Shared-1} + IPC_{Shared-2} \times IPC_{AR} \end{aligned} \tag{3.4}$$

Finally, with the help of Equation 3.2,

$$WS \propto EB_{Shared-1} + EB_{Shared-2} \times EB_{AR} \tag{3.5}$$

We observe that WS is a function of instruction throughput (sum of IPCs of individual applications) and also a function of EB-WS (sum of EBs of individual applications).

[7]The opt combinations are chosen via an exhaustive search of 64 different TLP combinations.

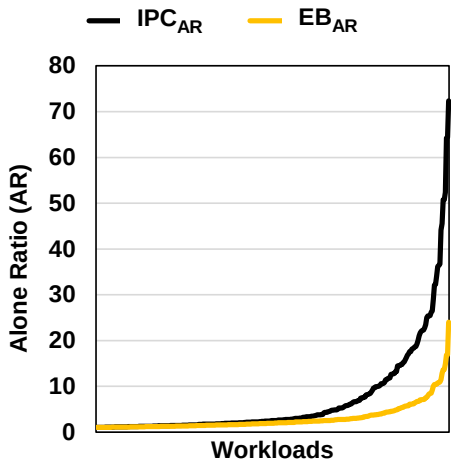


Figure 3.5: IPC_{AR} vs. EB_{AR}

However, maximizing IT or EB-WS will lead to sub-optimal WS, if IPC_{AR} and EB_{AR} are much greater than 1. This is due to the bias caused by alone ratios (IPC_{AR} and EB_{AR}) towards one of the co-scheduled applications. On average across all possible two-application workloads formed using the evaluated 26 applications, we find that EB_{AR} is much lower than IPC_{AR}, as shown in Figure 3.5[8]. Therefore, we choose EB to optimize system-wide metrics.

To substantiate this claim quantitatively, Figure 3.4(b) shows EB-WS for each workload along with its respective EBs for each application (EB-1 and EB-2). We make the following two observations:

• **Observation 1:** The TLP combination that provides the highest sum of EB (EB-WS) provides also the highest system throughput (WS). This trend is present in almost all evaluated workloads (a few exceptions are discussed in Section 3.6). We find this observation interesting as it means that optimizing for EB-WS is likely to improve WS (SD-based metric) as discussed earlier via Equation 3.5. Also, EB-WS metric *does not* incorporate any alone-application information making it easier to calculate directly in a multi-application environment.

[8] As the alone ratio bias can be towards any one of the co-scheduled applications, we show max(M1/M2, M2/M1), where M is IPC_{AR} or EB_{AR}.

• **Observation 2:** The TLP combination (optIT) that provides the highest instruction throughput (IT) (i.e., sum of IPCs across all the concurrent applications) *does not* always provide the highest WS and FI (e.g., in `BFS2_FFT`, `BLK_BFS2`). It implies that a mechanism that attempts to maximize IT may not be optimal to improve system throughput as analytically demonstrated earlier.

Analysis of Fairness. Extending the above discussion for fairness, we also find that EB-FI correlates well with the SD-based FI (i.e., differences in SDs in a workload is correlated with those of EBs.). Therefore, a careful balance of effective bandwidth allocation among co-scheduled applications can lead to higher fairness in the system (as demonstrated by optFI in Figure 3.4(b)). However, there are a few outliers (e.g., `BLK_TRD`) (i.e., the difference between EB-1 and EB-2 is much larger than that of SD-1 and SD-2 breakdowns). The main reason behind these outliers is EB_{AR}, which can still be larger than one (Figure 3.5) leading to a bias towards one of the applications. To reduce the outliers and increase the correlation between EB-FI and SD-FI, we appropriately scale the EB with the alone EB information of each application. These *scaling-factors* either can be supplied by the user or can be calculated at runtime. In the former case, each application uses the average value of alone EB for the *group* it belongs to (see Table 3.4). In the latter case, each application uses the value of EB when it executes alone and uses bestTLP. As we cannot get this information unless we halt the other co-running applications, we approximate it by executing the co-runners with the least amount of TLP (i.e., 1) so that they induce the least amount of interference possible. Note that in our evaluated workloads, we did not find the necessity of using these scaling factors while optimizing WS because of the limited number of outliers (Section 3.6). However, we do use them to further optimize fairness and harmonic weighted speedup.

In summary, we conclude that maximizing the total effective bandwidth (EB-WS) for all the co-runners is important for improving system throughput (WS). Further, a better balance between the effective bandwidth (determined by EB-FI) of the co-scheduled applications is required for higher fairness (FI).

3.5 Pattern-based Searching (PBS)

In this section, we provide details on the proposed TLP management techniques for multi-application execution followed by the implementation details and hardware overheads.

3.5.1 Overview

Our goal is to find the TLP combinations that would optimize different EB-based metrics. A naive method to achieve this goal is to periodically take samples for all the possible TLP combinations over the course of workload execution and ultimately choose the combination that satisfies the optimization criteria. However, that would incur significant runtime overheads in terms of performance. Instead of high-overhead naive searching, we take advantage of the following guidelines and *patterns* to minimize the search space for optimizing the EB-based metrics.

Guideline-1. The EB-based metrics are sub-optimal when a particular TLP combination leads to under-utilization of resources (e.g., DRAM bandwidth). Therefore, for obtaining the optimal system throughput, *it is important to choose a TLP combination that does not under-utilize the shared resources.*

Guideline-2. When increasing an application's TLP level, its EB starts to drop only when the increase in its BW can no longer compensate for the increase in its CMR (i.e., EB at its inflection point). Therefore, *it is important to choose a TLP combination that would not overwhelm resources as it is likely to cause sharp drops in one or all applications' EB, leading to inferior system throughput and fairness.*

Patterns. In all our evaluated workloads, we find that when resources in the system are sufficiently utilized, distinct inflection points emerge in EB-based metrics. These inflection points tend to appear consistently at the same TLP level of an application, regardless of the TLP levels of the other co-running application. We name this consistency of the inflection points as *patterns*. Moreover, the sharpest drop in EB-based metrics is usually attributed to one of the co-running applications, namely the *critical* application.

High-level Searching Process. We utilize the observed *patterns* to reduce the search space of finding the optimal TLP combination. We first ensure that the TLP values are high enough to sufficiently utilize the shared resources. Subsequently, we find the *critical* application and its TLP value that leads to the inflection point in the EB-based metrics. Once the TLP of the *critical* application is fixed, the TLP value of the non-critical application is tuned to further improve the EB-based metric. Because of the existence of the *patterns*, we expect that the *critical* application's TLP still leads to a inflection point regardless of the changes in TLP of the non-critical application. This final stage of tuning is similar to the one application scenario, where tuning of TLP is performed for optimizing the effective bandwidth (Section 3.3.1). We find that this searching process based on the *patterns* (i.e., *pattern-based searching (PBS)*) is an efficient way (reduces the number of TLP combinations to search) to find the appropriate TLP combination targeted for optimizing a specific EB-based metric. In this context, we propose three PBS mechanisms: PBS-WS, PBS-FI, and PBS-HS to optimize for Weighted Speedup (WS), Fairness Index (FI), and Harmonic Weighted Speedup (HS), respectively.

3.5.2 Optimizing WS via PBS-WS

As per our discussions in Section 3.4, our goal is to find the TLP combination that would lead to the highest total effective bandwidth (EB-WS). We describe this searching process for two-application workloads, however, it can be trivially extended for three or more application workloads as described later in Section 3.6.4.

Consider Figure 3.6(a) that shows the EB-WS for the workload `BLK_TRD`. We show individual EB values of each application, that is, EB-BLK and EB-TRD in Figure 3.6(b). The *pattern* demonstrating sharp drop in EB-WS (i.e., inflection points) is in the shaded region. We follow the high-level searching process described earlier. First, when both applications execute with TLP values of 1, the EB-WS is low (0.55) due to low DRAM bandwidth utilization (29%, not shown). Therefore, as per Guideline-1, this TLP combination is not desirable.

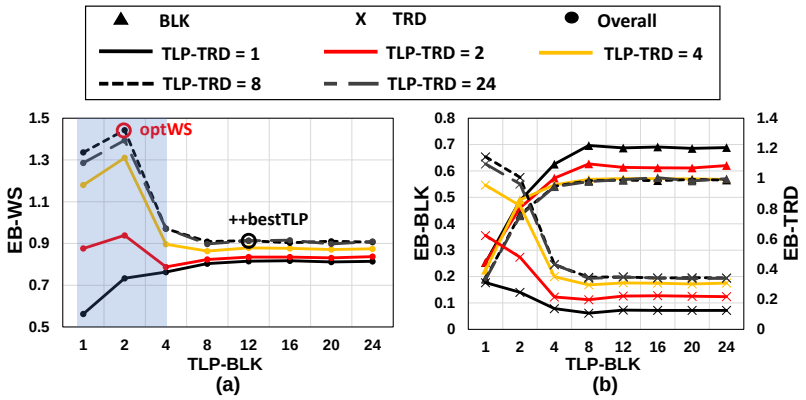


Figure 3.6: Illustrating the patterns observed in `BLK_TRD`.

Second, we focus on finding the *critical* application. The process is as follows. We execute each application with TLP of 1, 2, 4, 8 etc. by keeping the TLP of the other application to be fixed at 24. The TLP value of 24 ensures that the GPU system is not under-utilized. This process is repeated for every application in the workload. The application that exhibits a larger drop in EB-WS is *critical* and its TLP is fixed. We decide `BLK` as the *critical* application as it affects EB-WS the most (Figure 3.6(a)) – the sharp drop in EB-WS after TLP-BLK=2 is prominent.

Third, the next step is to tune the TLP for the non-critical application to reduce the contention and further improve the EB-WS. The searching for TLP of the non-critical application is stopped when the EB-WS no more increases. Therefore, in our example, after fixing TLP-BLK to 2, we start tuning the TLP-TRD to further optimize the EB-WS. The searching process stops at the TLP-TRD = 8, leading to the optimal TLP combination to be (2,8), which is also the optWS[9]. As evident, the whole search process requires only a few samples and does not require an exhaustive search across all combinations.

[9]There is a possibility that this final process of tuning can free up just enough resources so that the infection point of EB-WS shifts to the right (i.e., *pattern* does not hold), leading to a sub-optimal TLP combination. However, we never observed such a scenario in our experiments.

3.5.3 Optimizing Fairness via PBS-FI

In this scheme, our goal is to find a TLP combination that would lead to a better balance between the individual effective bandwidth of the co-scheduled applications. Therefore, we strive to find the TLP combination that would lead to the highest EB-FI. For all the evaluated workloads, we find that a *pattern* also exists in their EB-FI curves (not shown). As a result, we are able to first find the *critical* application that affects the EB-FI the most, followed by tuning the TLP of other non-critical applications.

To intuitively understand the searching process, we study the EB-difference between two applications and plot this difference against TLP to understand the trends in them. A lower absolute value of the difference indicates a fairer system (higher EB-FI) as the EB values of applications are similar (see Section 3.4).

Consider the example of `BLK_TRD` in the context of fairness. Figure 3.7 (a) and (b) show two different views of the same data related to EB-difference – one being TLP-BLK as x-axis and curves representing iso-TLP-TRD states (Figure 3.7 (a)), and vice versa for the second view (Figure 3.7 (b)).

We examine the effect on EB-difference when the TLP of a particular application changes with the TLP of the other application fixed at 24. This process is repeated for every application in the workload. The application that causes larger changes in EB-difference is considered to be *critical*. For example, in Figure 3.7 (a) and (b), `BLK` is more *critical* than `TRD` because changes in TLP-BLK of `BLK` induces larger changes in the EB-difference, when TLP-TRD is kept constant at 24 (Figure 3.7 (a)). We then keep the *critical* application's TLP fixed (e.g., TLP-BLK is 2), where EB-difference is near zero. After fixing, we start to tune the TLP of the other application (i.e., `TRD`). The searching is stopped when the lowest absolute EB-difference is found.

We observe that this searching process stops when TLP-TRD is 4 (Figure 3.7 (b)). However, optFI is (2,20) instead of (2,4). This difference is caused because the EB-FI uses scaling factor that is approximately calculated by either sampling or user-given group

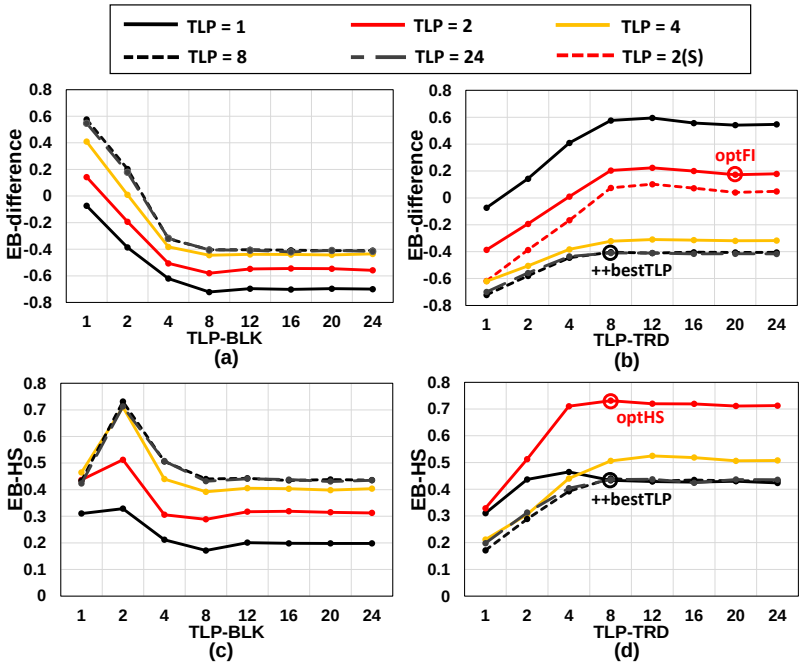


Figure 3.7: Illustrating the working of PBS-FI (a & b) and PBS-HS (c & d) schemes for BLK_TRD.

information. We plot the curve (dashed red line, Figure 3.7 (b)) with the exact scaling factor (Table 3.4). We are able to locate the correct optFI (2,20) as that point is the closest to 0 on the dashed red line.

3.5.4 Optimizing HS via PBS-HS

In this scheme, our goal is to optimize EB-WS. We again take advantage of the *patterns* and observations discussed earlier in the context of PBS-WS and PBS-FI. For all the evaluated workloads, we find that a *pattern* exists in their EB-HS curves. Therefore, we are able to first find the *critical* application that affects the EB-HS the most, followed by TLP tuning of the non-critical application.

Consider the example of BLK_TRD in the context of HS. Figure 3.7 (c) and (d) show two different views of the same data related to EB-HS metric – one being TLP-BLK as x-axis and curves representing iso-TLP-TRD states (Figure 3.7 (c)), and vice versa for the second view (Figure 3.7 (d)). PBS-HS starts with examining the effect on EB-HS when

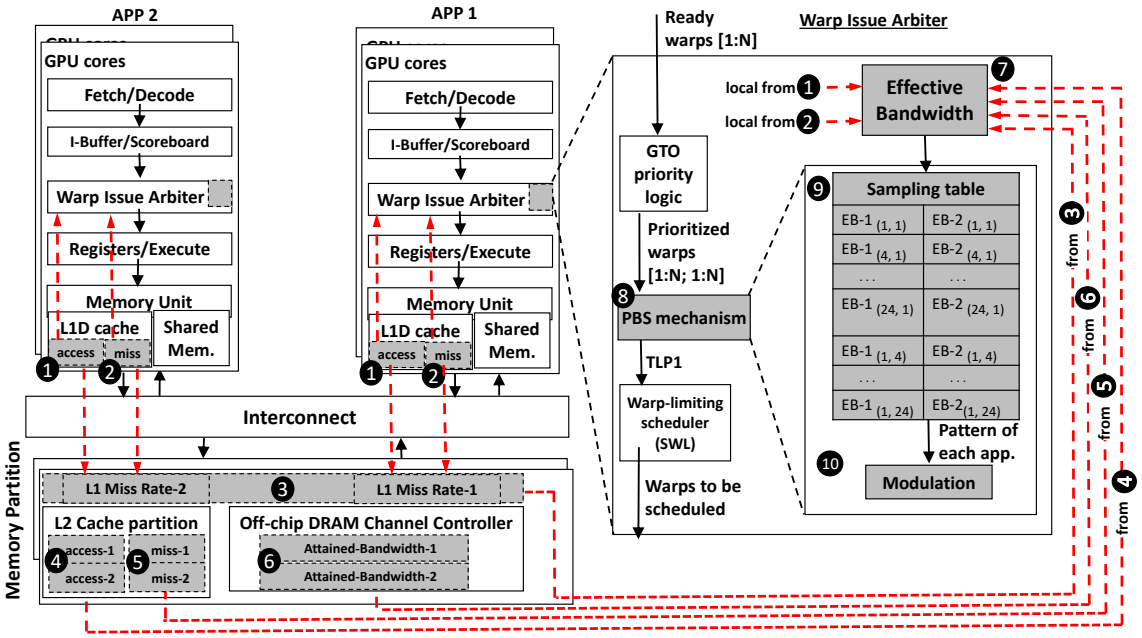


Figure 3.8: Proposed hardware organization. Additional hardware is shown via shaded components and dashed arrows.

the TLP of a particular application changes and the TLP of the other application is fixed at 24. This process is repeated for every application in the workload. The application that causes larger drops in EB-HS value is considered to be *critical*. For example, in Figure 3.7 (c) and (d), *BLK* is again the *critical* application as it affects the EB-HS the most (larger drop in TLP-TRD=24 curve as TLP-BLK increases, Figure 3.7(a)). After fixing TLP-BLK to be 2, we start tuning TLP-TRD so as to further optimize the EB-HS. The searching process stops at TLP-TRD=8, leading to the optimal combination of (2,8), which is exactly the optHS.

3.5.5 Implementation Details and Overheads

Our mechanism requires periodic sampling of cache miss rates at L1 and L2, and memory bandwidth utilization. In our experiments, we observe uniform miss rate and bandwidth distribution among the memory partitions and uniform L1 miss rates across cores that execute the same application. Therefore, to calculate EB in a low-overhead manner, instead

of calculating EB by collecting information from every core and L2/memory partition, we collect: a) L1 miss rate information only from one core per application, and b) attained bandwidth and L2 miss rate information of every application only from one of the L2/memory partitions.

Figure 3.8 shows the architectural view of our proposal. First, after each sampling period, we use the total number of L1 data cache accesses (❶) and misses (❷), from each designated core, and calculate the miss rate of the application. The calculated miss rates are sent through the interconnect to the designated memory partition and stored in their respective buffers (❸). Then, the miss rate of each application, L2 cache accesses (❹) and misses (❺), and attained bandwidth (❻) from the designated memory partition are forwarded to each core to be used along with the locally collected L1 data. Such data is used, per core, to calculate EB (❼). The calculated EB values are then fed to our PBS mechanism (❽), which resides inside the warp issue arbiter within each core, and stored in a small table (❾). In this table, each line represents the EB of both applications, corresponding to the TLP combination used for the current sampling period (indicated by the subscript). After sampling, our PBS mechanism extracts the pattern from each application and then changes the TLP value for one application accordingly. The next step, modulation (❿), varies the TLP value of the other application to maximize the relevant EB-based metric. Finally, the calculated TLP is sent to the warp-limiting scheduler.

We break down overhead in terms of storage, computation, and communication. In terms of storage, two 12-bit registers per core, and three 12-bit registers and one 15-bit register per memory partition are required to track per-application L1 miss rate, L2 miss rate, and BW, respectively. The sampling table needs 60 bytes. In terms of computation, the sampled data is fed to the PBS mechanism module (❽), which performs a simple search over the 16×2 samples collected over the sampling window. In terms of communication, using a crossbar, the designated memory partition relays the collected information (12 bits $\times$ 6 + 15 bits $\times$ 2 = 102 bits) to the cores every sampling window. We conservatively assume that the counter values are sent to the cores with a latency of 20 cycles.

All the runtime overheads are modeled in the PBS results presented in Section 3.6. We empirically find that a monitoring interval of 3000 cycles for each TLP combination searched via PBS is sufficient as trends do not change significantly beyond 3000 cycles. The PBS is re-started when any kernel is re-launched.

3.6 Experimental Evaluation

In this section, we evaluate our proposed PBS schemes (Section 3.5) and compare them against ++bestTLP, opt (optWS, optFI, optHS), and the following additional schemes:

Brute-Force (BF). BF scheme performs an offline exhaustive search across all the possible TLP combinations (64) to find the one that provides the best EB-based metric. Therefore, BF has three different versions: BF-WS, BF-FI, and BF-HS that optimizes EB-WS, EB-FI, and EB-HS, respectively. BF schemes provide a good estimate of the potential of improving the SD-based metrics (the ones we finally report) via optimizing EB-based runtime metrics. Note that opt schemes are also brute-force but instead they perform an exhaustive search to find the best SD-based metric.

PBS (Offline). PBS-Offline schemes follow the exact same procedure as previously described in the PBS schemes (Section 3.5) but do not consider: a) any runtime overheads, and b) dynamic changes in interference across different kernel executions in the workload. We consider this comparison point to decouple the runtime effects from the inherent benefits of the proposed schemes. Similar to PBS, PBS-Offline also has three versions: PBS-WS (Offline), PBS-FI (Offline), and PBS-HS (Offline).

Mod+Bypass [66]. In addition to ++DynCTA, we compare the PBS mechanisms against the recently proposed TLP management mechanism Mod+Bypass [66] for multi-application scenario. They use both CTA modulation and cache bypassing mechanism to enhance the system throughput.

3.6.1 Effect on Weighted Speedup

Figure 3.9 shows the impact of different schemes on the WS for 10 representative workloads (out of 50 evaluated) along with Gmean across evaluated workloads. The results are normalized to the WS obtained under ++bestTLP. Six observations are in order. First, on average, benefits of BF-WS are as good as optWS (within 2%) implying that optimizing EB-WS is a good candidate to improve SD-based WS. However, as EB-WS is a proxy for WS and it may not work for all workloads as discussed in Section 3.4. As the number of outliers are very few, we did not consider scaling factors for optimizing WS. Second, PBS-WS (Offline) overall performs as good as optWS. This shows the inherent effectiveness of PBS in finding the TLP combination that results in higher WS over ++bestTLP.

Third, on average, PBS also performs as good as the PBS-WS (Offline). Note that PBS-WS (Offline) offers a trade-off. As it is a static technique it does not incur runtime overhead but also cannot adapt to different runtime interference patterns for locating better TLP combination within the same workload execution. Therefore, we observe that the benefits of PBS-WS can be: 1) similar to PBS-WS (Offline) (e.g., `BLK_TRD`), where the runtime benefits cancel out with the overheads; 2) worse than PBS-WS (Offline) (e.g., `FWT_TRD`, where the runtime overheads hamper the WS; or 3) better than PBS-WS (Offline) (e.g., `3DS_TRD`, `BLK_BFS2`), where the runtime tuning of TLP combination provides benefits. To illustrate the last point, Figure 3.11(a) shows the dynamic changes in TLP (TLP-BLK above and TLP-BFS2 below) over the course of `BLK_BFS2` execution. The shaded areas represent the sampling period including the time during which the decision cannot be taken because the execution time of the kernel is too short. As expected and discussed before in Section 3.3, (2,2) is the most preferred TLP combination for `BLK_BFS2` and is chosen for the longest duration of time. Other TLP combinations are chosen during other time intervals to boost the WS further.

Fourth, ++DynCTA provides additional benefits over ++bestTLP (7% on average) because of its ability to adapt under a shared environment. However, it is still far from PBS

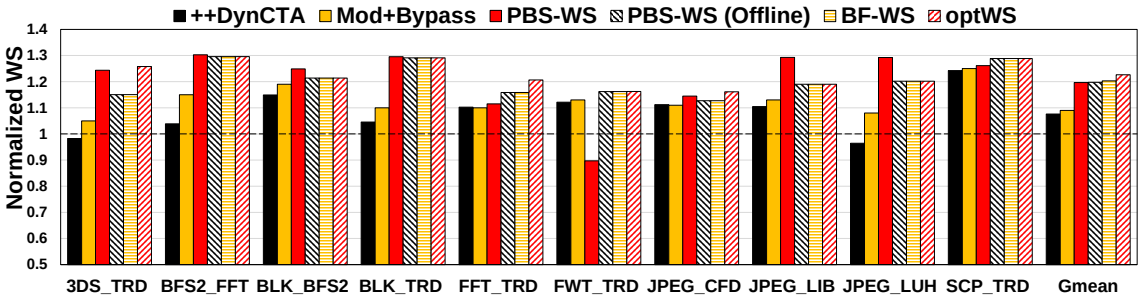


Figure 3.9: Impact of our schemes on Weighted Speedup. Results are normalized to ++bestTLP.

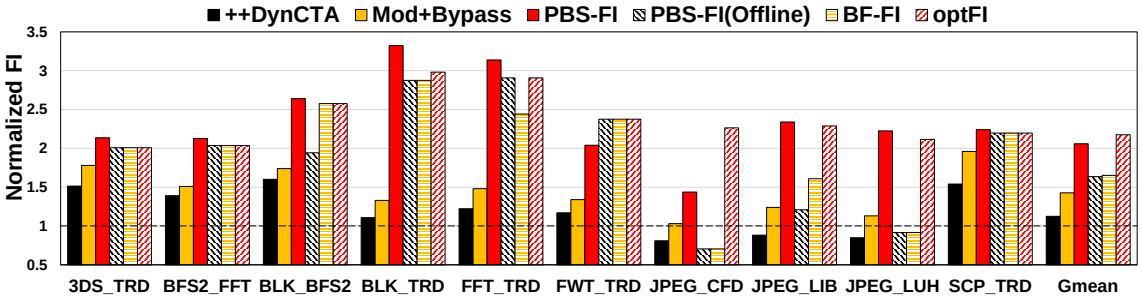
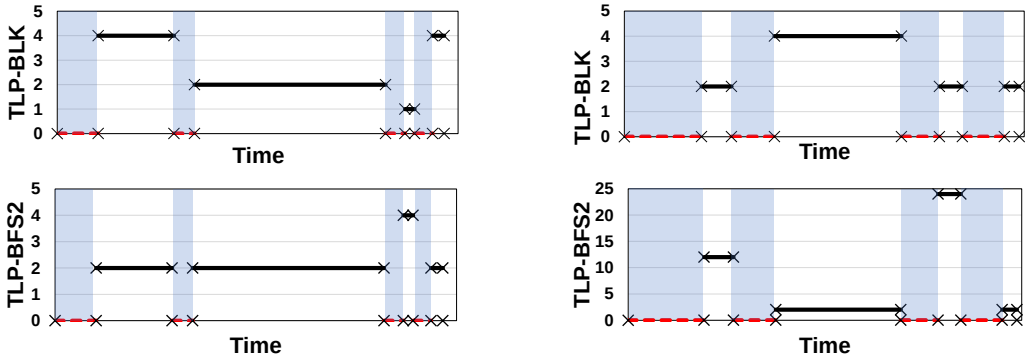


Figure 3.10: Impact of our schemes on Fairness. Results are normalized to ++bestTLP.

and other schemes as ++DynCTA attempts to enhance the performance based on application's local information and hence can overwhelm the memory system. Fifth, Mod+Bypass technique helps in improving the performance further over ++DynCTA mainly because it also bypasses the application that does not take advantage of caches, thereby reducing the cache contention. However, this mechanism is still far from optWS as it does not consider the memory bandwidth consumption and the combined effects of TLP modulation. Finally, PBS-WS performs significantly better (20%, on average) than the ++bestTLP because of the reasons extensively discussed in Section 3.3 and Section 3.5. `FWT_TRD` is the only exception as PBS-WS is not able to find the optimal TLP combination due to a smaller sampling period.



(a) Optimizing WS (b) Optimizing FI

Figure 3.11: Effect of changes in TLP over time for BLK_BFS2 with: a) PBS-WS and b) PBS-FI.

3.6.2 Effect on Fairness Index

Figure 3.10 shows the impact of different schemes on FI for 10 representative workloads (out of 50 evaluated) along with Gmean across evaluated workloads. The results are normalized to the FI obtained under ++bestTLP. Five observations are in order. First, on average, benefits of BF-FI are not as close as optFI implying that runtime optimizations play an important role in achieving high fairness. To evaluate the impact of scaling factor, we calculated BF-FI both using grouping as well as sampling information (Section 3.4). We find that BF-FI calculated using grouping information is 16% (not shown) better in FI, averaged across all workloads. However, the grouping information needs to be supplied by the user. If exact scaling factors are used (Table 3.4), BF-FI is close to optFI as expected. For a fair comparison, Figure 3.10 shows the sampling-based BF-FI for comparisons against other dynamic counterparts.

Second, PBS-FI (Offline) overall performs as good as BF-FI implying that the scheme itself is effective in providing high fairness. Third, PBS-FI is able to capture the runtime effects well and is better than PBS-FI (Offline) in many workloads. As an example, Figure 3.11(b) shows the dynamic changes in TLP (TLP-BLK above and TLP-BFS2 below) over the course of `BLK_BFS2` execution. The shaded areas represent the sampling period. We observe that despite higher sampling overhead (as it includes additional sampling to

calculate the scaling factor), PBS-FI is able to provide much higher benefits than other schemes. It is because TLP combination of (4,2) allowed to reduce the slowdown of BLK while preserving the slowdown for BFS2 when it was executing non-cache sensitive kernels. Fourth, ++DynCTA and Mod+Bypass provide additional benefits over ++bestTLP (12% and 42% on average, respectively) because of their ability to adapt under a shared environment. However, both ++DynCTA and Mod+Bypass themselves are not designed to improve fairness in the multi-application environment and only focus on performance. Finally, PBS-FI performs significantly better (2×, on average) than the ++bestTLP because of the reasons discussed before.

3.6.3 Effect on Harmonic Weighted Speedup

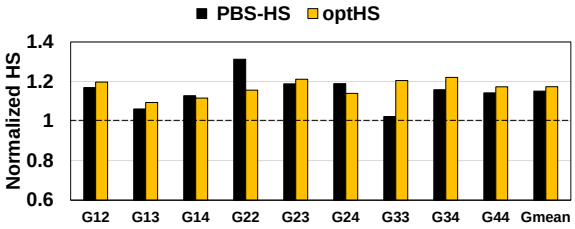


Figure 3.12: Impact of our schemes on HS.

Figure 3.12 shows the impact of the schemes on all the evaluated 50 workloads. For brevity, we do not show the results for each workload separately. Instead, we use the grouping information (Table 3.4) to form 2-application clusters and report the average (geometric mean) of the HS improvements of every workload in the cluster. As there are four groups, 10 such clusters are possible. Figure 3.12 shows the results of nine such clusters. We do not study G11 cluster as both the applications belonging to that cluster have low individual EB and interference. We observe that PBS-HS enhances HS on average by 15% over ++bestTLP. Additionally, compared to optHS, PBS-HS performance is behind by 2%, on average. We conclude that PBS-HS is a technique that can significantly enhance both system throughput and fairness over ++bestTLP.

3.6.4 Case Studies

We perform four sensitivity studies to understand the impact of the proposed schemes under different core and cache partitioning, application scaling, and memory scheduling scenarios.

Core partitioning. We test PBS under two different core partitioning scenarios: 2:1 and 1:2 partitioning (e.g., in 2:1, the first and the second applications are assigned 20 and 10 cores, respectively) and compare it to our baseline that allocates 15 cores to each application. Figure 3.13(a) shows the benefits of PBS over ++bestTLP observed in WS, FI, and HS averaged across all our 50 workloads. The bar denoted by Equal represents the average improvement of PBS with equal partitioning, and the bar denoted by Unequal represents the average improvement of PBS with the best performing partitioning scheme among 2:1 and 1:2 (found separately for each workload and then averaged). PBS enhances WS, FI, and HS under both equal and unequal core partitioning scenarios, compared to ++bestTLP. However, the benefits of PBS reduce with unequal partitioning because with different core partitioning each application executes with a different amount of TLP. As we choose the best (among 2:1 and 1:2) core partitioning configuration, the interference is also alleviated because of core partitioning itself in addition to our TLP management schemes. As the design of core partitioning is itself an interesting and non-trivial research problem, we conclude that these techniques still do not completely solve the interference problem and TLP management techniques like PBS can provide additional benefits.

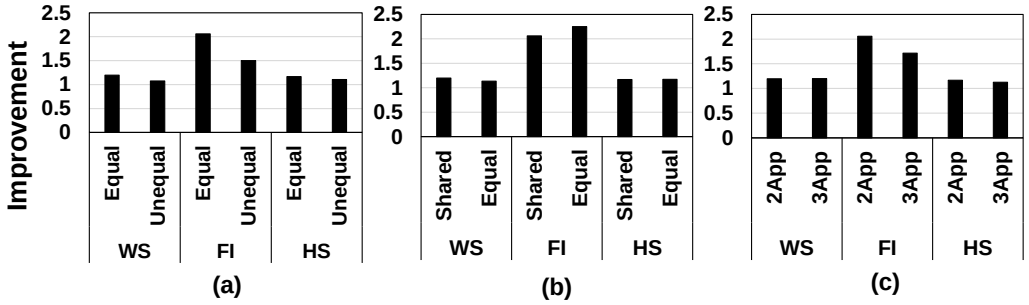


Figure 3.13: Effect of PBS over ++bestTLP with: a) core partitioning, b) cache partitioning, c) 3-application scaling.

Cache partitioning. Figure 3.13(b) shows the benefits of PBS when the L2 is way-partitioned equally (denoted by Equal) across two applications, and compare it to our baseline where the L2 cache is shared (denoted by Shared). We observe that PBS improves all three metrics (WS, FI, and HS) under both the scenarios. Cache partitioning can alleviate the interference at L2, but it might be suboptimal in scenarios where different applications in the same workload utilize the caches differently. This might lead to a portion of the cache to be not utilized well. On the other hand, PBS changes the cache demand of each application by TLP modulation.

Application Scalability. For a k-application workload, we first rank the criticality of each application in the workload based on the magnitude of the EB drop. While determining the ranking, the TLP of other applications is fixed at 24 (same as discussed before in Section 3.5). If N is the number of TLP choices, the procedure for deciding the criticality takes less than $N \times k$ steps. Subsequently, the tuning of TLP of each application would take N $\times$ $(k-1)$ steps. Therefore, the associated overall search complexity is linear to the number of applications (O($N \times k$)). In this case study, we evaluate PBS on three-application workloads by performing some straightforward extensions to the PBS. Therefore, we find the critical application one by one under the interference of the other two remaining applications, keeping other steps the same. We choose 20 representative three-application workloads to compare the average benefits to that of workloads with two applications (Figure 3.13(c)). We observe that the benefits of PBS are reasonably stable as the number of applications scales. We conclude that our techniques are not limited to workloads that consist of only two applications.

Memory scheduling. We find that PBS provides significant benefits (15% in WS and 1.92$\times$ in FI, on average across 50 workloads) over WEIS memory scheduler designed for multi-GPU execution [48]. We conclude that TLP management techniques are more effective than the previously proposed memory scheduling techniques.

3.7 Related Work

To our knowledge, this is the first work that proposes TLP management techniques for improving system throughput and fairness in a multi-application environment for GPUs. In this section, we outline some particularly relevant works.

TLP management techniques in GPUs. Rogers *et al.* proposed a mechanism that limits the TLP based on the level of thrashing in each core's private L1 data cache [99]. Kayiran *et al.* proposed a TLP optimization technique that works based on the latency tolerance of individual GPU cores [54]. Jia *et al.* proposed a mechanism that consists of a reference reordering technique and a bypassing technique such that the cache thrashing and also resource stalls at the caches reduce [44]. The input to this mechanism is from the L1 caches, and the decision is local. Sethia and Mahlke devised a method that controls the number of threads and core/memory frequency of GPUs [112]. Sethia *et al.* used a priority mechanism that allows better overlapping of computation and memory accesses by limiting the number of warps that can simultaneously access memory [111]. The work by Zheng *et al.* allows the execution of many warps concurrently without thrashing the L1 cache by employing cache bypassing [145]. All these works use *local* metrics available at the GPU cores and do not consider resource contention at the L2 caches and the memory. However, we propose mechanisms where *all GPU applications* control their TLP while being *aware of each other*. Further, our mechanisms are optimized for improving system throughput and fairness and *not* instruction throughput, which was the focus of aforementioned works.

Hong *et al.* proposed various analytical methods for estimating the effect of TLP on performance [41, 42], but do not propose run-time mechanisms. Kayiran *et al.* devised a mechanism where GPU cores modulate their TLP based on system-level congestion when GPU applications execute alongside CPU applications in a shared environment [56]. Their mechanism uses system-level metrics to unilaterally control the TLP of a *single GPU application*, whereas our mechanisms control TLP while being aware of *all applications* in

the system.

Concurrent execution of multiple applications on GPUs. Xu *et al.* proposed running multiple GPU applications *on the same GPU cores* and assigning CTAs slots to different applications to improve resource utilization of GPU cores [135]. Likewise, Wang *et al.* proposed running multiple GPU applications *on the same GPU cores*, and augmented it with a warp scheduler that adopts a time-division multiplexing mechanism for the co-running applications based on *static profiling* [133]. These intra-core partitioning techniques are used to partition resources within a core. However, co-running kernels interfere with each other significantly, especially in small L1 GPU caches. In such cases, running these kernels separately on different cores can be more effective for avoiding intra-core contention. GPU Maestro dynamically chooses between intra-core and inter-core techniques to reap the benefits of both [87]. However, none of these mechanisms change the shared cache and memory footprint of each application, and thus directly alleviate the shared memory interference. Our goal is to address the shared resource contention in L2 caches and main memory by managing TLP of each application differently. PBS allows each application to dynamically change its cache and memory footprint cognizant of the other applications' state. Pai *et al.* proposed elastic kernels that allow a fine-grained control over their resource usage [84]. Their work targets increasing the utilization of computing resources by accounting for the parallelism limitation imposed by the hardware, whereas our mechanism considers the memory system contention to modulate parallelism. Li *et al.* proposed a technique to adjust TLP of concurrently executing kernels [66], which we quantitatively and qualitatively compare in Section 3.6. We conclude that even if a new resource partitioning technique (see case study (Section 3.6.4)) is employed, the problem of multi-application contention in the memory system remains.

Cache and memory management. In the context of traditional CPUs, several works have investigated coordinated cache and memory management, and throttling for lower memory system contention. Zahedi *et al.* proposed a game-theory based approach for partitioning cache capacity and memory bandwidth for multiple software agents [141].

Ebrahimi *et al.* proposed a throttling technique that improves system fairness and performance in multi-core memory systems [24]. Eyerman *et al.* analyzed the effects of varying degrees of TLP on performance, in various multi-core designs [26]. Heirman *et al.* proposed a technique that matches the application's cache working set size and off-chip bandwidth demand with the available system resources [38]. Qureshi and Patt proposed a cache partitioning mechanism in the context of multi-core CPUs [95]. Their mechanism, based on the cache demand of each application, allocates cache space to co-running applications, whereas our mechanism changes the cache demand of each application by controlling their TLP.

3.8 Chapter Summary

This chapter analyzed the problem of shared resource contention between multiple concurrently executing GPGPU applications and showed that there is an ample scope for TLP management techniques for improving system throughput and fairness in GPUs. Our detailed analysis showed that these metrics are highly correlated with *effective bandwidth*, which is defined as the ratio of attained DRAM bandwidth to the combined cache miss rate. Based on this observation, we designed pattern-based effective bandwidth management schemes to quickly locate the most efficient and fair TLP configuration for each application. Results show that our proposed techniques can significantly improve the system throughput and fairness in GPUs compared to previously proposed state-of-the-art mechanisms. While this work focused on a specific platform with concurrently executing GPU applications, we believe that the presented analysis and the insights can be extended to other systems (e.g., chip-multiprocessors, systems-on-chip with accelerator IPs, server processors) where contention in shared caches and memory resources are performance-critical factors.

Chapter 4

Address-Stride Assisted Approximate Load Value Prediction in GPUs

4.1 Introduction

A promising strategy for reducing data movement is value prediction, whereby the values are not necessarily required to be *fetched* from memory as they can be *predicted* at the core. In the context of CPUs, previous techniques [90, 92, 91, 28, 25, 107, 108, 67] used to both predict and fetch the data. The predicted values are later compared with the fetched values. If the prediction turns out to be correct, the data-dependent stall cycles are reduced significantly. However, in the case of a misprediction, the execution is rolled back leading to the flushing of the dependent instructions in the pipeline. Such performance and data movement overheads are the critical impediments towards leveraging the benefits of value prediction. To address the challenges of *precise* value prediction, recent research has explored *approximate* value usage [73, 104, 103, 134, 139, 58], which leverages the observation that for approximable applications the requirement of rollbacks can be omitted as long as the application-level loss in quality is within an acceptable range.

While rollback-free value approximation has received significant attention in the context of CPUs [73, 104, 103, 127, 58], only a few works have explored it in the context of GPUs [134, 139]. Application execution in GPUs relies on multi-threading, where associated threads are scheduled on GPU cores at the granularity of *warps*, where a warp usually consists of 32 threads. Each load instruction in a warp can generate one or more cache block request depending on how well the data is coalesced across threads within the warp. As hundreds of warps can concurrently execute and cache sizes in GPUs are much smaller than CPUs [80], data movement between caches and memory is a serious performance and energy efficiency bottleneck [129, 49, 50, 54]. If values of these requests can be correctly predicted at the core, the data movement and stall cycles can be significantly reduced thereby improving latency tolerance, performance, and energy efficiency. However, if the predictor predicts incorrectly, each mispredicted cache line leads to a certain level of quality loss in the application's final output. This quality loss is dependent on many factors such as the prediction coverage (defined as the ratio of predicted load requests to the total load requests), the magnitude of error in value prediction, and the error resilience of instructions that use erroneous values as their operands. Therefore, if values can be predicted more accurately, higher coverage can be applied for better performance and energy efficiency.

The goal of this work is to improve the accuracy of value prediction in GPUs. One of the major challenges in achieving this goal is to identify the value stride pattern(s) in a highly multi-threaded environment where thousands of memory requests can be on-the-fly and their access order is highly dependent on GPU-specific features such as warp scheduling and coalescing. Previous works for CPUs used large per-thread prediction tables to achieve high accuracy [120, 75, 106]. However, it can become prohibitively expensive to apply those approaches directly to the highly multi-threaded environment in GPUs [139]. To address this problem, we take advantage of our key new observation that consideration of memory addresses and the relationship with their value strides is effective for providing high value prediction accuracy. Specifically, we find that for many realistic

inputs used by GPGPU applications, particular address strides have linear correlations with their value strides. For example, Figure 4.1 shows that for the extracted pixels, an address stride of $1 \times data_size$ correlates to a value stride of -1. Meanwhile, an address stride of $1 \times row_size$ correlates to a value stride of 1.

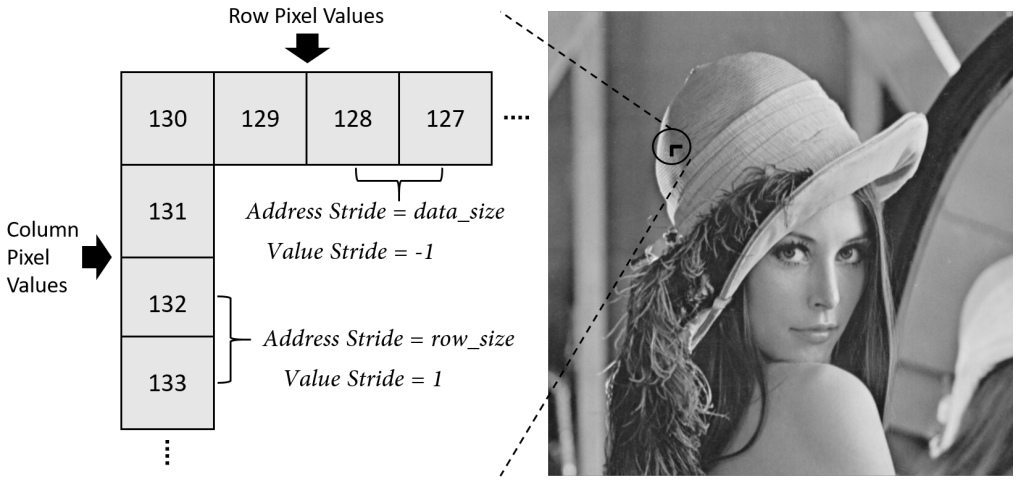


Figure 4.1: Pixel values of consecutive row and column positions.

Based on this new observation, we propose an Address-Stride Assisted Approximate Value Predictor (ASAP), which predicts the values only if it detects strides in their corresponding addresses. Each entry in the ASAP prediction table carefully keeps track of one type of address stride and their corresponding value stride. We find that as the number of address stride patterns in typical GPGPU applications is usually limited, the number of prediction table entries is significantly reduced, thereby making it area and power-efficient (Section 4.4). We also show that ASAP remains effective even under different address patterns, which can be influenced by warp scheduling and coalescing (Section 4.6).

To the best of our knowledge, this is the first work that shows that there is a high correlation between address stride and value strides in several GPGPU applications and this observation can be used to design an efficient GPU-specific value predictor. Our simulation results across a set of diverse GPGPU applications show that ASAP can significantly improve the prediction accuracy over the state-of-the art GPU value predictor

while providing high performance improvement (up to 40%) and energy reduction (up to 30%). Specifically, the previously proposed RFVP-style value predictor [139] incurs 3.48% (up to 40.08%) and 8.10% (up to 63.59%) Application Error, at 10% and 20% coverage, respectively. In contrast, under a similar area budget, our ASAP predictor produces on average only 0.26% and 0.43% Application Error, respectively.

4.2 Background

This section provides background on the GPU architecture followed by details of the existing value prediction techniques in GPUs.

4.2.1 Baseline Architecture and Metrics

Figure 4.2 shows the baseline GPU architecture with a value predictor (VP). We assume a value predictor is attached to each SM. We simulate our baseline architecture using a cycle-level simulator – GPGPU-Sim [12] and faithfully model all key parameters (Table 4.1). The energy measurements are gathered using GPUWattch [64].

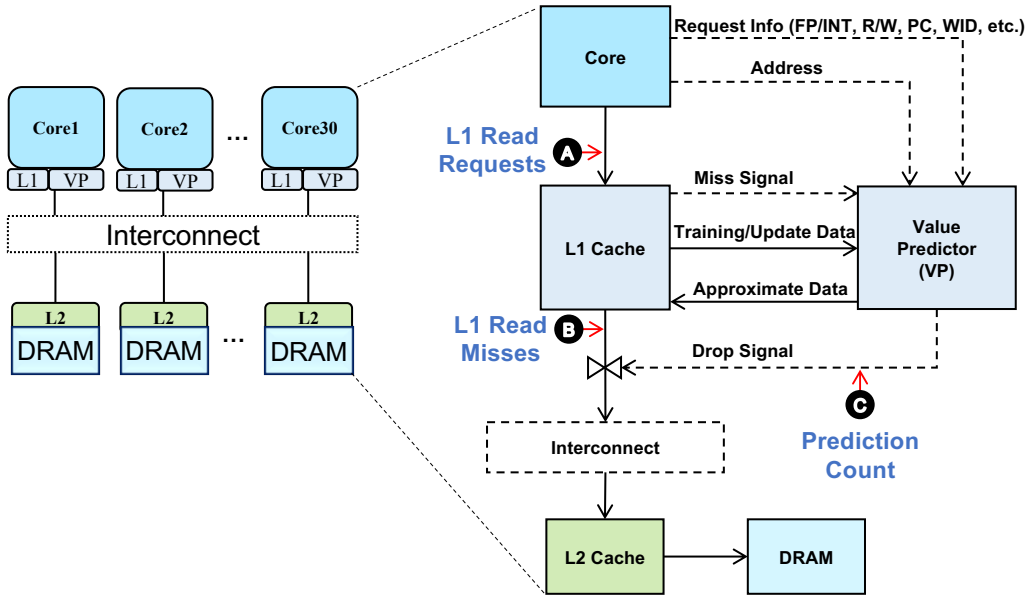


Figure 4.2: Baseline GPU Architecture with a value predictor.

Table 4.1: Key configuration parameters of the simulated GPU configuration. See GPGPU-Sim v3.2.2 [34] for the full list.

Core Features	1400MHz core clock, 30 SMs, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32KB register file Up to 1536 threads (48 warps, 32 threads/warp)
L1 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
L2 Cache	8-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 (MCs) FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock Global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing [43], $t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$
Interconnect	1 crossbar/direction (30 SMs, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

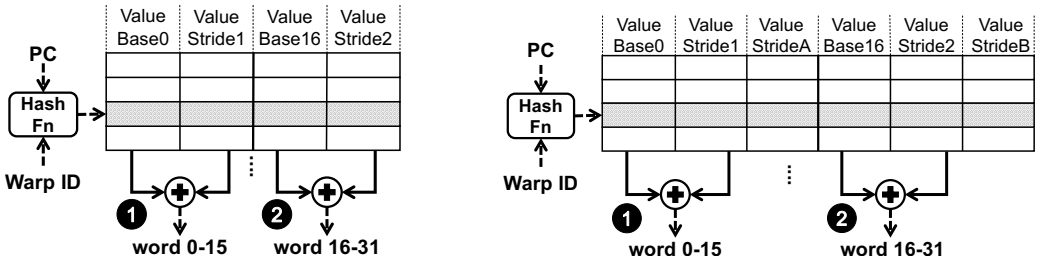
Evaluation Metrics. We summarize the metrics evaluated in this work with the help of Figure 4.2. *Coverage* is the ratio between *Prediction Count* C (i.e., cache lines that are predicted and not sent to the lower level) and *L1 Read Requests* A. Since the number of L1 Read Requests is constant for an application, the prediction accuracy across different predictors can be compared at the same coverage. *Miss Match Rate (MMR)* is the maximum achievable ratio between *Prediction Count* C and *L1 Read Misses* B. The prediction quality is measured in terms of *Application Error*, which is defined as the average relative error between the output of the approximate version and the baseline accurate version of an application.

4.2.2 Baseline Value Predictors

Figure 4.2 (right side) shows the general structure of the value predictor and its operation. The value prediction works concurrently with the cache access. We rely on user-supplied annotations to identify approximable instructions (more details are in Section 4.4.4). When a load request is issued from the core, its information (e.g., address, program counter (PC), Warp ID (WID), a bit to indicate floating-point vs. integer value (FP/INT), user-supplied annotation, memory space) is passed to the predictor. If the

cache access results in a miss, a miss signal is generated to inform the value predictor. If the value predictor is able to predict the associated cache line, it will: a) issue a drop signal to inform the MSHR to not send the cache request to the lower level of the hierarchy, and b) fill the L1 cache with the predicted data. Whenever the L1 cache is filled with a request fetched from the lower level of memory, it will be sent to the value predictor for training and update (see Section 4.4.1).

Our baseline value predictor is based on rollback free value predictor (RFVP) for GPUs [139]. RFVP takes advantage of the prediction tables implemented in the hardware to track the patterns in the data values. Specifically, RFVP uses a hash of Warp ID and PC to map different requests to particular entries in the prediction table. The prediction is performed at a granularity of the memory access size (typically 4 bytes) of a thread. However, as the prediction of all the words in a cache line is desired, the observation of intra-warp value similarity is used to predict values within the cache line. Our baseline predictor has two sub-predictors [139]. The first sub-predictor is responsible for predicting the first word, which is then copied to the first half of words (words 1 to 15). Similarly, the second sub-predictor is used for the second half (words 17 to 31). The following discussion provides the necessary background on two different RFVP-style baseline predictors.



(a) One Stride Predictor (OSP) (b) Two Stride Predictor (TSP)

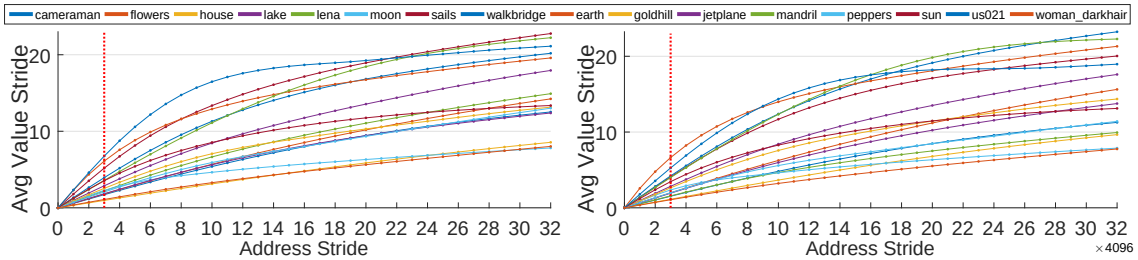
Figure 4.3: Design of the baseline value predictors.

RFVP-Style One Stride Value Predictor (OSP). Figure 4.3(a) shows the design of OSP, which uses a hash function [139] based on PC and Warp ID to map requests to entries in the prediction table. Each cache line is predicted with two one-stride sub-predictors.

The predicted value of word 0 (which is later copied to words 1 through 15) is the sum of ValueBase0 and ValueStride1 (❶). Similarly, the predicted value of word 16 (which is later copied to words 17 through 31) is the sum of ValueBase16 and ValueStride2 (❷).

Before prediction, both sub-predictors need to be trained. For the training, at least two successive cache lines are needed from the main memory. ValueBase0 is updated by the word 0 of the second cache line and ValueStride1 is updated to the difference between the word 0 of the second cache line and the first cache line. The same process is repeated for ValueBase16 and ValueStride2 of the second sub-predictor with word16 (instead of word0) of the two successive cache lines. To control the accuracy, data is periodically fetched from the main memory to update the base and stride values.

RFVP-Style Two Stride Value Predictor (TSP). Figure 4.3(b) shows the design of TSP, which uses the same hash function as OSP. However, the cache line is predicted with the help of two two-stride sub-predictors. The prediction process of TSP is similar to OSP. The predicted value of word 0 (which is later copied to words 1 through 15) is the sum of ValueBase0 and ValueStride1 (❶), and the predicted value of word 16 (which is later copied to words 17 through 31) is the sum of ValueBase16 and ValueStride2 (❷). The training process of TSP is different from OSP only regarding how the stride is calculated. For the training, at least three cache lines are needed from the memory. With three successive cache lines, ValueBase0 is updated by the word 0 of the third cache line, and the ValueStrideA is updated to the difference between the word 0 of the third and the second cache line. ValueStride1 is updated to the value of ValueStrideA only if ValueStrideA also equals the difference between the word 0 of the second and the first cache line, otherwise, the sub-predictor is considered to be not trained. The second sub-predictor adopts the same process for the values of ValueBase16 and ValueStride2. The training process stops when both ValueStride1 and ValueStride2 are found. Again, the accuracy can be controlled by periodically fetching data from the memory.



(a) Addresses are incremented row-wise. (b) Addresses are incremented column-wise.

Figure 4.4: Illustrating the relationship between average value stride of data with different address strides for a variety of inputs.

4.3 Motivation and Analysis

In this section, we first analyze the relationship between address and value strides in the inputs of GPGPU applications, followed by a discussion on how this relationship helps in improving the accuracy of the value prediction.

4.3.1 Analysis of Address and Value Strides

We find that a wide range of GPGPU workloads work on inputs that have regular values strides (also discussed in Section 4.1). In general, we observe that a large number of nearby pixels in the image are similar or have gradually changing grayscales leading to regular value strides. To validate this observation, we picked a series of images including the commonly used standard test images to analyze the correlation between their address strides and value strides. Figure 4.4 shows the average absolute value strides with increasing address strides for all pixels in each image. For example, for the address stride of 1, the corresponding average absolute value stride is the average absolute value difference between every two pixels with consecutive addresses. The unit of the address stride is the size of the data type used. Specifically, Figure 4.4(a) shows how the average value stride changes along the row of the image. Meanwhile, Figure 4.4(b) shows how the average value stride changes along the column of the image. As we can observe from the two figures, different images show different extent of linear correlations. Overall, the smaller

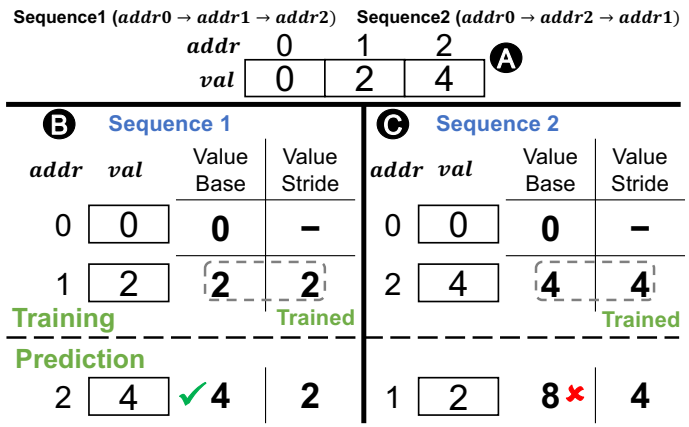


Figure 4.5: Illustrative example showing the importance of request order on value strides and the ease of value predictability.

the address stride, the more linear the correlation. Specifically, for the value strides of the next three nearby pixels (i.e., on the left of the red dashed line) both row-wise and column-wise, all images show nearly constant slopes between their address strides and value strides.

4.3.2 Motivation

We find that the observation of linear correlation between address stride and value stride can help to improve the accuracy of value prediction in GPUs. Consider an illustrative example shown in Figure 4.5. Assume that three cache line requests are generated from three different warps with addresses 0,1,2 and values 0,2,4, respectively (A). Therefore, the address stride and value stride are linearly correlated. However, these requests can be generated in different orders based on the warp scheduling policy in a GPU. Consider two possibles address sequences: Sequence I (0,1,2) (B) and Sequence II (0,2,1) (C), as shown in Figure 4.5.

In the first sequence, OSP is trained with the first two accesses. It can accurately predict the third value for Sequence I because the predicted value stride conforms with the actual stride. However, if the predictor is trained with Sequence II, a large relative error is detected because the calculated stride is incorrect. Hence, if a new value predic-

tor is able to take advantage of the address and value stride correlation, then it is able to generate an approximation with better quality for images with similar attributes as shown in Figure 4.4. Meanwhile, the same data movement reduction and performance improvements are also achieved (Section 4.6).

To confirm this intuition, we analyzed a variety of real GPGPU applications[1]. Our profiling analysis examines the value strides by calculating the average stride difference between every two *consecutive* observed value strides. For example, if the values of three consecutive loads are V1, V2, V3, we examine the difference between (V2-V1) and (V3-V2). A smaller stride difference means that strides are more regular and hence it is easier to predict the values of future loads. As the value of stride difference is dependent on the load access order, we measure it on a simulated baseline GPU architecture (Section 4.5) under three scenarios. Note that we use the first 4B of cache lines accessed by load instructions to determine value strides. First, the stride difference is calculated from the loads belonging to the same PC and are generated as determined by the baseline GTO warp scheduler. Such a scenario mimics a PC-based value predictor that only considers value patterns of loads that have the same PCs (i.e., PC-Based). Second, the stride difference is calculated from loads that do not necessarily belong to the same PC but their addresses have regular strides (i.e., Address-Stride-Based). Third, as indicated in Figure 4.4 that nearby data tend to show stronger address and value stride correlation, we use the same design as in the second scenario but restrict the address stride to accept the closest data only for each application depending on their inputs (i.e., Address-Stride-Based-Restricted). The selection process of the restricted address stride is described in Section 4.5.2.

Figure 4.6 shows the normalized results for these scenarios: PC-based, Address-Stride-Based and Address-Stride-Based-Restricted, respectively. We observe that in the second scenario where the address strides are considered, the average stride difference is much lower. This indicates that consideration of memory addresses with regular strides can

[1]More details on the application characteristics/inputs and evaluation methodology are discussed in Section 4.5.

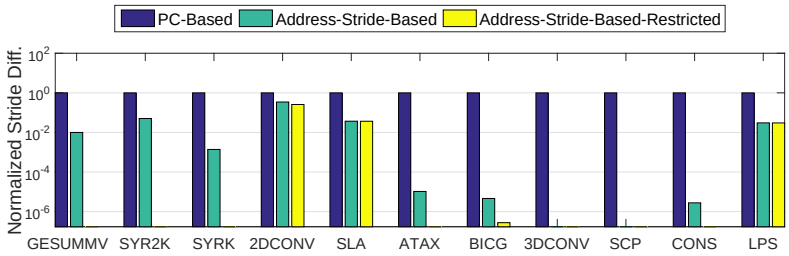


Figure 4.6: Normalized Stride Difference (in log scale) between *consecutively* observed value strides. Considering address-stride-based (2nd and 3rd bar) improves the value predictability over traditional PC-based approach (1st bar).

facilitate detecting regularities among value strides. In the third scenario, the average stride difference is even lower, as the average stride difference of many applications even reach 0. This confirms our observation in Figure 4.4. However, this scenario requires the user to specify an acceptable address stride. Considering that the Address-Stride-Based scenario already provides good improvements, we propose Address-Stride Assisted Value Predictor (ASAP) with the default mode and also evaluate the restricted mode for comparison purposes.

4.4 Design and Operation

In this section, we describe the design and operation of ASAP via answering the following these high-level questions: 1) How do we recognize the patterns in the memory address stream and leverage them for improving the accuracy of value prediction? 2) How do we handle irregular memory access orders? and 3) How do we ensure the design of the value predictor to be area-efficient?

4.4.1 Design of ASAP

Overview. The Address-Stride Assisted Value Predictor (ASAP) provides two modes: default mode and restricted mode. Both of them have the same operations and working sequence except that the restricted mode only accepts user-defined address strides. For this reason, we do not differentiate them when introducing the design of ASAP. Figure 4.7

shows the overall design of ASAP that is built upon the baseline predictor as described earlier in Section 4.2.2. There are two major changes associated with ASAP. First, ASAP does *not* rely on the PC or Warp ID based tags or hash functions but uses the address of the memory requests to map them to the prediction table entries. Second, each prediction table entry is appended with additional fields containing the information of AddressBase (i.e., Cache Block Index) and AddressStrides (❶) to facilitate in predicting the strides in the memory addresses. The key idea behind these changes is to identify and then leverage address patterns in the memory requests in order to facilitate value prediction. If the next address is predicted correctly, *only then* its corresponding value can be predicted. Essentially, we treat each entry of the prediction table as a holder for a certain kind of address pattern in the access stream. As we observe that the types of different address stride patterns in GPGPU applications are limited, we find that eight entries are sufficient (sensitivity studies are discussed in Section 4.7).

ASAP has two versions: ASAP-OSP and ASAP-TSP, based on the type of sub-predictor it employs. For brevity, we only discuss the design of ASAP-OSP (Figure 4.7) as it captures all the design issues of ASAP-TSP. We use the same entry to store either floating-point or integer data and use 1 bit (FP/INT bit) to differentiate between them. The FP bit also indicates whether floating-point adders or integer adders should be used.

The Prediction Process. In order to track various stride patterns in the memory access stream, we use two types of AddressStride fields: AddressStrideShort and AddressStride-Long. For tracking the strides in the value stream we use two types of ValueStride fields: ValueStrideShort and ValueStrideLong. If the incoming address equals to (i.e., the address matches) the sum of AddressBase and AddressStrideShort or AddressStrideLong (❷)), then its value can be predicted. We define such a situation as a *match* (❸). For example, if an entry has AddressBase 2, AddressStrideLong 2, and AddressStrideShort 1, then the entry is able to match the next request with address 3 or 4. Once a match is detected and if the address is correctly predicted using AddressStrideShort or AddressStrideLong, then the value of word0 is predicted with the sum of ValueBase0 and ValueStrideShort1 or

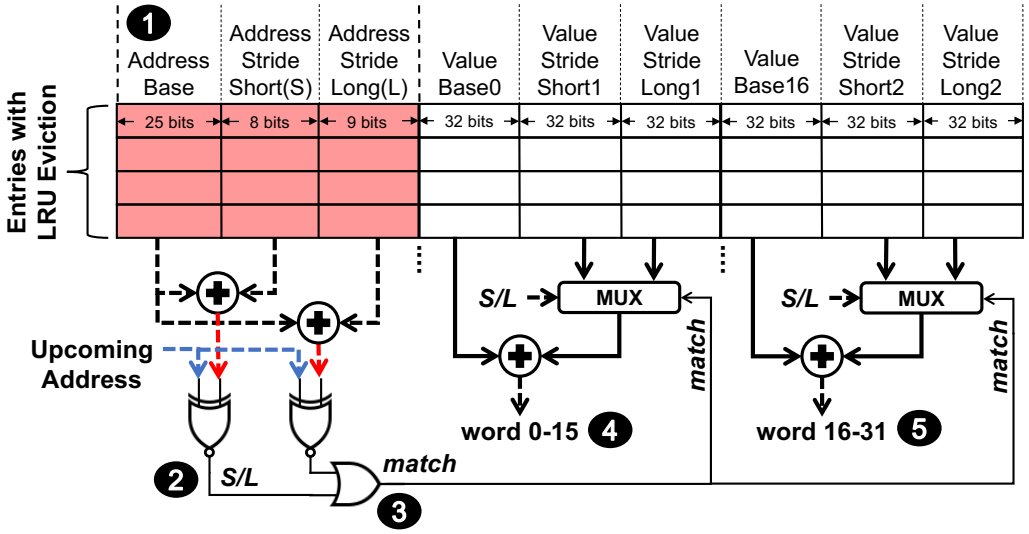


Figure 4.7: Design of the Address-Stride assisted value predictor.

ValueStrideLong1 (❹), which is later copied to words 1 through 15. The value of word16 is predicted with the sum of ValueBase16 and ValueStrideShort2 or ValueStrideLong2 (❺), which is later copied to words 17 through 31. After each prediction, AddressBase is updated to the matched address. ValueBase0 and ValueBase16 are updated to the predicted values of word0 and word16, respectively.

The Training Process. Before prediction, an entry must be trained. Each entry is responsible for tracking different address stride patterns in the access stream and is trained with at least two memory requests. For a sequence of requests, AddressBase will be set to the last address that accessed the entry. AddressStrideShort will be set to the difference between the two most recent addresses. A third request is required to train AddressStrideLong as it is the sum of the most recent two AddressStrideShort values. Therefore, the training is based on the last three requests mapped to the entry. For example, let us assume that addresses 1, 2, 4 will consecutively update an entry. After the 2nd address comes, AddressBase will be 2, AddressStrideShort will be 1 (2-1), and AddressStrideLong will remain unchanged. After the 3rd address comes, AddressBase will be 4, AddressStrideShort will be 2 (4-2), and AddressStrideLong will be 3 (1+2). During an entry's training phase, we also create and update a new entry with the 2nd and 3rd

requests of that entry. This step is performed to warm-up new entries for faster matching. New entries are created based on least recently used policy.

The Matching Process. At the first match of an entry, it leaves its training phase (i.e., it is trained) and enters the prediction phase. The value of AddressStrideShort will be set to the chosen stride (Short or Long) and the value of AddressStrideLong will be set to twice the value of AddressStrideShort in order to capture missing intermediate addresses as we will discuss in Section 4.4.3. Similarly, the corresponding Value-Stride (ValueStrideShort or ValueStrideLong) will be assigned to ValueStrideShort, and the ValueStrideLong will be twice the value of ValueStrideShort. Subsequently, the values of AddressStrideShort and AddressStrideLong will remain fixed during the lifetime of the entry. Note that before an entry is trained, StrideLong is not necessarily equal to the twice of StrideShort, as mentioned in the training process.

The Updating Process. When an L1 Miss is not predicted, the fetched cache line is used to update the AddressBase, ValueBase, ValueStrideShort, and ValueStrideLong of the matched entries to increase the accuracy of future predictions, while the AddressStrideShort and AddressStrideLong remain unchanged. The ratio of prediction and update is controlled by the desired coverage that a user can specify. Hence, the predictor will predict only if the desired coverage has not been reached. Both the prediction and the update can only happen in a matched entry. To ensure the updates are evenly distributed, we predict and update in a fine-grained manner. For example, 50% coverage can be achieved by doing 5 consecutive predictions followed by 5 consecutive updates.

Note that there should be at least 2 consecutive updates for ASAP-OSP and 3 for ASAP-TSP in order to update the value strides in their corresponding sub-predictor (Section 4.2.2). When an update or a prediction request comes to the predictor, entries are checked one by one to see if there is a match in any of the entries. We find that this small extra latency does not affect the performance benefits obtained from dropping the request. The match for AddressStrideShort and AddressStrideLong inside each entry happens in parallel. If no match is found, we replace an old entry with a new entry based on LRU

policy and start its training phase.

4.4.2 Operation of ASAP

Figure 4.8 illustrates the operation of ASAP-OSP and its advantages over RFVP-Style-OSP by considering a sequence of addresses (0,1,2,4,3,5). The corresponding data values are shown in boxes next to each other (A). In this example, we assume that the address sequence is generated from the same PC. When RFVP-Style-OSP is employed (B), the first two requests train the entry. After training, ValueBase is 2 and ValueStride is 2. The third request is correctly predicted because its value is the sum of ValueBase and ValueStride. However, the values of fourth and fifth memory requests are incorrectly predicted because the ValueStride of 2 does not correctly capture the value pattern. However, the value of the sixth request is correctly predicted as it matches with the sum because the ValueBase was still being updated even when the predictions were wrong. Overall, the coverage is 66% (4 out of 6 requests are predicted) and accuracy is 50% (2 out of 4 predictions are accurate).

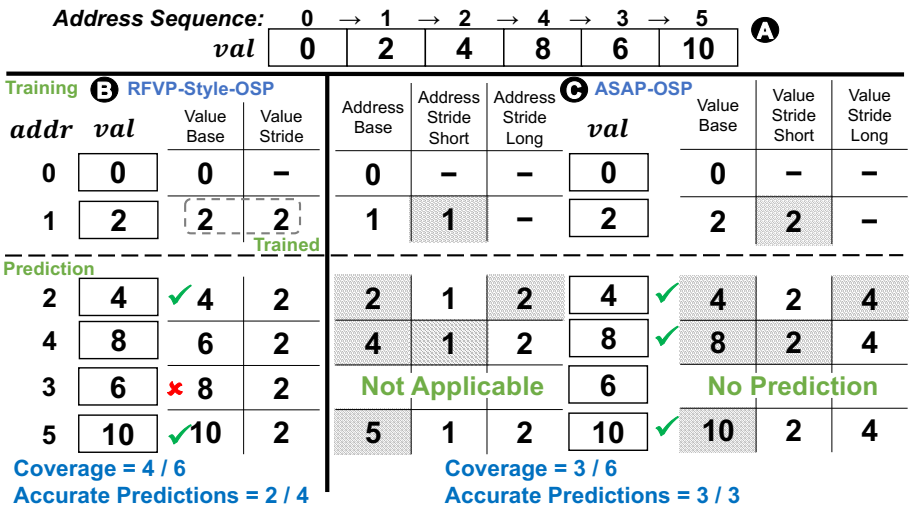


Figure 4.8: Operation of ASAP and its advantages over OSP. The matched addresses, predicted values, relevant strides are shaded.

In the case of ASAP-OSP (C), AddressBase, AddressStrideShort, and AddressStrideLong are responsible for detecting strides in the addresses. After the first two accesses,

AddressBase and AddressStrideShort are trained in addition to ValueBase and ValueStrideShort. As the third address is the sum of AddressBase and AddressStrideShort, it implies that there is a match and its corresponding value can be predicted. We observe that ASAP-OSP can correctly predict its value as its sum is equal to ValueBase and ValueStrideShort. At this point, AddressStrideLong and ValueStrideLong are also set as equal to twice of AddressStrideShort and ValueStrideShort, respectively. The fourth address is also a correct match as the sum of AddressBase and AddressStrideLong matches with the predicted address. Therefore, its value can be predicted using the sum of ValueBase and ValueStrideLong, which is also correctly predicted. The fifth request is not predicted because its address does not match the pattern in the addresses (Addr 3 is neither equal to the sum of AddressBase and AddressStrideShort nor AddressBase and AddressStrideLong). Finally, the sixth request can be correctly predicted as its address pattern can be captured via AddressStrideShort. Overall, the coverage is 50% (3/6 requests are predicted) and accuracy is 100% (3/3 predictions are accurate). In summary, as opposed to the RFVP-Style-OSP, ASAP-OSP can take advantage of the readily available address information and improve the accuracy significantly by trading-off coverage.

4.4.3 Use Cases of ASAP

For the address sequences which are generated from the core, we find that there are two possible cases which can make them difficult to be captured. The first case is that in an address sequence with a particular stride some of the intermediate addresses are missing. The second case is that multiple address sequences are interleaved together leading to a complicated address sequence. Our ASAP design takes these two cases into consideration and we will also evaluate its effectiveness with real applications later in Section 4.7. To help understand how ASAP can capture different kinds of strides in the addresses, we present three scenarios.

Scenario I: Regular Address Pattern – Demonstrating the utility of multiple entries. Consider a scenario when consecutive address sequences: (0, 1, 2, 3) and (10,

Request Order ↓

Block Index	Entry0			Entry1			Entry2		
	Address Base	Address Stride Short	Address Stride Long	Address Base	Address Stride Short	Address Stride Long	Address Base	Address Stride Short	Address Stride Long
0	0	NA	NA						
1	1	1	NA	1	NA	NA			
2	2	1	2	2	1	NA	2	NA	NA
3	3	1	2						
10				10	8	9	10	8	NA
11							11	1	9
12							12	1	2
13							13	1	2

Figure 4.9: Working steps of ASAP in Scenario I: Regular Address Pattern. The address stream considered is: 0, 1, 2, 3, 10, 11, 12, 13. The matched addresses and relevant strides are shaded.

11, 12, 13) are generated back to back. Figure 4.9 shows the values of AddressBase, AddressStrideShort, and AddressStrideLong for each entry. For brevity, we only show the first three entries that are relevant for this example. After the first two addresses are mapped to the first entry (Entry0), the remaining addresses of the sequence (2,3) are matched as they belong to the same stride pattern (AddressStrideShort of Entry0 is set to 1). Note that AddressStrideLong is set to twice of AddressStrideShort and the next Entry (Entry1) is also prepared in anticipation of other possible patterns in the addresses by setting the AddressBase to be 2 and AddressStrideShort to be 1.

When the second sequence of addresses arrive at the predictor, the first address (10) among them cannot be matched by Entry0 because neither the sum of AddressStrideShort or AddressStrideLong with AddressBase matches with the address. Therefore, it will be mapped to Entry1. AddressStrideShort is set to 8 (10-2) and AddressStrideLong becomes the sum of the previous two AddressStrideShort (8+1) for Entry1. After Entry1 is trained with 3 requests, it cannot match the next address 11, so again 11 is put into a new entry (Entry2). The Entry2 is trained with 2, 10, 11 and is able to match remaining addresses of the sequence (12,13).

Scenario II: Interleaved Address Pattern – Demonstrating the utility of Ad-

dressStrideLong. The interleaved address pattern may be caused by the interleaved execution of two warps, or by the poorly coalesced requests from certain load instructions. For example, one warp generates addresses (1, 2), another warp generates (4, 5), and so on. Figure 4.10 demonstrates such a sequence: (1, 2, 4, 5, 7, 8, 10, 11). For Entry0, it is trained with three addresses 1, 2, 4. Also, 2, 4 are copied to Entry1. For the next address 5, since Entry0 has reached its maximum training count of 3, it can only be put into Entry1 which still has 1 slot for training. At this point, both Entry0 and Entry1 have AddressStrideLong equal to 3. So when address 7 comes, it matches with AddressStrideLong in Entry0. The next address 8 also matches with AddressStrideLong in Entry1. Addresses 10, 11 can also be matched with Entry0 and Entry1, respectively.

Request Order ↓

Block Index	Entry0			Entry1		
	Address Base	Address Stride Short	Address Stride Long	Address Base	Address Stride Short	Address Stride Long
1	1	NA	NA			
2	2	1	NA	2	NA	NA
4	4	2	3	4	2	NA
5				5	1	3
7	7	3	6			
8				8	3	6
10	10	3	6			
11				11	3	6

Figure 4.10: Working steps of ASAP in Scenario II: Interleaving Address Pattern. The addresses considered are: 1, 2, 4, 5, 7, 8, 10, 11. The matched addresses and relevant strides are shaded.

Scenario III: Missing Intermediate Address Pattern. We present an example of handling a missing intermediate address in a consecutive address sequence. The missing intermediate address pattern may be caused by the non-consecutive scheduling of warps or control divergence. Figure 4.11 demonstrates such case with a sequence: (0, 1, 2, 3, 5). After 0, 1, 2 are mapped to Entry0, it is trained with the value of AddressStrideShort to be 1 and AddressStrideLong to be 2. Hence, after address 3 is matched, it can match address 5 from the AddressBase 3 directly using AddressStrideLong. Without the AddressStride-

Long, we would not have matched with address 5, thereby limiting the coverage.

Block Index 4 is missing

Block Index	Entry0			Entry1		
	Address Base	Address Stride Short	Address Stride Long	Address Base	Address Stride Short	Address Stride Long
0	0	NA	NA			
1	1	1	NA	1	NA	NA
2	2	1	2	2	1	NA
3	3	1	2			
5	5	1	2			

Request Order

Figure 4.11: Working steps of ASAP in Scenario III: Missing Intermediate Address Pattern. The addresses considered are: 0, 1, 2, 3, 5. The matched addresses and relevant strides are shaded.

4.4.4 Output Quality Control

The rollback-free value prediction eliminates pipeline rollbacks, which is prohibitively expensive in GPUs. However, it also introduces errors in GPU pipelines. These errors lead to different types of consequences if no restrictions are enforced to control them. First, the application may crash or lead to an unknown behavior if errors are generated for critical values. For example, an incorrect PC or address value will likely cause a fatal error. Second, the application's execution trace can become vastly different and produces unexpected results if errors are generated for values involved in conditional branching. For example, an incorrect counter in a *for* loop can produce unusual results in an application's output. Third, the application's output can lose a certain level of quality. For example, some mispredicted values in an input matrix may cause a certain level of distortion to an application's output. In this case, the level of quality loss depends on: a) the accuracy of the individual predictions, b) the number of values predicted (i.e., prediction coverage), and c) the future operations that will be applied to these predicted values.

ASAP guarantees that only limited output quality loss can happen by requiring the programmer's annotations of approximable load values and taking the input of prediction coverage values. The compiler is also slightly modified to accept these additional directives

to facilitate value approximation. For example, as shown in Listing 5.1, 10% prediction coverage is specified by the fetch and predict ratio of 9 to 1. The programmer has also indicated to approximate the value of vector B in the following memory load operation. Therefore, as shown in Listing 4.2, the added directives will inform the predictor on two items: a) the amount of load requests to approximate (i.e., coverage), and b) which load instruction to approximate.

```
#pragma add_pred{fetch, 9, predict, 1}
...
#pragma approx{B}
C[i] = A[i] + B[i];
```

Listing 4.1: annotated CUDA code

```
.fetch 9
.predict 1
...
ld.global.u32.approx %r0, [%r1]
```

Listing 4.2: generated PTX code

ASAP uses prediction coverage to trade off output quality for performance. To satisfy a certain output quality threshold, ASAP can rely on the programmer to provide an appropriate prediction coverage so as to maximize the performance gains under this threshold. Previous works [14, 69, 102, 88] have indicated that the application error cannot be bounded automatically in the first kernel invocation as the error of approximation depends on the semantics of the application. However, a multi-invocation approach is still able to automatically find the optimal prediction coverage. Hence, ASAP can employ a searching method which is similar to the proposed approach of prior work [101] to find the highest prediction coverage for a given output quality requirement. As we will show in Section 4.6.1, at the same prediction coverage, ASAP can lead to less output quality loss

Table 4.2: List of evaluated GPGPU applications.

Abbr.	Input	Category	Coalescing	Int.
GESUMMV [93]	2048 × 2048 Matrix	BW-Bound	Good	No
SYR2K [93]	128 × 128 Matrix × 3	Latency-Bound	Good	No
SYRK [93]	256 × 256 Matrix × 2	Latency-Bound	Good	No
EMBOSS (2DCONV) [93]	4096 × 4096 Image	BW-Bound	Poor	Yes
BLUR (2DCONV) [93]	4096 × 4096 Image	BW-Bound	Poor	Yes
ATAX [93]	4096 × 4096 Matrix	Latency-Bound	Good	No
BICG [93]	3072 × 3072 Matrix	Latency-Bound	Good	No
3DCONV [93]	256 × 256 × 256 Matrix	BW-Bound	Poor	Yes
SLA [12]	Size 24000000 Array	Energy-Bound	Good	No
LPS [12]	256 × 256 × 256 Matrix	BW-Bound	poor	Yes
SCP [12]	Vector × 16384	BW-Bound	Good	No
CONS [12]	8192 × 8192 Matrix	Energy-Bound	Good	Yes

than the state-of-the-art value predictor for GPUs. Reciprocally, ASAP is able to achieve higher performance improvements under the same output quality threshold.

4.4.5 Hardware Overhead

Figure 4.7 shows that ASAP-OSP uses 234 bits per entry. Additionally, it uses three fields (not shown) namely Status (5 bits), LRU (3 bits), and Floating-point (1 bit), making the overhead per entry 243 bits. Status bits are used to track the current status of the entry (e.g., training phase, predicting phase or update phase) in order to decide the entry's action for the next matched request. We have discussed the transitions between different statuses in Section 4.4.1. The LRU bits are used to track the LRU information. The Floating-point bit is used to differentiate the floating-point and integer data. Since ASAP uses eight entries per core, the total overhead is 243 bits $\times 8 = 1944$ bits/Core. ASAP-TSP has four extra fields per entry. These fields are ValueStrideShortA, ValueStrideShortB, ValueStrideLongA, and ValueStrideLongB, thus, the total overhead is $(243 + 32 \times 4) \times 8 =$ 2968 bits/Core (0.36KB/Core). In addition to these bits, each core employs four integer adders, two floating-point adders, four 2×1 MUXes, two comparators, and one OR gate.

4.5 Evaluation Methodology

4.5.1 Application Characteristics

We consider a variety of GPGPU applications from Polybench [93] and CUDA SDK [12] as shown in Table 4.2. We chose them as they show diversity in terms of memory intensity and coalescing behavior. Also, these applications use matrix or vector inputs with strided values provided by their corresponding benchmark suites and they can accept realistic images as their inputs. We use annotations to mark the approximable loads. We ensure that they do not contain pointers or lead to fatal errors, and thus can be approximated safely. The programmer can also tune the aggressiveness of value approximation by adjusting the prediction coverage [73, 103, 139] or using only restricted address strides (Section 4.5.2). If there are drastic variations in value strides for all given address strides, the programmer can choose to turn off the value predictor.

We classify applications into multiple categories. The **BW-Bound** applications have high DRAM bandwidth utilization (at least 40%) and relatively low IPC (at most 500). The **Latency-Bound** applications have low IPC (less than 100) and low bandwidth utilization (less than 10%). For both these classes, we expect value prediction would provide performance and data movement reduction benefits. The **Energy-Bound** applications have high IPC (more than 500) and significant off-chip traffic. For such applications, we expect value prediction would provide data movement reduction benefits but not necessarily performance benefits. Finally, we also considered coalescing conditions. The loads of `EMBOSS`, `BLUR`, `3DCONV`, and `LPS` are poorly-coalesced (i.e., two or more cache line requests are generated per load instruction per warp.). Other applications have good coalescing characteristics. Our applications use integer or floating point data. The last column of Table 4.2 shows whether the data type is integer (Int.) or floating point.

4.5.2 Choice of the Restricted Address Strides

For the restricted mode of ASAP, we manually set the acceptable address strides. As we have discussed in Section 4.3, we would prefer address strides that correspond to closer data on the same row or column of the input. However, as cache lines in GPUs typically contain 128 consecutive bytes, the address stride need to be set to at least 128 in order to predict cache lines in the same row. Therefore, the address stride of $\pm row_size$ of inputs is used for all applications, which corresponds to the closest cache lines in the same column. Also, other address strides are used if they show linear correlations with their corresponding value strides.

4.6 Experimental Evaluation

We compare the proposed ASAP design with the prior RFVP-Style predictors adopting both OSP and TSP sub-predictors (Section 4.2.2). For a fair comparison, we use the same number of entries (i.e., 8) across all predictors. They are RFVP-OSP-8, RFVP-TSP-8, ASAP-OSP-8, ASAP-TSP-8, ASAP-OSP-8-Restricted and ASAP-TSP-8-Restricted. We also compare ASAP design with the oracle implementations of RFVP-Style-OSP and RFVP-Style-TSP, namely RFVP-OSP-Unlimited and RFVP-TSP-Unlimited, respectively. These oracle implementations assume unlimited hardware budget for the prediction table entries. Hence, the loads from each PC and Warp ID combination can use a separate entry such that predictions from different PCs and warps do not affect each other.

4.6.1 Effect on Output Quality

Figure 4.12 shows the comparison of Application Error of 10% and 20% coverage. We limit the predictors' coverage to be under 20% to restrict the error. However, if the user has knowledge of the application's good predictability (e.g., `SCP`), coverage can be set higher for more performance and data movement reduction benefits. We make the following observations. First, we find that at the same coverage, our proposed predictors have

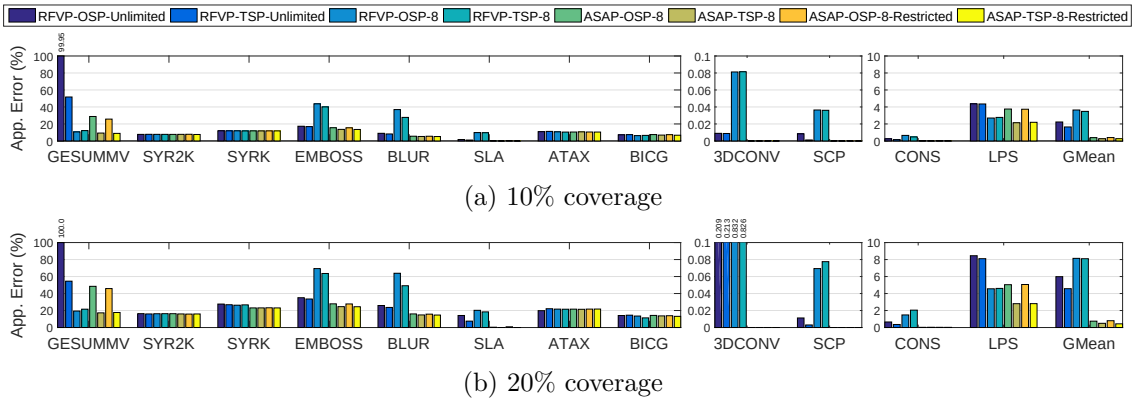
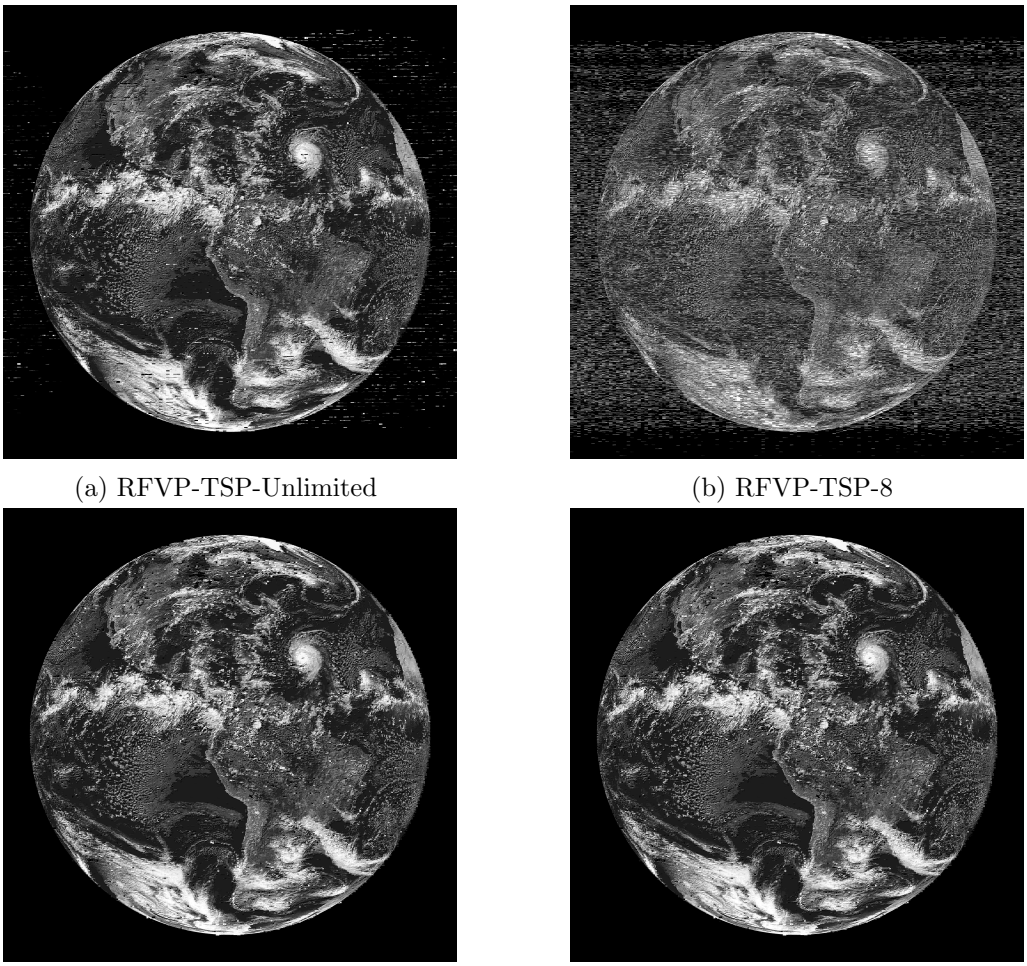


Figure 4.12: Application Error for different value predictors at (a) 10% (b) 20% coverage.

better accuracy than the previous predictors, even when the number of entries for ASAP is much less than RFVP. On average, RFVP-Unlimited predictors predict more accurately than RFVP-8 predictors, and ASAP-8 predictors are more accurate than RFVP-Unlimited predictors. Also, for each category of predictors, the TSP predictors are more accurate than the OSP predictors, showing the effectiveness of predicting with more regular strides. At 10% coverage, both ASAP-TSP-8 and ASAP-TSP-8-Restricted reduce Application Error by 92% over RFVP-TSP-8 and 84% over RFVP-TSP-Unlimited. At 20% coverage, ASAP-TSP-8 reduces Application Error by 94% over RFVP-TSP-8 and 89% over RFVP-TSP-Unlimited. ASAP-TSP-8-Restricted reduces Application Error by 95% over RFVP-TSP-8 and 91% over RFVP-TSP-Unlimited.

Second, we find that for certain applications, (i.e., `EMBOSS`, `BLUR`, `SLA`, `3DCONV`, `SCP`, `CONS`, `LPS`), the increase in application error with increasing coverage is at a much slower rate in ASAP compared to that in other predictors. This implies that ASAP is able to better exploit the relationship between address and value strides to improve the accuracy even at higher coverages. For applications `SYR2K`, `SYRK`, `ATAX`, `BICG`, the accuracy benefits of ASAP and RVFP are comparable, implying no obvious address and value stride correlations exist in them. There are cases where RFVP-8 predictors have better accuracy than the RFVP-Unlimited predictors (i.e., `GESUMMV`, `LPS`). This indicates that not sharing the prediction table entries across warps degrades the prediction accuracy in some

cases. However, ASAP-TSP-8 and ASAP-TSP-8-Restricted still have better accuracy in this case, because they capture more stable value strides according to the address pattern observed across different warps and PCs.



(a) RFVP-TSP-Unlimited (b) RFVP-TSP-8

(c) ASAP-TSP-8 (d) ASAP-TSP-8-Restricted

Figure 4.13: `EMBOSS(2DCONV)` outputs at 10% coverage.

For the image output quality, we pick `EMBOSS(2DCONV)` at 10% coverage to study the difference between predictors. As shown in Figures 4.13(a) to (d), there is significantly less noise in ASAP-TSP-8 and ASAP-TSP-8-Restricted than in RFVP-TSP-8. Further, there is slightly less noise in ASAP-TSP-8 and ASAP-TSP-8-Restricted than in RFVP-TSP-Unlimited. This trend matches the Application Error result. For these outputs,

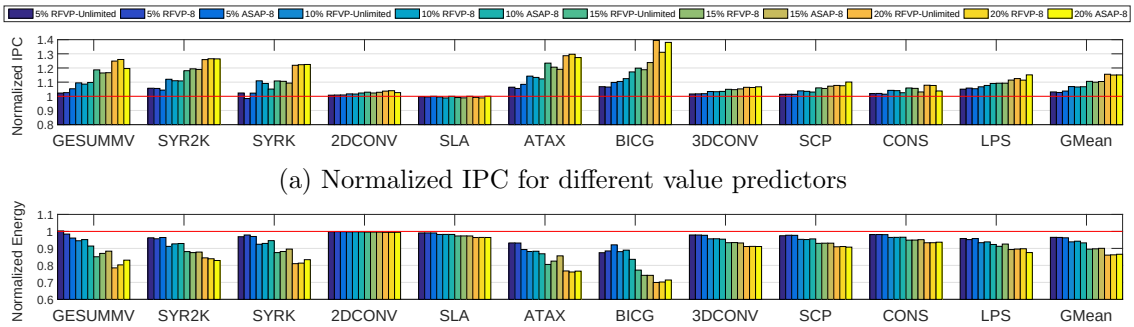
ASAP-TSP-8 shows 13.6% Application Error and ASAP-TSP-8-Restricted shows 13.5%. We find that these errors do not cause significant quality losses.

We conclude that by leveraging the address and value stride correlation, (regardless of the PC and Warp ID information), ASAP can effectively improve the prediction accuracy over the RFVP-Style predictors with a similar or lower area budget. Without extra burden on the user, ASAP's default mode can provide accuracy close to that of the restricted mode. Meanwhile, the restricted mode can further increase the accuracy with user-specified address strides.

4.6.2 Effect on Performance and Energy

Figure 4.14 shows the performance and energy benefits of applying value prediction. For brevity, we show results of RFVP-TSP-Unlimited, RFVP-TSP-8, and ASAP-TSP-8 from each of the three predictor categories. We also confirm that other predictors show similar trends. Since the RFVP and ASAP predictors predict similar numbers of cache lines at the same coverage, they provide similar IPC and energy benefits. However, ASAP produces smaller errors. On average, ASAP-TSP-8 improves IPC by 7% at 10% coverage and improves IPC by 15% at 20% coverage. Specifically, for the Latency-Bound applications, we observe an average IPC improvement of 11% at 10% coverage and an average IPC improvement of 29% at 20% coverage. On the other hand, ASAP-TSP-8 reduces GPU energy consumption by 7% at 10% coverage and reduces GPU energy consumption by 14% at 20% coverage. We conclude that the prediction coverage is the dominant factor of performance and energy benefits in value approximation. Value approximation can effectively improve performance and reduce energy consumption. Also, these benefits grow larger when the prediction coverage increases.

On average, the best performing ASAP predictor produces only 0.26% (up to 13.51%) Application Error at 10% coverage and 0.43% (up to 24.47%) Application Error at 20% coverage. In contrast, with the same number of entries, the best performing RFVP predictor incurs 3.48% (up to 40.08%) Application Error at 10% coverage and 4.57% (up to



(a) Normalized IPC for different value predictors

(b) Normalized total energy consumption for different value predictors

Figure 4.14: GPU performance and total energy consumption at different coverages.

54.54%) Application Error at 20% coverage. Even when using different entries for each PC and Warp ID combination, RFVP incurs 1.65% (up to 51.74%) Application Error at 10% coverage and 8.10% (up to 63.59%) Application Error at 20% coverage, which is much higher than that of ASAP with 8 entries. This means that ASAP can employ higher coverages and consequently obtain more performance and energy benefits if a certain error threshold needs to be satisfied. We conclude that ASAP can achieve more performance and energy benefits under a specific error threshold.

4.7 Sensitivity Studies

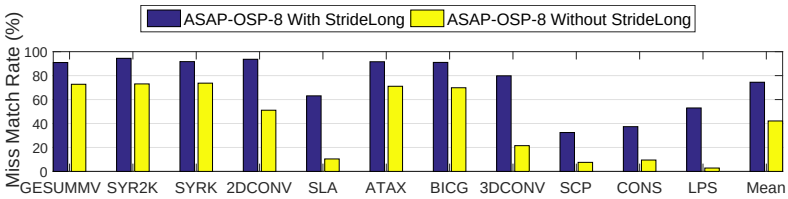


Figure 4.15: Effect of AddressStrideLong on Miss Match Rate.

Effect of AddressStrideLong. The Miss Match Rate (MMR) reflects how effective ASAP is able to capture the address patterns in GPUs. It is important for ASAP to achieve an acceptable MMR, as it determines the maximum coverage of ASAP. For example, the MMR needs to reach at least 20% for applications with 100% L1 Miss Rate, if the desired coverage is 20%. To prove the effectiveness of AddressStrideLong, we show the MMR

of ASAP-OSP-8 with and without it. As shown in Figure 4.15, we find that for poorly-coalesced applications (i.e., `EMBOSS`, `BLUR`, `3DCONV`, `LPS`), ASAP-OSP-8 has much higher MMRs with StrideLong. We also find that for well-coalesced applications, the MMR can become low if StrideLong is not employed (i.e., `SLA`, `SCP`, `CONS`). This limits their maximum coverage, data movement reduction, and performance benefits. This effectively shows ASAP's ability to match for interleaving and missing intermediate address patterns with the help of AddressStrideLong (see Section 4.4). Other ASAP predictors also show the same trend. We conclude that ASAP's address matching ability under complex patterns can be significantly improved with AddressStrideLong.

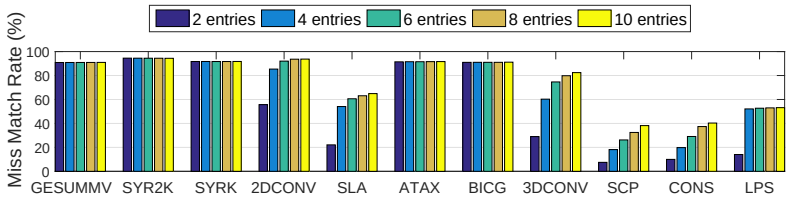


Figure 4.16: Miss Match Rate with different entry numbers.

Effect of Number of Entries. Figure 4.16 shows the MMR of ASAP-OSP-8 with different numbers of entries. We make two observations. First, applications `SLA`, `3DCONV`, `SCP`, `CONS` need more entries to achieve high MMR as there are more co-existing address patterns. Others can reach high MMR even with 2 entries, which indicates that they have fewer co-existing address patterns. Second, after size 8, all applications' MMRs do not improve significantly and therefore we use it as ASAP's default configuration. Other ASAP predictors also show the same trend. We conclude that ASAP achieves good MMR and accuracy without incurring large hardware overhead.

Effect of Warp Scheduling Policy. The choice of warp scheduler can affect the warp execution order, thereby affecting the address patterns [50]. Figure 4.17 shows the MMR of ASAP-OSP-8 working under: The baseline, Greedy-Then-Oldest (GTO) scheduler, and Round-Robin (RR) scheduler. We make two major observations. First, ASAP-OSP-8 has high MMR for both schedulers proving that it is adaptive for different warp schedulers.

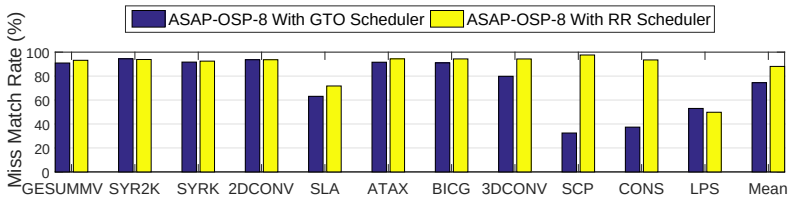


Figure 4.17: Miss Match Rate with GTO and RR Scheduler.

Second, ASAP-OSP-8 usually achieves higher MMR with the RR scheduler. This is because the address order is more regular under the RR scheduler [50].

4.8 Related Work

Previously proposed RFVP for GPUs [139] relies significantly on the program counter (PC) to detect value patterns in the memory requests. Such PC-based mapping mechanism implicitly assumes that the memory requests originated from that particular PC are ordered such that they would facilitate the prediction of values. However, as multiple warps can execute the same instruction (i.e., using the same PC) independently at different times in GPUs, the memory request order from a particular PC can be highly influenced by factors such as the choice of warp scheduling scheme. Therefore, as we show quantitatively in Section 4.6, the PC-based predictors cause a significant loss of accuracy in GPUs. This problem can be partially addressed by using a separate entry for each PC and Warp ID combination. However, such a mechanism can become prohibitively expensive as the number of concurrent warps and schedulers grows with each new generation of GPUs [84, 46, 86, 63, 72]. Moreover, using separate entries also disallow the detection of value patterns that might exist across requests from different warps (e.g., when nearby pixels of an image with regular value strides are handled by nearby warps). Wong et al. [134] proposed to exploit the intra-warp value similarity such that only one representative thread within a warp is required to perform the computation. Value approximation techniques [65] are proposed to reduce GPU energy consumption by carefully considering lower precision data/instructions. We believe ASAP is complementary to them as it

eliminates the need for accessing the main memory for the predicted cache lines.

Several value prediction techniques [90, 92, 91, 28, 25, 107, 108, 67] in the context of CPUs are based on PC-based hash mechanisms, which have similar limitations as that of RVFP described earlier. Load-value approximation techniques [73, 104, 103] and context-based value predictors [120, 75, 106] designed for CPUs consider memory addresses and other metadata for effective approximations. However, such techniques require significant *per-thread* hardware resources, which can become prohibitively expensive in GPUs as it concurrently executes thousands of threads.

4.9 Chapter Summary

In this chapter, we presented a low-overhead value predictor for GPUs that considers the correlation between address strides and value strides in order to improve the prediction accuracy. Compared to the state-of-the-art value predictor, RFVP, we find that our predictor can significantly improve value prediction accuracy even at a high value of prediction coverage (leading to significant performance and data movement benefits). We also show it is also able to function effectively even under complicated address patterns. We believe that this work can open up interesting research avenues that consider other readily available information locally at the core (e.g., address stride information) to improve the accuracy of value prediction.

Chapter 5

Exploiting Latency and Error Tolerance of GPGPU Applications for an Energy-efficient DRAM

5.1 Introduction

A large fraction of DRAM access energy is related to the fact that multiple high-energy consuming DRAM operations such as row activations and precharges must be performed, so as to access data from a DRAM row (page). These operations are required to ensure the data from the correct row is present in the row buffer, which is a limited-sized hardware structure attached to each DRAM bank. If accesses to the same row can be scheduled together without switching in and out the row buffer data (i.e., row buffer locality can be enhanced), they can incur much less row energy. Quantitatively, this energy can be around 25-50% of the total DRAM energy [142, 122, 16, 83] and is dependent on the row buffer locality workloads (higher the row buffer locality, the lower the DRAM energy). Hence, it is preferred to reuse the buffered data of a row as much as possible to improve the row buffer locality and reduce the energy consumption.

We observe that several GPGPU applications suffer from poor row buffer reuse (also

referred to as *row thrashing*). It can happen even with the popular First-Row First-Come-First-Serve (FR-FCFS) scheduler that leverages a large re-order pending request queue and an open-row policy which is typically employed to maximize the row buffer locality. This is not only caused by the GPU scheduling policies at the core but is also dependent on the applications' algorithms and their data placement mechanisms. Moreover, the multi-threading nature of the GPUs can cause severe contention and interleaving of requests at the memory controller, which can also lead to poor row buffer locality. To address this problem, we performed a detailed characterization of row buffer locality in GPUs and revealed two key insights. First, the current GPU memory scheduling policies are too aggressive in reducing latencies of requests: requests in the pending queue are issued to their destined DRAM banks as soon as these DRAM banks finish serving the previous requests. Second, the current memory scheduling policies are too strict in terms of fetching only the *exact* values from the DRAM banks. Therefore, an entire DRAM row has to be fetched into the row buffer even if it is poorly reused. We argue that these aggressive and strict policies are sub-optimal towards improving row buffer locality.

Our lazy memory scheduler relaxes the aforementioned constraints by leveraging the fact that several GPGPU applications are latency and error tolerant [134, 50]. Specifically, our proposed memory scheduler works in two modes: delayed and approximate. The delayed memory scheduling (DMS) carefully delays (i.e., increases the access latency) the issuing of both read and write pending memory accesses so that more requests can be accumulated in the FR-FCFS pending queue. This helps the memory scheduler to find more requests (i.e., will have more visibility) that can be co-scheduled back to back to the same DRAM row leading to improved row buffer locality. Because several GPGPU applications are inherently latency tolerant as they spawn thousands of threads to hide the long memory access latencies (which is not the case for most of the workloads executed on CPUs), we find that the additional delay does not affect performance significantly for many GPGPU applications. However, for certain applications that cannot tolerate latency significantly, DMS is also able to find an appropriate delay to avoid severe loss in

performance.

The approximate memory scheduling (AMS) is based on our observation that a large portion of row activations is caused by only a small portion of memory accesses. To this end, the goal of AMS is to find these accesses with low row buffer localities in the pending queue and return them immediately instead of issuing them to the DRAM banks. The values of such a small portion of memory accesses can then be approximated using various existing techniques [104, 103, 139] on their way back to the cores. These techniques bound the error with the help of programmer annotations and by predicting only a fraction of memory requests (called as prediction coverage). We demonstrated the effect of approximation on the application output by using a simple value predictor, which makes use of the readily available data in the associated L2 caches of the memory partitions. Because of the fact that many GPGPU applications are error tolerant or can accept limited losses in the output quality [134, 139], we find that such an approach can help in significantly reducing the number of row activations. Overall, AMS focuses on the problem of *when* to approximate and allow the new or existing works [104, 103] to address the equally important problem of *how* to approximate.

To the best of our knowledge, this is the first work that improves the row buffer locality and reduces row energy in GPUs via carefully delaying and/or approximating the memory requests (i.e., trading-off modest performance and application accuracy for better row buffer locality). In summary, this work makes the following contributions.

- We demonstrate that delaying the scheduling of memory requests can significantly improve the overall row buffer locality because the memory controller can find more requests that can be scheduled back to back to the same row. Given that several GPGPU applications are latency tolerant, we do not observe notable performance reduction in such applications. To control the performance loss caused by delays, we devise a low-overhead dynamic mechanism that limits the delay by ensuring that utilization of DRAM stays above a certain threshold.
- We demonstrate that a small fraction of memory requests can cause a large fraction of

row activations (i.e., there is non-uniform reuse of row buffers). Therefore, approximating a limited number of requests (bounded by the prediction coverage) can significantly reduce the row energy, without notably degrading the output quality of error-tolerant GPGPU applications. To improve the row buffer locality more effectively under a limited prediction coverage, we devise a low-overhead dynamic mechanism that is able to prioritize the approximation of requests with relatively low row buffer localities.

• Our newly proposed lazy memory scheduler for GPUs realizes the aforementioned contributions via delayed memory scheduling (DMS) and approximate memory scheduling (AMS), respectively. We show that DMS and AMS can work separately or together while improving the effectiveness of each other. Our evaluation shows that across a variety of GPGPU applications, row energy can be reduced by 12% using DMS, 33% using AMS, and 44% using a combination of both schemes. We achieve these results with less than 1% IPC loss, with an acceptable loss in application accuracy, and without requiring additional buffer space beyond what already exists in the baseline memory controllers.

5.2 Background and Metrics

Table 5.1: Key configuration parameters of the simulated GPU.

SM Features	1400MHz core clock, 30 SMs, SIMT width = 32 (16 × 2)
Resources / Core	32KB shared memory, 32KB register file, Max. 1536 threads (48 warps, 32 threads/warp)
L1 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
L2 Cache	8-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 (MCs), FR-FCFS scheduling [98], 16 DRAM-banks/MC, 4 bank-groups/MC, 924 MHz memory clock, global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing, $t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$
Interconnect	1 crossbar/direction (30 SMs, 6 MCs), 1400MHz interconnect clock, islip VC and switch allocators

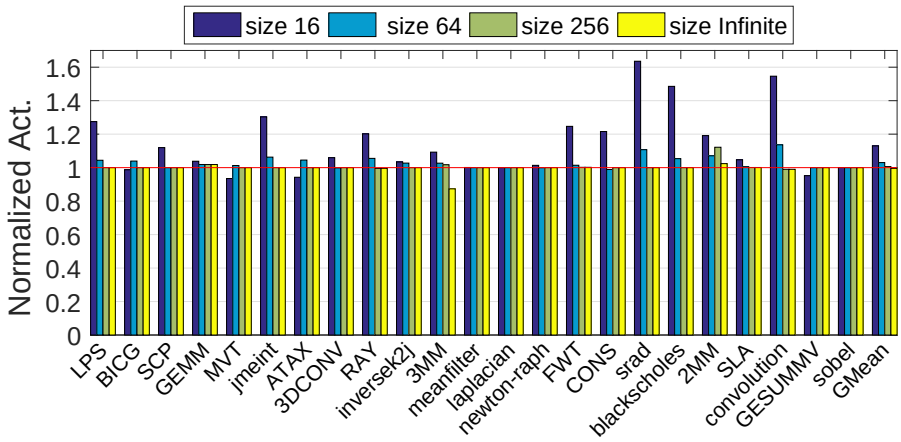


Figure 5.1: Effect of pending queue size on the number of row activations (Act.). Results are normalized to the case of pending queue size 128.

5.2.1 Evaluation Methodology and Metrics

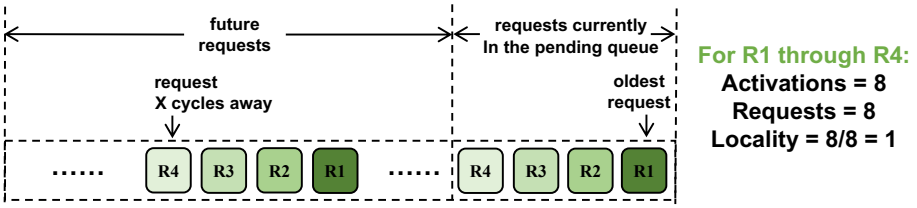
We use First-Row First-Come-First-Serve (FR-FCFS) with a open-row policy as our baseline memory configuration (Chapter 2.3). For a series of GPGPU applications, Figure 5.1 shows that the number of activations reduces (i.e., row buffer locality increases) with larger pending queue sizes. As the rate of decrease saturates after the size of 128, we use it as our baseline configuration. We evaluate the proposed techniques on a cycle-level GPU simulator – GPGPU-Sim [12] (Table 5.1) and collect energy-related measurements using GPUWattch [64]. We summarize the definitions that will be used in this work.

DRAM Locality-related Terminology. Row Buffer Locality (RBL) is defined as the number of requests that are scheduled back-to-back to the same DRAM row during the time it is activated in the row buffer. In this context, the notation RBL(X) would imply that X requests access the same row back-to-back before it is closed. The Average Row Buffer Locality (Avg-RBL) is defined as the ratio of the total number of memory requests to the total number of row activations. We also use the notation RBL(X - Y) to denote all the rows which have RBLs that belong to the range RBL(X) to RBL(Y).

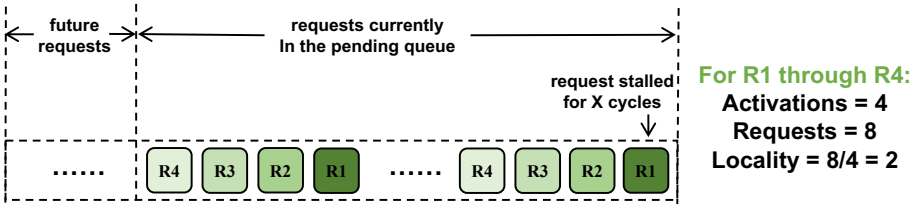
Delay-related Terminology. We define Delay as the minimum number of required cycles spent by every request in the pending queue before it can be considered for schedul-

ing. These required cycles are enforced by our proposed *delayed* memory scheduling (DMS) which will be introduced in the following sections. In this context, we use the notation DMS(X), where X indicates the minimum required cycles of delay, to denote the delay configuration of the pending queue. The largest value of X at which the application performance (in terms of Instructions-per-Cycle (IPC)) degrades no more than a user-defined percentage is defined as the Maximum Tolerable Delay (MTD). For our purposes, we tolerate up to 5% IPC degradation compared to the baseline but this number can also be changed by the user.

Approximation-related Terminology. The coverage is defined as the percentage of memory global read requests that are *not* served by the DRAM banks but instead *dropped* from the memory pending queue and returned immediately to the reply queue. It will then be recognized and *approximated* by the value predictor on its way back to the core. We consider these global read requests for approximation only when they are in rows with low RBLs. In this context, we define RBL-Threshold, Th_{RBL}, which is the value up to which the row is considered to have low RBL and hence those requests are the candidates for approximation. For example, if Th_{RBL} is equal to 3, it implies that all rows with RBL(1), RBL(2), and RBL(3) have low RBL. The dropping of requests in rows with low RBL is executed by our proposed *approximate* memory scheduling (AMS), which will be introduced in the following sections. We use AMS(Th_{RBL}) to denote the approximation configuration. The approximation conducted by AMS and value predictor can cause a certain level of output quality degradation, which we estimate with the application error. The application error is defined as the average relative error between the output of the baseline version of an application and the output of the same application with load value approximation. In general, higher coverage can lead to larger application error [139].



(a) Pending queue with the baseline FR-FCFS scheduling.



(b) Pending queue with DMS.

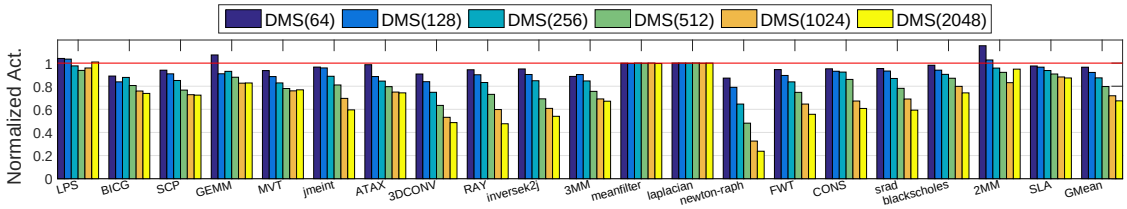
Figure 5.2: An example illustrating the benefits of delayed memory scheduling due to increased visibility to the memory controller. Eight requests are shown in total destined to four DRAM rows (R1, R2, R3, R4).

5.3 Motivation and Analysis

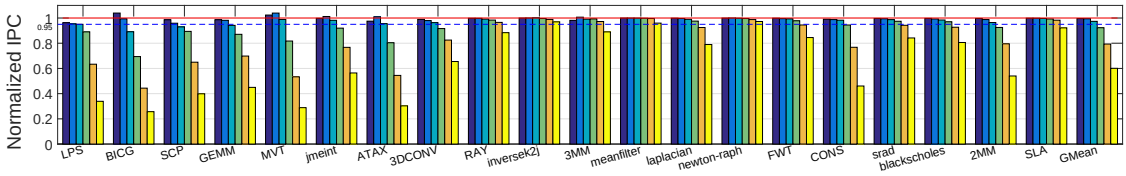
Our goal is to improve the average row buffer locality (i.e., Avg-RBL) by reducing the number of poorly reused rows. To this end, we propose two mechanisms: a) delayed memory scheduling (DMS), which trade-off scheduling delay (and potentially performance) for better Avg-RBL and b) approximate memory scheduling (AMS) which trade-off output quality for better Avg-RBL. In this section, we will motivate these trade-offs and show their effectiveness. We will also discuss how both these scheduling techniques can work together for even higher improvements in the Avg-RBL.

5.3.1 Delayed Memory Scheduling (DMS)

The baseline FR-FCFS scheduler attempts to schedule pending memory requests to the DRAM bank as soon as it is idle. Interestingly, we find that such timely scheduling of requests by the memory controllers actually disallows optimal reuse of data present in row buffers. To understand this observation, consider an illustrative example shown in Figure 5.2. The first scenario in Figure 5.2(a) depicts the baseline case of FR-FCFS



(a) Effect of delay on the number of row activations.



(b) Effect of delay on performance (IPC).

Figure 5.3: Effect of delayed memory scheduling on the number of activations and performance. Results are normalized to the baseline architecture (Section 5.2), which does not employ delayed or approximate scheduling.

scheduling. As shown in the figure, there are currently four pending requests in the memory controller's pending queue and these four requests belong to four different DRAM rows (R1, R2, R3, R4) of the same bank. Also, there are many more requests destined to the same bank but have not yet arrived at the pending request queue. Among such requests, there are four more requests that belong to the same four DRAM rows (R1, R2, R3, R4). For the baseline scheduler that timely issues all these requests, we find that the first four requests in the pending queue are issued back to back to the DRAM bank, leading to 4 activations for R1 through R4. When the remaining four requests arrive at the pending queue, four additional activations will also be required to serve them. Therefore, eight activations are required to serve all eight requests of R1 through R4, leading to an Avg-RBL of 1.

In order to improve the Avg-RBL, we propose the *delayed* memory scheduling (DMS). DMS carefully delays the issuing of each pending memory request in the hope that more requests destined to the same row of a bank will show up in the pending queue. To illustrate this, consider the case as shown in Figure 5.2(b) where the issuing of all requests have been delayed for X cycles. Hence, by the time the other four requests have reached

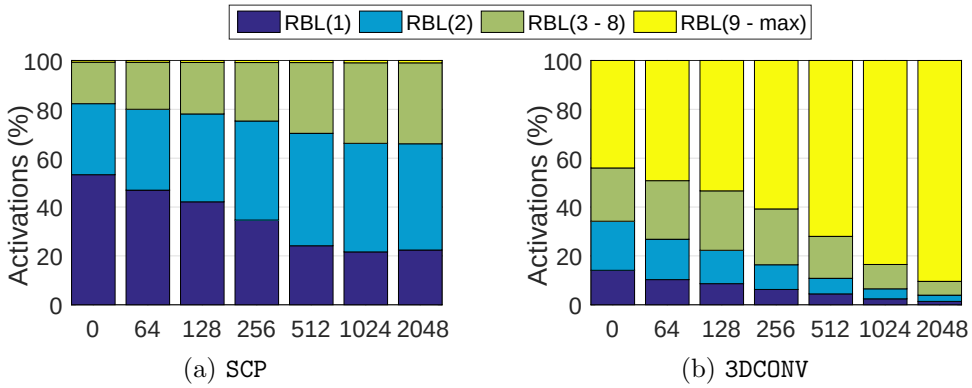


Figure 5.4: Effect of delayed memory scheduling on activation proportions of each RBL. x-axis indicates delay. y-axis indicates each component's proportion to the total number of activations.

the pending queue, the first four requests to R1 through R4 are still in the pending queue. Therefore, only four activations are required to serve all eight requests, leading to an Avg-RBL of 2 (twice of the baseline case).

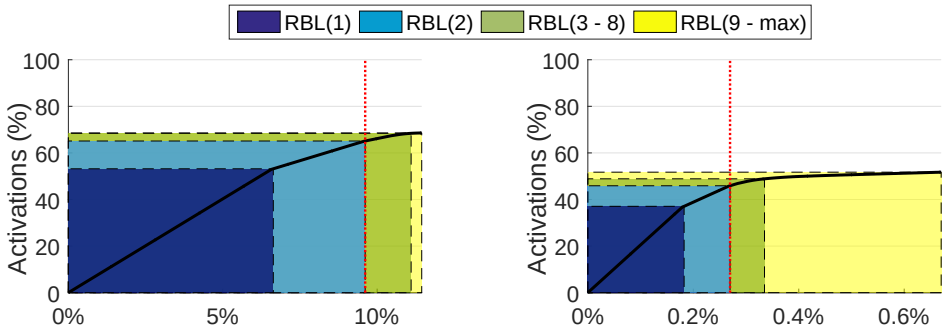
Figure 5.3(a) shows the normalized number of activations across a variety of GPGPU applications. For all of these applications, each of their requests (that does not lead to a row hit) is delayed by X cycles in the pending queue, denoted by DMS(X), before it can be served by a DRAM bank (more details are explained in Section 5.4). We show the results for when X is equal to 64, 128, 256, 512, 1024, and 2048 cycles. We find that many applications are sensitive to delay – the higher the delay, the higher the chance of finding requests destined to the same rows, which leads to fewer row activations. On average, the activation reduction can be as high as 31%, when a delay of 2048 cycles is used. Figure 5.4 shows the distribution of row activations based on their RBLs with different delays for two applications. As we observe, for both applications, the proportion of row activations with RBL(1) (i.e., only one request accesses the activated row before it is closed – Section 5.2.1) reduces significantly with the increase of delay. Meanwhile, the proportions of row activations with higher RBLs have increased. This shift in the RBL of row activations effectively shows how DMS can help to improve the Avg-RBL for real applications.

On the negative side, the increase in delay can degrade the overall performance. Thanks to the latency tolerance of GPGPU applications, the increase of delay has a limited impact on the performance as shown in Figure 5.3(b). Many applications retain their baseline performance up to 95% even at very large delays (e.g., 1024 cycles). However, IPC's sensitivity to delay varies for different applications and hence it is critical to determine an appropriate value of delay to carefully trade-off the activation reduction with the performance.

5.3.2 Approximate Memory Scheduling (AMS)

In order to further improve the Avg-RBL, we determine which pending requests have low RBLs and propose to return these requests immediately instead of issuing them to the DRAM banks. Subsequently, their values are approximated using existing techniques on their way back to the cores. Our proposal is motivated by the observation that for many GPGPU applications, a small portion of memory requests contributes to a high proportion of total row activations. The cause of this is multi-fold as it depends not only on the applications' algorithms and data placement mechanisms but also on the runtime behaviors driven by the warp or thread-block scheduling techniques. Nevertheless, as we will discuss further, our proposed techniques are also complementary to other optimizations that may improve Avg-RBL separately.

AMS works on row activations that only contain memory read accesses, as memory write accesses are typically not the targets for value approximation techniques. Figure 5.5 shows the proportion of row activations from the rows that are opened to serve only global read requests. We sort these requests in increasing order of their associated row activations' RBLs. Note that the x-axis denotes the proportion to the total number of requests. The y-axis denotes the proportion to the total number of activations. The shaded regions on the curve indicate the portions contributed by each RBL category. As shown in Figure 5.5(a), for GEMM around 10% of memory read requests associated with RBL(1) and RBL(2) cause about 65% of the total row activations. Similarly, as shown in

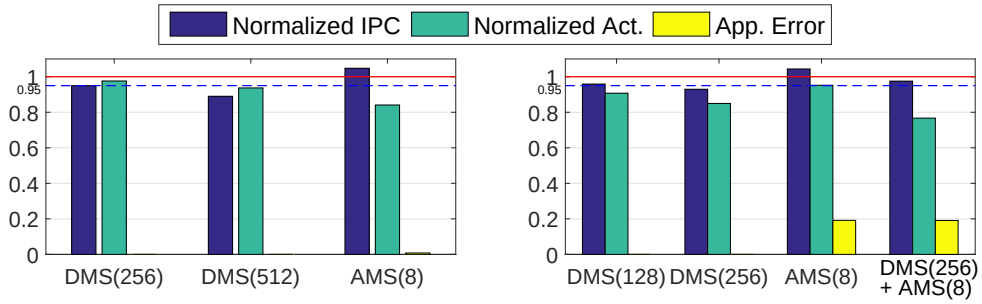


(a) CDF for GEMM (b) CDF for 3MM

Figure 5.5: The cumulative distribution of total row activations for requests associated with different RBLs. x-axis is the proportion of requests sorted by their RBLs.

Figure 5.5(b), for 3MM around 0.2% of memory read requests associated with RBL(1) and RBL(2) cause about 45% of the total row activations. This implies that a large fraction of row activations is caused by only a small fraction of memory requests.

In order to leverage this observation to further reduce row activations, we propose *approximate* memory scheduling (i.e., AMS). AMS first recognizes the pending read requests which are not destined to the same rows as any of the pending write requests. Then AMS decides if these requests are associated with low-RBL row activations, which means that the RBLs that these requests are expected to bring are no greater than a specific RBL-Threshold (i.e., Th_{RBL}, more details in Section 5.4). Subsequently, AMS returns these requests immediately without issuing them to the DRAM banks. Finally, the values of such requests will be provided by a value approximation technique on their way back to the cores. We denote this as AMS(Th_{RBL}). Such an approach eliminates these low-RBL row activations in the DRAM banks, thereby significantly improving the Avg-RBL and reducing the DRAM energy. On the negative side, such an approach can lead to application-level error, which needs to be acceptable to the user. To control the application-level error, the number of approximated requests (i.e., prediction coverage) needs to be limited. Thus, within the coverage limit, finding the row activations with relatively low RBLs among all the activations is the goal of AMS. Further details of AMS are in Section 5.4.



(a) LPS (b) SCP

Figure 5.6: Examples illustrating how approximate memory scheduling can help delayed memory scheduling.

5.3.3 Delayed and Approximate Scheduling

Having discussed the benefits of DMS and AMS separately, we now discuss how both DMS and AMS can work together to provide further benefits in terms of reducing the number of row activations and improving the performance. In this context, we consider the following two questions:

5.3.3.1 How can approximate memory scheduling help delayed memory scheduling

We find that AMS can help DMS especially for applications that belong to two categories:

Case 1. *The application's number of row activations is not sensitive to the change of delay.* For example, Figure 5.6(a) shows the normalized IPC and the normalized number of row activations for application LPS under three different cases. LPS has only a limited activation reduction (i.e., 2%) with its maximum tolerable delay (MTD) of 256 cycles. However, with a delay value of 512 cycles, LPS can reach its highest activation reduction (i.e., 6%), but also at the price of an IPC loss of 11%. On the contrary, if AMS is applied instead and approximates the requests associated with RBL(1-8) row activations (i.e., AMS(8)), LPS can get 16% activation reduction and 5% IPC improvement, only at the cost of less than 1% application error which is a minimal quality loss. Therefore, AMS is useful when DMS cannot effectively reduce the number of row activations as shown in

this case.

Case 2. *The application's number of row activations is sensitive to the change of delay, but the performance loss is preventing DMS from adopting higher delay values.* For example, Figure 5.6(b) shows different metrics for application `SCP` under four different cases. With DMS(128), the activation reduction can reach 9% at the cost of a 4% IPC loss. As the value of delay increases from 128 to 256, the activation reduction can further reach 15% at the cost of a 7% IPC loss. However, if we required that the performance loss must be under 5%, then DMS(256) should not be adopted and the further activation reduction cannot be achieved.

On the other hand, when applying AMS alone (the results of AMS(8) as shown in Figure 5.6(b)), the number of row activations reduces and also the IPC increases at the cost of increased application error. However, if we combine both DMS and AMS together (the results of DMS(256) + AMS(8) as shown in Figure 5.6(b)), `SCP` can adopt DMS(256) to obtain more activation reduction and still achieve less than 5% IPC loss. This means that the increase of IPC provided by AMS can compensate for the IPC loss caused by DMS. As a result, the value of delay can be further increased to obtain more activation reduction from DMS. In addition, AMS is able to work synergistically with DMS to further reduce the number of row activations, leading to a higher activation reduction. Therefore, AMS is useful to help increase the delay value in DMS as shown in this case.

5.3.3.2 How can delayed memory scheduling help approximate memory scheduling

We find that DMS can also help AMS in terms of activation reduction, as delaying the issuing of pending requests can help to more accurately identify the low-RBL row activations. To illustrate this, consider Figure 5.7, which shows that 9 requests are destined across 5 rows (i.e., R1 through R5) of the same DRAM bank and AMS is trying to find a request associated with an RBL(1) row activation to drop. Figure 5.7(a) shows a case when AMS is applied alone and there are 4 more requests destined to R1 through R4 of

the same bank that have not yet reached the pending queue. Also, the time required for the bank to serve a request is sufficient for these 4 future requests to reach the queue. Since that the memory scheduler only has visibility of the requests currently in the pending queue, it observes 5 RBL(1) row activations at this point. Therefore, if AMS were to choose a request to be dropped, it would drop the first R1 as it is the oldest pending request. However, this would lead to even an Avg-RBL decrease from 1.8 (9/5) to 1.6 (8/5). This is because the total number of activations for these 9 requests is still 5, but the total number of requests is reduced from 9 to 8 (the first R1 is dropped). AMS cannot accurately drop R5 because the row indexes of future requests are unknown.

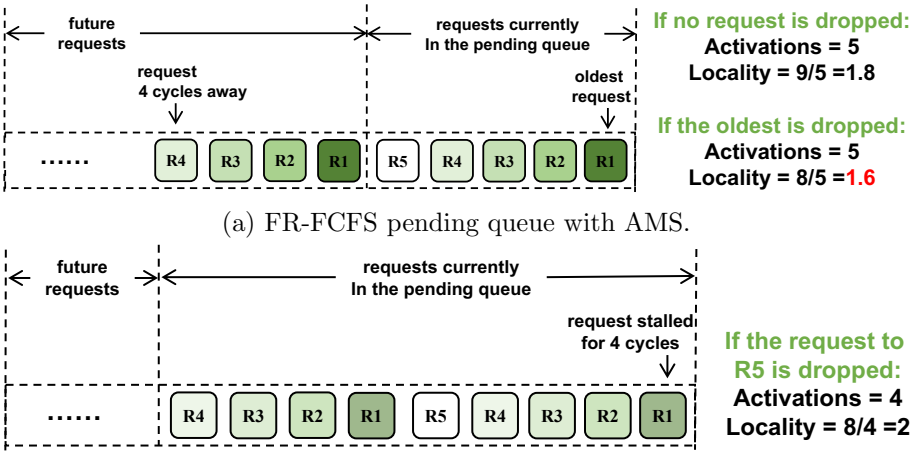


Figure 5.7: Example illustrating how delayed memory scheduling (DMS) can help approximate memory scheduling (AMS) by comparing different schemes.

On the other hand, Figure 5.7(b) shows the case when AMS is applied together with DMS. As a result of the added delay by DMS, AMS will correctly drop R5 as it can observe now that only R5 has an RBL(1) row activation. Hence, the total number of activations is reduced from 5 to 4, and the total number of requests is reduced from 9 to 8, leading to an Avg-RBL increase from 1.8 (9/5) to 2 (8/4). In this case, AMS can more accurately identify low-RBL row activations as more requests are visible in the pending queue on account of applying DMS.

In summary, we find that both DMS and AMS can provide significant benefits in terms

of activation reduction. Furthermore, they can also improve the efficiency of each other when applied together. In the next section, we will provide implementation details for both memory scheduling techniques.

5.4 Design and Operation

5.4.1 Overview

Figure 5.8 shows a high-level overview of our design. The L2 misses (A) are buffered at the pending queue after they arrive at the memory controller. These pending requests are then issued to the DRAM banks following the FR-FCFS scheduling policy (B) as soon as their destined DRAM banks become available (Section 5.2). Our proposal focuses on seamlessly integrating our new memory scheduling schemes: DMS and AMS with the baseline FR-FCFS scheduler. In this context, Figure 5.8 shows three major components (shaded in gray color) of the lazy memory scheduler: delayed memory scheduling unit (DMS unit), approximate memory scheduling unit (AMS unit), and value prediction unit (VP unit). The DMS and AMS units coordinate with the memory controller to decide *which* and *when* the requests should be issued to the DRAM banks. The AMS unit also coordinates with the VP unit to decide *which* and *how* the requests will be approximated. Consequently, these units decide the sequence of row activations of DRAM banks so as to maximize the Avg-RBL.

The DMS unit can either work independently or with the AMS unit. In the former case, before opening a new DRAM row, the DMS unit checks whether the oldest request has spent at least X cycles (i.e., DMS(X)) in the pending queue. If true, then this oldest request is issued to the memory banks (B) and its corresponding DRAM row is opened. The other pending requests destined to the same row are also issued back to back (regardless of their ages) as per FR-FCFS policy. To keep track of the delayed cycles per request, each request is assigned with a time stamp when it enters the pending queue. This time stamp is used by the DMS unit to check against the current time to get the value of delay (C)

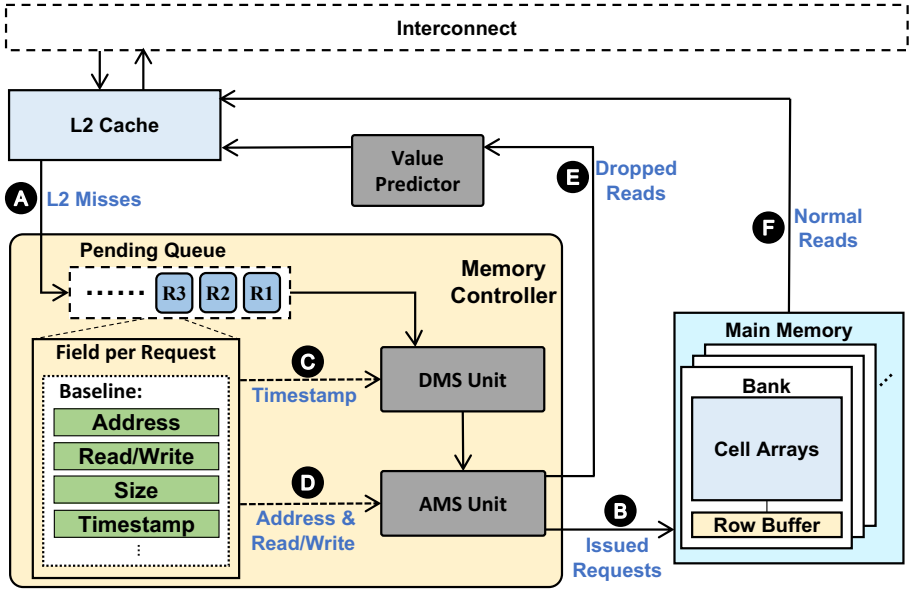


Figure 5.8: Design overview of the lazy memory scheduler and associated components.

(more details are in Section 5.4.2).

In the latter case, the DMS unit also checks whether the oldest request has spent at least X cycles in the pending queue. If true, it then checks the current prediction coverage, Th_{RBL}, and the pending requests' information Ⓓ to decide if this request should be dropped (more details are in Section 5.4.3). If all criteria are satisfied, the AMS unit will drop the request from the pending queue and send a dropped read signal Ⓔ to the VP unit to generate an approximate value. Otherwise, if the criteria are not satisfied, the request is issued to the memory banks Ⓑ, and the L2 cache is filled with accurate data served by the memory banks Ⓕ (the same as the baseline case).

5.4.2 Delayed Memory Scheduling Schemes

As discussed earlier in Section 5.3, finding an appropriate value for delay is important for DMS. Higher values of delay would create more opportunities for the memory scheduler to improve the Avg-RBL, however, at the possible loss of performance. In this context, we propose two schemes: Static-DMS and Dyn-DMS, which calculates the value of X statically and dynamically, respectively.

Static-DMS: Static Delayed Memory Scheduling. The Static-DMS uses a delay of 128 cycles (i.e., DMS(128)), based on our empirical evaluations. As shown in Figure 5.3, 128 cycles is the maximum delay that can lead to less than 5% IPC losses across all tested applications. However, this static value of delay misses out on the opportunity of improving Avg-RBLs in applications with higher latency tolerances. It may also lead to more than 5% IPC losses in untested applications. Therefore, we further propose a scheme that dynamically decides the value of delay based on the latency tolerance of an application.

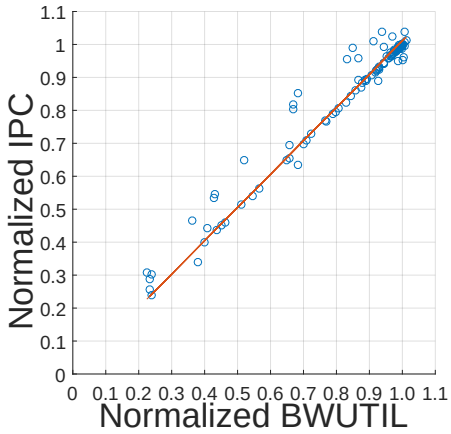


Figure 5.9: Illustrating the relationship between IPC and BWUTIL.

Dyn-DMS: Dynamic Delayed Memory Scheduling. We propose a profiling-based dynamic scheme, which is based on the fact that the performance degradation can be tracked locally at the memory controller via observing the bandwidth utilization (BWUTIL). For all the applications we used, we tested their BWUTILs and IPCs with different values of delay. As shown in Figure 5.9, their BWUTILs and IPCs are linearly correlated, which is also confirmed in previous works [47, 130]. For this reason, we can track the changes in DRAM bandwidth utilization locally at the memory controller to keep track of the changes in the overall performance.

Our Dyn-DMS mechanism is an iterative mechanism that attempts to find the maximum value of delay such that performance (reflected by bandwidth utilization) does not

drop significantly (our threshold is 5%) compared to the baseline no-delay scenario. Dyn-DMS first samples the baseline BWUTIL for a window of 4096 memory cycles.[1] Note that in order to accurately sample the baseline BWUTIL, the co-running AMS scheme is temporarily halted during this window when applying DMS and AMS together. Then starting from a delay value of 128 cycles, the DMS unit gradually increases the value of delay (X) for the following 4096-cycle windows in steps of 128 delay cycles. At a particular delay, if the BWUTIL of that window starts to drop below 95% of the baseline, the iterative method stops and set the delay to be the last value that leads to a BWUTIL more than 95% of the baseline. This delay value X is also recorded. To capture the phases changes within an application, we restart the process after every 32 windows, however, we set the previously recorded delay value of X as the starting point for this iterative procedure to quickly settle to the optimal value. Note that the maximum value of X we use is 2048 and the minimum is 0 (baseline case).

5.4.3 Approximate Memory Scheduling Schemes

As discussed earlier in Section 5.3, with a coverage limitation, finding and dropping requests associated with relatively low RBLs are more favorable to reduce the number of activations. Therefore, an appropriate value for Th_{RBL} is important for AMS(Th_{RBL}). High values of Th_{RBL} would lead to unnecessarily approximating requests associated with high RBLs and wasting the limited prediction coverage. On the other hand, low values of Th_{RBL} may not provide enough opportunities to approximate if there are not enough requests associated with low RBLs, thereby limiting the potentials of Avg-RBL improvements.

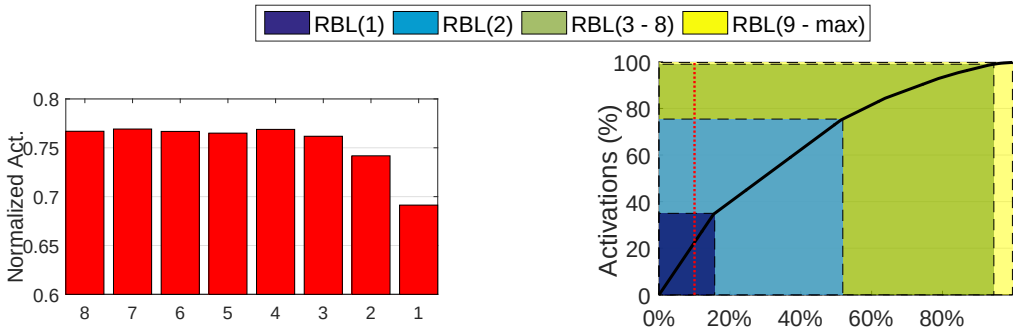
The working procedure of the AMS unit has multiple steps. As using approximate value for critical data (e.g., pointers) may cause fatal errors for applications, we use *pragma* to annotate the approximable regions of data to guarantee the safety of applying

[1]Based on our experiments, 4096 cycles is a suitable window size. An overly large window does not timely reflect the current BWUTIL, meanwhile, an overly small window is too sensitive to local spikes in BWUTIL (or coverage).

value approximation. Hence, the AMS unit will only proceed if it detects that the oldest pending request is approximable. Second, the AMS unit verifies if the oldest request satisfies the delay criteria determined by DMS. Third, if the first criterion is satisfied, the AMS unit calculates the coverage based on the total number of requests dropped and the total number of requests received so far. It then checks if the coverage is less than the user-defined coverage value (we use 10%). Fourth, if the second criterion is also satisfied, the AMS unit iterates through the pending queue to obtain the RBL value associated with the request and checks if it is less or equal to Th_{RBL}. Also, during this iteration, the AMS unit ensures that all the other requests destined to the same row are global read requests, as we only approximate load values. If the fourth criterion is also satisfied, then this request will be dropped from the pending queue, instead of being issued to the memory bank. In addition, all other pending requests destined to the same row will also be dropped sequentially in the following memory cycles. If any of these three steps are not successful, as default, the request will be issued to the memory banks following the FR-FCFS policy.

We propose two schemes to realize the above goals and procedures: Static-AMS and Dyn-AMS, which calculates the value of Th_{RBL} statically and dynamically, respectively.

Static-AMS: Static Approximate Memory Scheduling. Based on our empirical evaluations, we found that the Th_{RBL} value of 8 is appropriate as it does not allow unnecessary approximations for requests associated with very high RBLs and at the same time provides enough prediction coverage for many applications. Therefore, AMS(8) is used for Static-AMS scheme. However, as different applications have very different RBL distributions, a static Th_{RBL} can be sub-optimal for some of the applications. For example, if the Th_{RBL} is too high for them, AMS cannot accurately target requests associated with lower RBLs under a limited prediction coverage. On the other hand, if the Th_{RBL} is too low for them, AMS cannot effectively reduce the number of activations because there are not enough requests for it to approximate. Therefore, there is a need to dynamically modulate the value of Th_{RBL} so as to more accurately target the low-RBL row activations



(a) The effect on activation reduction by reducing Th_{RBL} (x-axis) for `SCP`.

(b) CDF for `SCP`'s activations. x-axis is request percentage of read-only rows sorted by RBL.

Figure 5.10: Effect of reducing Th_{RBL}.

while also maintaining the user-defined coverage (10%).

Dyn-AMS: Dynamic Approximate Memory Scheduling. For some applications, the Static-AMS (i.e., AMS(8)) may be suboptimal. For example, as shown in Figure 5.10(a), application `SCP`'s number of activations can be further reduced when Th_{RBL} is reduced from 8 to 1. The reason for this can be explained with Figure 5.10(b). As shown in the Figure, most of the requests within the Th_{RBL} of 8 are associated with RBL(2 - 8). However, there are already more than 10% of the total requests associated with RBL(1) (i.e., the portion on the left of the red dashed line). Therefore, a Th_{RBL} value of 1 is most beneficial, as approximating 10% requests associated with RBL(1) leads to the highest activation reduction. Hence, dynamically modulating the Th_{RBL} is necessary to further improve the activation reduction for applications like `SCP`.

Based on this observation, we designed a profiling-based Dyn-AMS scheme. Similar to the Dyn-DMS, the Dyn-AMS is also an iterative approach that attempts to find the lowest value of Th_{RBL} such that the prediction coverage does not drop below the user-defined value. Note that we empirically use 10% coverage throughout the work and the Th_{RBL} range we use in the Dyn-AMS is 1 to 8. The AMS unit starts with the Th_{RBL} value of 8 and samples the coverages for consecutive windows of 4096 memory cycles. First, as long as the coverage can achieve the user-defined coverage, the AMS unit gradually decreases the Th_{RBL} value in steps of 1 in consecutive 4096-cycle windows. Second, once

the coverage goes below the user-defined coverage in a window, the AMS unit gradually increases the Th_{RBL} value in steps of 1 until the coverage returns to the user-defined coverage again in consecutive 4096-cycle windows. These steps are repeated until the end of application execution.

5.4.4 Value Prediction Unit

The Value Prediction Unit (VP unit) is responsible for approximating the values of requests that are dropped by the AMS unit. Since the VP unit works independently and is orthogonal to the memory scheduling schemes, we can support a large variety of previously proposed value prediction mechanisms such as [104, 73, 139, 103]. Similar to prior works, AMS uses programmer annotations to bound the approximation errors as the criticality of instructions presumably could only be identified by the programmer [14, 69, 102, 138, 88]. AMS requires the following information from the programmer, as shown in the example of Listing 5.1: a) the approximable loads which are error tolerant, and b) the prediction coverage which limits the total number of approximations.

```
#pragma pred_coverage{10%}
#pragma pred_var{B}
C[i] = A[i] + B[i];
```

Listing 5.1: Example of Code Annotation.

To demonstrate how AMS works, we designed a simple but effective VP unit that is based on the intuition that nearby addresses may store similar values and hence the value of a cache line can be approximated by a nearby cache line with limited error [104]. In order to predict the values for the dropped requests, we search in the nearby cache sets of the L2 cache and use the values from cache lines with nearest addresses as their approximate values.[2] To minimize the searching overhead, we carefully choose the search radius of

[2]In this simple model, we did not consider the error propagation caused by the reuse of approximated cache lines. However, we have tested with a more advanced model (that considers reuse) and have observed similar application error results.

nearby sets and take advantage of the existing associative search hardware to search in the cache ways of a set. We find that the searching overhead is negligible compared to the performance improvement introduced by value approximation. We will discuss the performance and output quality results in Section 5.5. Note that we first warm up the L2 cache with a sufficient number of requests to prepared for the searches, and thus AMS is initially disabled until the cache is ready.

5.4.5 Hardware Overhead

The DMS unit requires one comparator and one adder to do comparisons for the DMS functionalities. One 16-bit counter is required for Static-DMS and Dyn-DMS to store the current delay value of X. For Dyn-DMS, the DMS unit requires one 32-bit counter to store the baseline BWUTIL, one 32-bit counter to store the current BWUTIL, one 16-bit counter to store the cycles during profiling, one 8-bit counter to store the number of windows during profiling. The AMS unit requires one multiplier, one adder and one comparator for the operations of AMS. Static-AMS and Dyn-AMS require 1 bit to store the read/write condition and 1 bit to store the current memory space condition for the row of the oldest request, two 64-bit counters to store the total number of requests and approximated requests for calculating coverage, one 8-bit counter to store the RBL of the current request's row, one 8-bit counter to store the current Th_{RBL}, one 32-bit counter to store the index of the dropped request's row. For Dyn-AMS, the AMS unit requires one 16-bit counter to store the cycles during profiling.The VP unit requires nine adders, one MUX, one comparator for searching the nearest cache line, one 8-bit counter to store the radius, one 64-bit counter to store the tag of the dropped read request, two 64-bit counters to store the minimal tag distance and its corresponding address. Overall, the lazy memory scheduler requires 1 multiplier, 11 adders, 1 MUX, 3 comparators and 498 bits of buffer space in addition to the baseline memory controller. We believe this hardware overhead is modest compared to the energy savings provided by DMS and AMS. Finally, our mechanisms do not require any modification to the existing DRAM protocols.

Table 5.2: List of evaluated GPGPU applications. See Table 5.3 for more details.

Abbr.	Input	Group	Thrashing Level	Delay Related		Approximation Related	
				Delay Tol.	Act. Sens.	Th_{RBL} Sens.	Err. Tol.
RAY [12]	Matrix	3	High	High	High	Low	High
inversek2j [137]	Coordinates	3	High	High	High	Low	High
newtonraph [137]	Image	4	High	High	High	Low	Low
FWT [12]	Matrix	4	High	Medium	High	High	Low
MVT [93]	Matrix	2	High	Medium	High	Low	High
jmeint [137]	Coordinates	2	High	Medium	High	Low	Medium
ATAX [93]	Matrix	4	High	Medium	High	Low	Low
3DCONV [93]	Matrix	2	High	Medium	High	Low	Medium
CONS [93]	Matrix	4	High	Medium	High	Low	Low
srad [12]	Image	4	High	Medium	High	Low	Low
LPS [12]	Matrix	1	High	Medium	Low	High	High
BICG [93]	Matrix	1	High	Low	High	High	Medium
SCP [12]	Matrix	1	High	Low	High	High	Medium
GEMM [93]	Matrices	4	High	Low	Medium	High	Low
blackscholes [137]	Matrix	4	Medium	Medium	High	High	Low
2MM [93]	Matrices	4	Medium	Medium	Medium	Low	Low
3MM [93]	Matrices	3	Low	High	High	Low	High
SLA [12]	Matrix	4	Low	High	Medium	Low	Low
meanfilter [137]	Image	3	Low	High	Low	Low	High
laplacian [137]	Images	3	Low	Medium	Low	Low	Medium

5.5 Experimental Results

We evaluate our lazy memory scheduling techniques on a wide range of applications described in Table 5.2. The applications are selected so as to cover all important features that are relevant to our schemes. We list these features and their intensity classifications (e.g., Low, Medium, High) in Table 5.3. We use annotations to make sure that we only approximate global read requests which do not contain pointers or lead to fatal errors so that value approximation can be applied to all applications safely. For the ease of presenting results in this section, we group these applications into 4 different groups:

Group-1: These applications have high or medium error tolerance and also show high Th_{RBL} sensitivity. Therefore, both AMS and DMS can be applied and likely to benefit.

Group-2: These applications have high or medium error tolerance, thus the AMS related schemes can be applied. However, they show low Th_{RBL} sensitivity, so Dyn-AMS may not show clear benefits in terms of activation reduction.

Group-3: These applications have high or medium error tolerance, thus the AMS related schemes can be applied. However, since they either have very few requests associated with

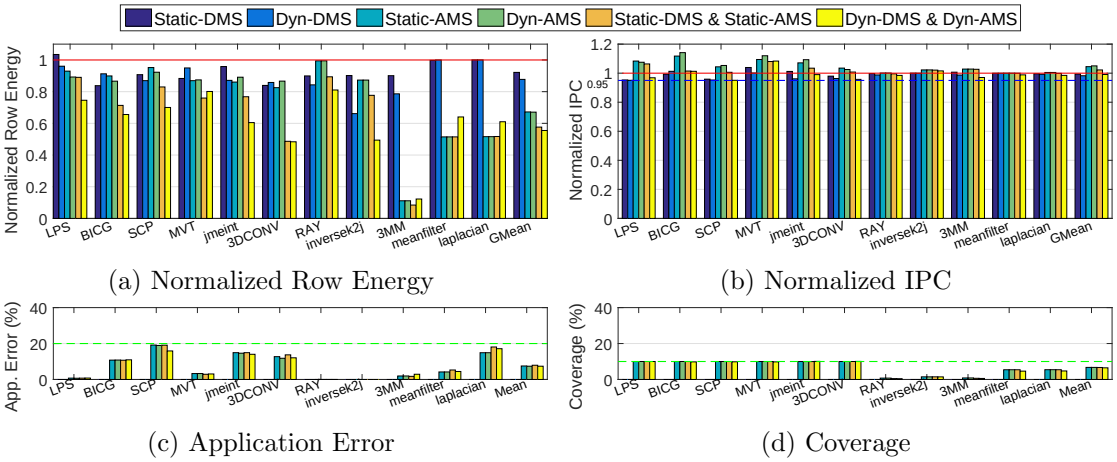
Table 5.3: Application features and intensity classifications. The thresholds are used only to facilitate the discussion in Section 5.5.

Feature	Description	Categories (by X Range)		
		Low	Medium	High
Thrashing Level	The application has $X\%$ requests in rows with RBL(1 - 8).	$[0, 3)$	$[3, 10)$	$[10, 100)$
Delay Tolerance	The application has a MTD of X.	$[0, 256)$	$[256, 1024)$	$[1024, +\infty)$
Activation Sensitivity	The application's activation reduction is $X\%$ compared to the baseline when 2048 cycles delay is applied to the FR-FCFS pending queue.	$[0, 10)$	$[10, 20)$	$[20, 100)$
Th_{RBL} Sensitivity	The application's maximum activation reduction is $X\%$ compared to the baseline when reducing its Th_{RBL} from 8 to lower values.	$[0, 5)$	NA	$[5, 100)$
Error Tolerance	The application shows $X\%$ application error when using our proposed value approximation technique (Section 5.4.4) at 10% coverage or its maximum available coverage less than 10%.	$[20, +\infty)$	$[5, 20)$	$[0, 5)$

RBL(1 - 8) (Low Thrashing Level), or have very limited rows that are only accessed by read requests when opened, their coverages cannot reach 10%.

Group-4: These applications have low error tolerance and thus the AMS related schemes should not be applied. However, for these applications, the DMS schemes can still be applied for reducing the number of row activations.

Effect on Row Energy. Figure 5.11(a) shows the normalized row energy across all schemes. We make four observations. First, overall the Static-DMS and Dyn-DMS are able to reduce row energies by 8% and 12%, respectively. Second, overall the Static-AMS is able to reduce 33% of row energy, which is more than that of the Static-DMS schemes. The Dyn-AMS does not show improvement over the Static-AMS for Group-2 and Group-3 applications. However, Group-1 applications overall show 7% row energy reduction in the Static-AMS and 11% in the Dyn-AMS. Third, for Group-1 and Group-2 applications, when combining Static-DMS and Static-AMS together, their average row energy reduces by 27%. This is 7% more than when Static-DMS and Static-AMS are applied separately. When combining Dyn-DMS and Dyn-AMS together, it shows the largest row energy reduction of 34%. This reduction is 7% more than when applying Static-DMS and Static-AMS together, and is 13% more than the total reduction of when applying Dyn-DMS and Dyn-AMS separately. Finally, when applying Dyn-AMS together with Dyn-DMS, Group-1,



(a) Normalized Row Energy

(b) Normalized IPC

(c) Application Error

(d) Coverage

Figure 5.11: Comparison of different schemes with different metrics for applications with Medium or High Error Tolerance. Row Energy and IPC results are normalized to the baseline that does not adopt DMS or AMS.

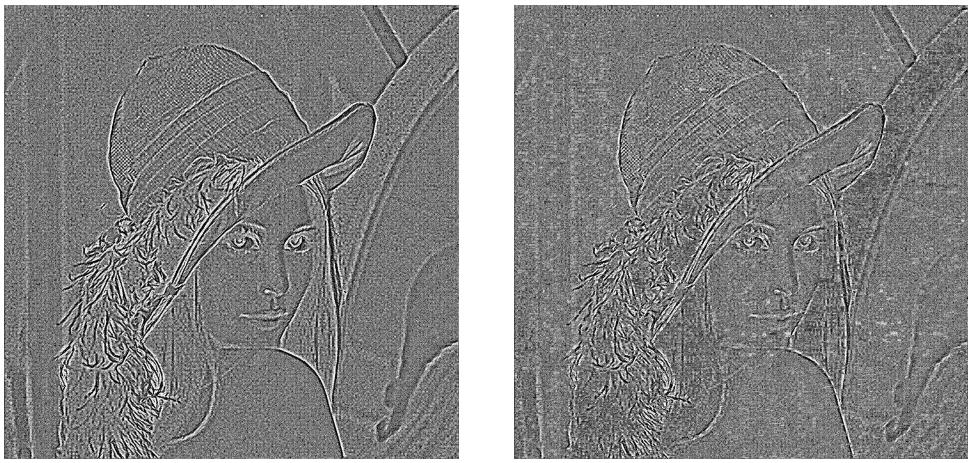
Group-2 and Group-3 applications overall achieve 44% row energy reduction. However, a few Group-3 applications (i.e., `3MM`, `meanfilter`, `laplacian`) show less row energy reduction than the other AMS related schemes. This is due to a small coverage decrease as shown in Figure 5.11(d) because Group-3 applications already have limited coverage and the profiling of Dyn-DMS reduces the number of requests dropped by Dyn-AMS (Section 5.4.2).

Effect on Memory Energy and Peak Bandwidth. The lazy memory scheduler's benefit in row energy reduction is caused by the improvement of the application's Avg-RBL. Therefore, it is independent of the memory technology used as long as it adopts similar structures as the row buffer. However, system-wise, its energy reduction is dependent on the memory technology. For example, if we apply Dyn-DMS and Dyn-AMS together on HBM1 where row energy constitutes nearly 50% of the memory system energy [16], we observe on average 22% memory system energy reduction with our tested applications. Similarly, for HBM2 where row energy can constitute 25% of its total energy, we observe on average 11% memory system energy reduction. Traditionally, the overall power budget of a high-end GPU card is limited to around 300W and its memory power budget is generally capped at 60W when operating at peak bandwidth. [83]. Therefore, in terms

of absolute savings with HBM2, the lazy memory scheduler can achieve: a) up to 8W memory power reduction while achieving the same peak bandwidth or b) up to 90 GB/sec higher peak bandwidth under the same 60W memory power budget.

Effect on Performance. Figure 5.11(b) shows the changes in IPC across all schemes. Overall, we find that all our schemes do not lose more than 5% IPC. We make three observations. First, the Static-DMS and Dyn-DMS show larger IPC losses because of the additional delay. Also, the IPC of Dyn-DMS can approach closer to the 95% threshold, resulting in more row energy reductions. Second, the Static-AMS and Dyn-AMS show IPC improvement. Specifically, overall Dyn-AMS shows more improvement than Static-AMS, indicating that it can improve more performance by potentially dropping the requests in rows with lower RBLs. Finally, when combining Static-DMS and Static-AMS together, overall the IPC improves by 2%. When combining Dyn-DMS and Dyn-AMS together, overall the IPC loss is less than 1%. Both cases show higher IPC than the Static-DMS or Dyn-DMS scheme, because of the usage of AMS. We conclude that all our schemes are able to effectively restrict the IPC loss to be less than 5%. Specifically, AMS can help to compensate for the IPC loss caused by DMS. The combination of DMS and AMS can provide a good trade-off between row energy reduction and performance loss.

Effect on Application Error. Figure 5.11(c) shows application errors across all schemes. Note that the application error for the Static-DMS and Dyn-DMS are all zeros because no approximation is applied. We find that with our VP unit design, different applications show different application errors, meanwhile for each application, there are only small differences of application error with similar prediction coverages (Figure 5.11(d)) across different schemes. With the 10% coverage limitation, the average application error is 7% for all the AMS related schemes. Figure 5.12 shows the image output of application `laplacian` for the accurate baseline case and the Dyn-DMS and Dyn-AMS combination case. We observe that with 17% application error, the image shows a limited level of quality degradation. We conclude that under our VP unit design, limiting the coverage is an effective way to limit the application error. Moreover, value approximation is a feasible



(a) Accurate Output (b) Approximate Output

Figure 5.12: Comparison between the accurate and the approximate output (which has 17% Application Error and is generated when the Dyn-DMS and Dyn-AMS schemes are applied together) for application `laplacian`.

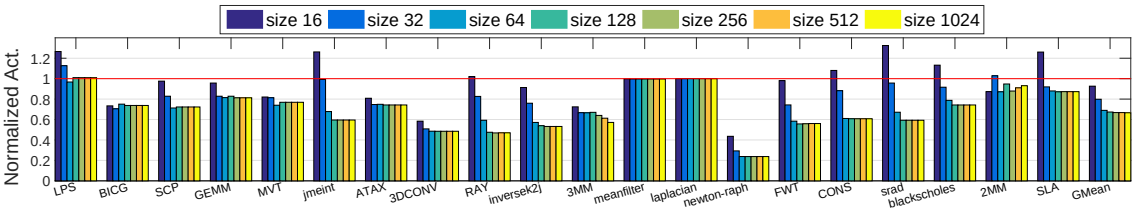
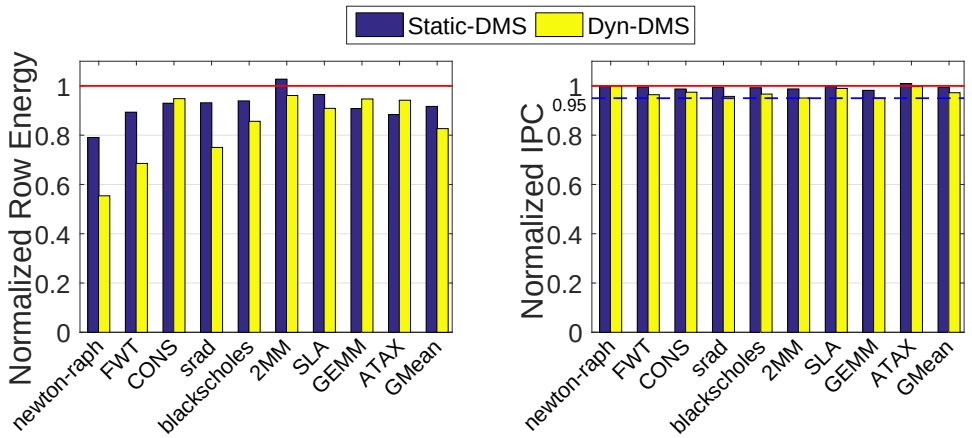


Figure 5.13: Effect of pending queue size on the number of activations (normalized to the baseline) with DMS(2048).

way to reduce row energy and improve performance as many applications can tolerate certain levels of error and are suitable for applying the AMS schemes. We also expect to see significant application error reduction if the AMS related schemes are applied together with the previously proposed value prediction techniques [104, 73, 139, 103] because they are more sophisticated and have shown much less application output quality loss when working with the same 10% coverage limitation.

Effect of FR-FCFS Pending Queue Size. When applying DMS, more requests are likely to be piled up in the pending queue, increasing the possibility to find row hits. However, if the pending queue is frequently full, future requests may often be blocked from entering it, limiting the Avg-RBL improvement of DMS. Therefore, it is important



(a) Normalized Row Energy (b) Normalized IPC

Figure 5.14: Comparison of different schemes in the delay-only mode for applications with Low Error Tolerance.

that the pending queue size is sufficient to support the increased pending requests in DMS. Figure 5.13 shows the effect on the number of row activations when using different pending queue sizes with the maximum allowed delay (i.e. DMS(2048)). And starting from size 128 the activation numbers for all applications tend to be stable. We conclude that a pending queue size of 128 (i.e., the baseline size) is sufficient to apply DMS.

Delay-Only Mode for Low Error Tolerance Applications. For applications with low error tolerance, even if AMS cannot be applied, we can still use DMS to reduce their row energy. Figure 5.14(a) and (b) show normalized row energy and IPC, respectively for Group-4 applications with the DMS schemes. We make two observations. First, both Static-DMS and Dyn-DMS can reduce row energy for Group-4 applications (one outlier is Static-DMS for application `2MM`). Also, Dyn-DMS can more effectively reduce row energy than Static-DMS. Second, both Static-DMS and Dyn-DMS have less than 5% IPC loss. And the IPC of Dyn-DMS can approach closer to 95% of the baseline. We conclude that for applications with low error tolerance, the DMS schemes can still effectively reduce their row energy with no more than 5% IPC loss. Dyn-DMS reduces more row energy by trading off a little more performance.

5.6 Related Work

To the best of our knowledge, this is one of the first works in the context of GPUs that consider the interplay between memory scheduling and application's tolerance to latency and errors. Our mechanisms achieve significant memory system energy savings while allowing the underlying hardware to remain dependable both in terms of performance and correctness [77, 121, 78]. Several prior works in the CPU domain [140, 143, 116, 144, 74, 76, 29] have focused on improving the row buffer locality. The goal of these works was to reduce the DRAM access latency because it is a first-order performance concern in single-threaded CPU workloads [15, 19, 36]. Other memory scheduling techniques for CPUs propose to partially delay the write request [74, 76], or conditionally employ an open-row policy [29] to improve the row buffer locality. But the purposes of these works are still to reduce the overall DRAM access latency. In contrast, DRAM access latency is not a primary concern in GPGPU applications as GPUs are capable of hiding long memory access latencies by spawning thousands of concurrent threads. Hence, in this work, we exploited this property to further enhance the row buffer locality for GPU memory.

In the context of GPUs, Jog et al. [51] proposed a criticality-aware memory scheduling mechanism to trade-off row buffer locality for servicing latency-critical requests. However, it will likely increase the DRAM energy consumption due to sub-optimal row buffer locality. Prior work on warp scheduling and throttling policies [50, 49, 54] can also improve the row buffer locality. However, these throttling/warp-scheduling decisions and memory scheduling decisions do not always remain in sync as they are taken physically far away from each other and are conducted at different granularities. This makes it important to design new memory scheduling decisions (as we do in this work) that consider the current DRAM status. Moreover, we believe our work is complementary to these prior works as they can provide additional benefits by shaping the access patterns such that they can benefit DMS and AMS.

5.7 Chapter Summary

This chapter focused on improving the DRAM row buffer locality in GPUs to reduce the memory system energy consumption. To this end, we proposed a lazy memory scheduler that can work in two modes: delayed or approximate. In the delayed mode, it carefully delays the scheduling of memory requests to allow more of them to accumulate at the memory pending queue. Such a mechanism increases the visibility of the memory scheduler thereby improving the chances of finding more requests that can be served by reusing the data in the row buffer. In the approximate mode, it carefully identifies a small fraction of requests with low row buffer locality and does not issue them to the DRAM banks. Instead, a simple but effective value predictor can be used to approximate the values for such requests. We also find that both these modes are synergistic and improve the effectiveness of each other when employed together. Our evaluation across a variety of GPGPU applications shows that row energy can be reduced by 12% with delayed memory scheduling, 33% with approximate memory scheduling, and 44% with a combination of both schemes. We hope that this work can open up new research directions that consider the interactions between scheduling, error resilience, and latency tolerance techniques at different levels of the memory hierarchy.

Chapter 6

Towards Architectural Support for Flexible Data Precisions

Despite the energy consumption of preserving the stored data, a large proportion of GPU memory energy is spent on transferring data across the memory hierarchy. This movement of data consumes the limited available bandwidth and power budget of GPU memory. Meanwhile, the energy consumed by data movement also depends on the value of the data itself. For example, when data transfers on the data IO bus of the memory, a bit value of 1 consumes considerably more energy than a bit value of 0 for the data's binary representation. Therefore, both the quantity and the bit values of the data transferred in the memory are important factors to determine the total memory energy consumption.

The goal of this work is to improve the memory energy efficiency by reducing the amount of data transferred and the per-bit energy cost of data movement. We observe that for several GPGPU applications (e.g., machine learning, image processing, etc.), the quantity of the data transferred can be reduced as not all values of bits in their data are required to perform an accurate or highly precise computation. Also, we observe that the bit values of the data transferred are sometimes not energy efficient because of the way the values are represented with the floating-point format. Furthermore, not all bit positions in the data's binary representation contribute equally towards the total memory

energy consumption. Based on these observations, we propose a novel memory system architecture to improve memory energy efficiency. Overall, this work makes the following contributions. First, through a detailed analysis of the floating-point format, we observe that different bytes positions in the data do not contribute equally to the energy consumption. Second, we propose a new data format to store the data, which requires minimal changes to the standard IEEE floating-point format. This data format is highly flexible and efficient in terms of data type conversion. Third, we make a case for a novel memory system architecture with a complete memory hierarchy that can dynamically determine the transfer energy and precision requirement of data. Therefore, data movement can be reduced by transferring fewer bytes when the precision requirement of data is low. Together with our newly proposed data format, this new memory architecture can convert data types locally at the DRAM banks without fetching them out of the memory. Finally, we will show the detailed evaluation results in the future.

6.1 Background and Metrics

6.1.1 Floating-Point Data Storage Formats

The floating-point data storage format is used to store floating-point numbers, as opposed to the integer data storage format which can only store integer numbers. Depending on the requirement, the floating-point format has different configurations. For example, the single-precision floating-point, or FP32, uses 32 binary bits to store floating-point data. As per the IEEE 754-2008 standard, FP32 has 1 sign bit, 8 exponent bits, and 24 significand bits, as shown in Figure 6.1. To calculate the value that the format represents, we can simply use equation 6.1 with all the given bit values. As shown in the equation, in most of the cases, the format just represents normal values. The normal values can be calculated with the use of sign bits, exponent bits, significand bits, and an offset for the exponent. This offset is usually format-specific and is used to adjust the magnitude coverage of the format. We have summarized the offsets as well as other configurations for some commonly

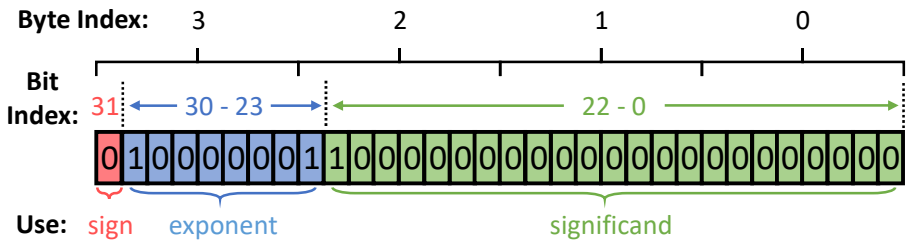


Figure 6.1: FP32 layout.

used floating-point formats in Table 6.1.

Also, there are four special cases of the floating-point value calculation. When the exponent bits equal their maximum representable value (i.e., all bits are 1), it can represent NaN (i.e. not a number) or Infinity cases, depending on the significand bits. If the exponent bits equal their minimum representable value (i.e., all bits are 0), then the format can represent 0 or subnormal values (i.e., values that are smaller than the smallest normal value), depending on the significand bits. As an example, we will show the value calculation process of the floating-point number shown in Figure 6.1. First, we recognize that the exponent is 129, which is neither the maximum nor the minimum value of the exponent. Therefore, secondly, we use the equation for normal value with sign equal 0, exponent equal 129, offset equal 127 (see Table 6.1), and 1.significand (a binary fraction) equal 1.5. This will give us the final result of 6, which is the value that the floating-point data represent.

$$Value = \begin{cases} (-1)^{sign} \times Infinity, & \text{if } exponent = Exp.Max \text{ and } significand = 0 \\ NaN \text{ (not a number)}, & \text{if } exponent = Exp.Max \text{ and } significand \neq 0 \\ 0, & \text{if } exponent = 0 \text{ and } significand = 0 \\ (-1)^{sign} \times 2^{1-offset} \times 0.significand, & \text{if } exponent = 0 \text{ and } significand \neq 0 \textbf{ (subnormal value)} \\ (-1)^{sign} \times 2^{exponent-offset} \times 1.significand, & \text{otherwise } \textbf{(normal value)} \end{cases} \tag{6.1}$$

Table 6.1: Configurations of common floating-point data formats.

Format	Description	Sign	Exponent	Exp. Max	Significand	Offset	Actual Exp. Range
FP64	IEEE double precision	1 bit	11 bits	2047	52 bits	1023	[-1022, 1023]
FP32	IEEE single precision	1 bit	8 bits	255	23 bits	127	[-126, 127]
FP16	IEEE half precison	1 bit	5 bits	31	10 bits	15	[-14, 15]
BF16	Brain floating point	1 bit	8 bits	255	7 bits	127	[-126, 127]
TF19	Nvidia Tensorfloat	1 bit	8 bits	255	10 bits	127	[-126, 127]
FP24	AMD fp24	1 bit	7 bits	127	16 bits	63	[-62, 63]

6.1.2 Value Dependency for Data Movement Energy

For the GPU's streaming multi-processors (i.e., SMs) to perform any computation, it first needs to fetch data from the off-chip memory, through the on-chip interconnect. This movement of data across the memory hierarchies consumes a substantial proportion of the total GPU energy [1, 6]. Recent studies have shown that the data movement energy is not only related to the quantity of data movement, but is also dependent on the values and order of the data moved [30, 7]. Primarily, it depends on two factors: the number of ones in the moved data (i.e., Hamming weight) and the number of bit toggling (i.e., Hamming distance) between moved data.

Number of Ones. The number of ones or Hamming weight in the binary data is simply the amount of bits that have value 1 in the data. For example, for an 8-bit variable A with binary representation 00001111, the number of ones or Hamming weight is 4. Prior work [30] have shown that the number of ones can significantly affect the memory read and write energy. Also, it slightly affects interconnect transfer energy. Therefore, the value of the moved data can affect data movement energy.

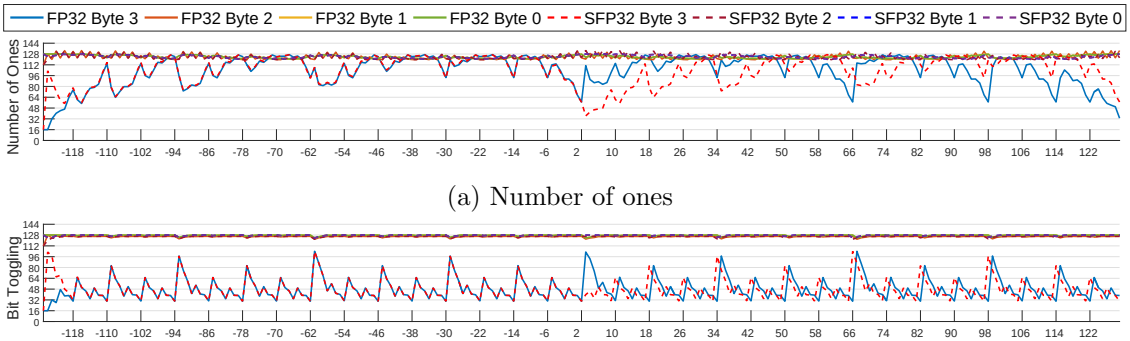
Number of Bit Toggling. The number of bit toggling or Hamming distance between binary data is the number of bit positions that have different values between consecutively transferred data. For example, let us assume that we are fetching data on an 8-bit wide data bus and we need three 8-bit variables A, B, and C with binary representations 00000000, 11111111, and 00000000 respectively. If we fetch them in order A, B, and C, the total bit toggling will be 16. This is because all eight bit positions will toggle from 0 to 1, and then toggle back to 0 again. However, if we fetch them in order A, C, and B,

the total bit toggling will be reduced to 8. This is because all eight bit positions will not toggle for the first two variables, A and C, but will only toggle from 0 to 1 when fetching the last variable, B. As shown in prior work [7], the number of bit toggling has a signficant impact on the interconnect transfer energy. Also, it has a small impact on the memory read and write energy. In this way, both the value and the order of the moved data can affect the data movement energy.

Data Bus Inversion. One of the effective ways to reduce either number of ones or bit toggling is to apply DBI (i.e., Data Bus Inversion) [40] to the memory system. DBI simply checks the number of ones or bit toggling of the current flit (i.e., the unit by which the data is transferred) to decide if all of its bits should be inverted or not. For example, DBI can be used to reduce the number of ones for memory reads. If more than half of the bits in the current flit are 1s, DBI can invert all bits of it so that more than half of its bits will be 0s. Also, DBI can be used to reduce bit toggling for the interconnect by storing the history value of the last flit. If more than half of the bit positions in the current flit are different from that of the last flit, DBI can invert all bits of the current flit so that more than half of its bit positions will not toggle. Because of its effectiveness and simple design, DBI is adopted by many newly released architectures [81].

6.1.3 Evaluation Methodology and Metrics

We use a memory model as described in Chapter 2.3. Our evaluation is performed on a cycle-level GPU simulator – GPGPU-Sim [12] (Table 5.1). We modified the memory controller implementation and added support for DBI for our experiments. Energy-related measurements are collected using GPUWattch [64]. We refined the logic of memory and interconnect power calculation to reflect the effect of data values on energy. We also introduce several truncation-related terminology used in this paper. The truncation ratio is defined as the ratio between the truncated data length and the original data length. The truncation coverage is the percentage of data that is truncated. The truncation error is the relative error of the truncated data value compared to its original value.



(a) Number of ones

(b) Bit toggling

Figure 6.2: Number of ones and bit toggling for FP32 and SFP32.

6.2 Motivation and Analysis

The goal of this work is to reduce the data movement energy with novel data format and memory system design. In this section, we will start by analyzing the current floating-point format. We will discuss why value truncation techniques are suitable for floating-point formats and what are the difficulties of applying value truncation techniques. Next, we introduce our proposed symmetrical floating-point (SFP) format. We will compare it with the current floating-point format in terms of value-dependent energy cost and value truncation overhead. Furthermore, we will discuss the unique memory system design opportunities enabled by the SFP and their benefits.

6.2.1 Analysis of Floating-point Formats

In terms of reducing value-dependent energy costs, we found that value truncation can bring large benefits. As shown in Figure 6.2, we tested the average number of ones and bit toggling in 128-byte-long cache lines with uniformly distributed FP32 values. We generated 16384 cache lines[1] for each label on the x-axis, which indicates the maximum actual exponent value that the FP32 data can have (i.e., it defines the value range in terms of 2's power). Note that we use the flit size of 32 bytes. Also, results are summarized according to the FP32 byte indices as shown in the legend.

[1]Experiments with more samples show negligible differences.

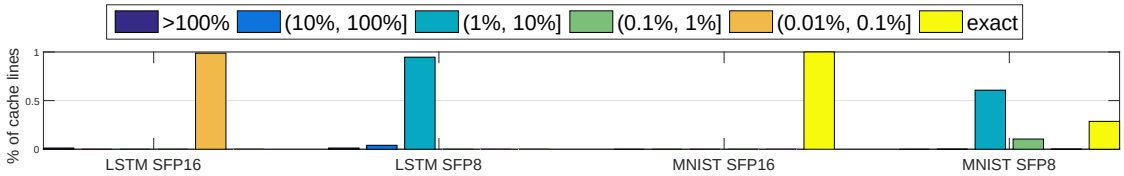


Figure 6.3: Average relative error distribution for LSTM and MNIST when using short formats.

As shown in the FP32 results of Figure 6.2a and Figure 6.2b, for both metrics, all bytes show results near 128 except for the byte 3 of FP32. Since we only have 256 bits for each byte index and have DBI enabled (Section 6.2.4), this shows the top level of data movement energy consumption. If we look at the FP32 format as shown in Figure 6.1, almost all the bits in byte 0, 1, 2 are bits from the significand and only byte 2 contains the lowest bit of exponent. For a certain range of data, these bits are most sensitive to the value changes of data. So these bits can have high chances of having the values of one and being toggled. On the other hand, for byte 3, except for the sign bit, all the exponent bits need a much larger change of the FP32 value than the significand bits in order to be toggled. Also, the chance of being one in exponent bits largely depend on the range of data and it can only be high for certain ranges. Hence, we can observe that in general, the number of ones is slightly lower in byte 3 than that of the other byte indices. And the bit toggling is much lower in byte 3 than that of the other byte indices. This means that less data movement energy is caused by byte 3.

However, similarly, the subtle changes of the FP32 value are captured by the lower bits of the significand. Meanwhile, the changes of higher bits of the significand, the exponent bits, and the sign bit represent much larger FP32 value changes. Therefore, this makes it very beneficial to reduce the length of the significand even with the loss of some information. Next, we will discuss in detail the techniques to achieve this purpose.

6.2.2 Value Truncation for Floating-point Formats

As we have discussed earlier, the significand bits of floating-point format can contribute to more data movement energy consumption but have less impact on the value of the data. If such bits contain the value of 0, then the data format can simply discard these bits without losing any information[2] when being converted to a shorter format. However, if such bits contain a value of 1, some information will be lost during the conversion. We call this action of discarding bits directly as value truncation.

However, despite the simplicity of value truncation, it cannot be used on the exponent bits of the floating-point format. As shown in Table 6.1, the offsets are different for format in different sizes. In order to be converted to a shorter format, the exponent will need to subtract its own offset and then add the new offset of the shorter format. Only after this, we are able to know if any information will be lost during the conversion by checking if the leading 1 of the result exceeds the length of the shorter exponent. And unlike the significand, any information loss in the exponent is undesirable due to its high impact on the value. Therefore, values which have smaller actual exponent ranges would be better targets to be converted into shorter formats to save energy. Fortunately, many data sets and applications heavily use this kind of data. For example, as shown in Figure 6.3, we show the average relative error distribution of the LSTM weight data in the Tango benchmark [52] and MNIST data set [62]. We show the result of when all their FP32 data is converted into two shorter formats, 16-bit-long SFP16 and 8-bit-long SFP8, which we will introduce shortly. We can see that for SFP16, almost all of the data shows very low error or no error. For SFP16, still, parts of the data show low or no error, which indicates that many of them fall into a good range for using shorter formats.

To be able to use shorter formats, existing techniques [11, 60, 61] would require to generate the converted data and copy them to the GPU memory before its usage. However, this approach has a few major drawbacks. First, as memory capacity is a major bottleneck

[2]We assume that data is zero-padded when converting to longer formats.

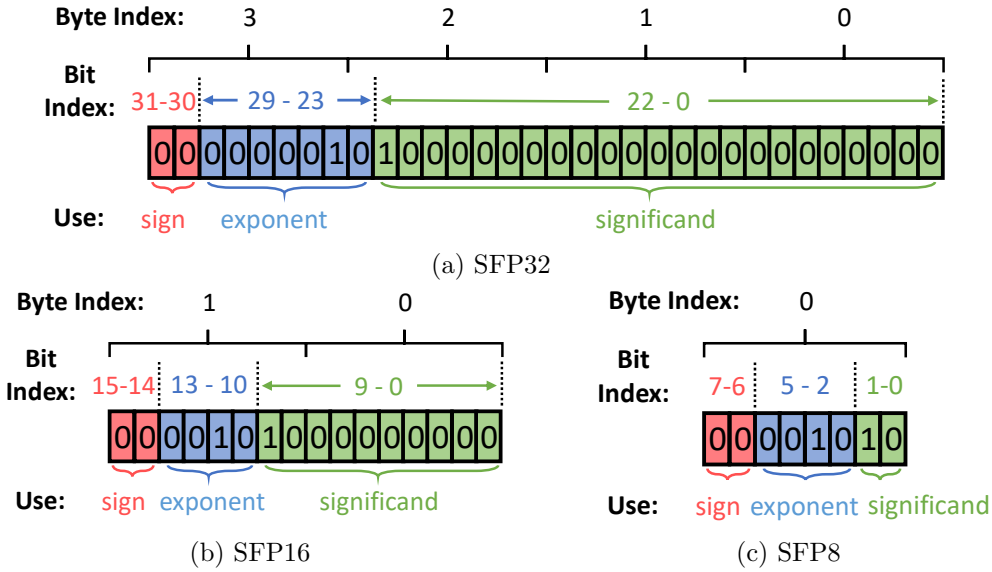


Figure 6.4: Layouts of the symmetrical floating-point (SFP) formats.

for current GPUs [10, 45, 71], this approach can make the situation worse, especially if multiple copies of different formats are used. Second, this approach is not dynamic, which means that the intermediate data generated during the execution does not benefit from shorter formats. Third, it depends on the programmer to provide means to control the error caused by the conversion. On the other hand, if we only store the original data in the GPU memory, it is almost prohibitively expensive to convert the data locally at the memory before sending it to the cores. To do the conversion, data must first be read from the memory, which produces data movement energy at the memory. This step alone would partially cancel the benefit of shorter formats. Moreover, the computation involved in the conversion could incur large energy costs, delays, and hardware overhead. To tackle these problems, we propose a new type of floating-point format, symmetrical floating-point (SFP). Next, we will introduce the SFP and the unique memory system design opportunities enabled by it.

$$Value = \begin{cases} (-1)^{sign} \times Infinity, & \text{if } exponent = Exp.Max \text{ and } E\text{-}sign = 0 \text{ and } significand = 0 \\ NaN \text{ (not a number)}, & \text{if } exponent = Exp.Max \text{ and } E\text{-}sign = 0 \text{ and } significand \neq 0 \\ 0, & \text{if } exponent = 0 \text{ and } E\text{-}sign = 1 \text{ and } significand = 0 \\ (-1)^{sign} \times 2^{1-Exp.Max} \times 0.significand, & \text{if } exponent = 0 \text{ and } E\text{-}sign = 0 \text{ and } significand \neq 0 \textbf{ (subnormal)} \\ (-1)^{sign} \times 2^{(-1)^{E\text{-}sign} \times exponent} \times 1.significand, & \text{otherwise } \textbf{(normal)} \end{cases} \tag{6.2}$$

Table 6.2: Configurations of symmetrical floating-point data formats.

Format	Sign	E-sign	Exponent	Exp. Max	Significand	Offset	Actual Exp. Range
SFP64	1 bit	1 bit	10 bits	1023	52 bits	NA	[-1022, 1023]
SFP32	1 bit	1 bit	7 bits	127	23 bits	NA	[-126, 127]
SFP16	1 bit	1 bit	4 bits	15	10 bits	NA	[-14, 15]
SFP8	1 bit	1 bit	4 bits	15	7 bits	NA	[-126, 127]

6.2.3 Symmetrical Floating-point (SFP) Format

As introduced in Section 6.2.4, one feature of the current floating-point formats is that the raw values of their exponent bits are not symmetrical around 0. This means that in order to get the actual exponent value, an offset must be deducted from the raw exponent value. Depending on the format specification, this offset is also different (Table 6.1). Due to this feature, its exponent bits cannot be simply truncated during a conversion, as discussed earlier. To this end, we propose the symmetrical floating-point (SFP) format as shown in Figure 6.4.

The key feature of SFP is that its actual exponent values are symmetrical around 0 so that the offset is not needed. This means that the raw value of any actual exponent value equals its own absolute value. The length of exponent bits is also one bit shorter than that of its corresponding FP format. Meanwhile, an E-sign (i.e., exponent sign) bit is added to the sign bit, to represent the sign of the exponent. For example, both the actual exponent value of 1 and -1 will have the same exponent bits 0000001, with E-sign 0 and 1 respectively. Despite the changes in the exponent, SFP can store the exact same information as FP formats with the same length. For example, an SFP data as shown in

Figure 6.4a stores the exact same content as an FP data as shown in Figure 6.1. Also, as shown in Table 6.2, we listed the corresponding SFP formats for FP64, FP32, and FP16. Compared to Table 6.1, we can see that although SFP has a different exponent design, its actual exponent ranges are not affected. To calculate the represented value of SFP, we can use equation 6.2 with all the given bit values. We can observe that this formula is almost identical to equation 6.1, except for the exponent calculation and simple E-sign condition checks. Therefore, the SFP requires little or no changes to the existing hardware for calculation. Also, it can be converted to FP formats efficiently.

Compared to the FP formats, SFP formats show several advantages. First, it can be easily converted into different SFP formats without any additional computation. For example, SFP32 as shown in Figure 6.4a can be easily converted into SFP16 (Figure 6.4b) and further into SFP8 (Figure 6.4c). We can observe that, since no offset is involved for the exponent, we can simply truncate the high bits of exponent and low bits of significand. Note that we can easily guarantee that no information is lost in the exponent by checking if the truncated exponent bits contain 1 or not. This leads to the high flexibility in the usage of the format. For example, even if SFP8 does not have a corresponding FP8 format supported, it can be converted into SFP32 by padding zeros whenever needed. But SFP8 can be used only during the data movement from the memory. Second, SFP favors value-dependent energy cost for values that have smaller actual exponent ranges, which are better targets to be converted into shorter formats to save energy. For example, as shown in Figure 6.2, the byte 3 results of SFP32 are lower than that of FP32 when the actual exponent values are between 2 and 32, between 2 and 10, respectively for the number of ones and bit toggling. This is due to the fact that small actual exponent values do require fewer exponent bits for SFP, but always need all exponent bits for FP. Third, SFP does not consume additional space in the memory to benefit from shorter formats. One copy of SFP data can be truncated into shorter formats on-demand locally at the memory without any overhead. Fourth, SFP can dynamically use intermediate shorter formats, as different SFP formats can switch flexibly. Finally, as we will show later, error bounding for SFP

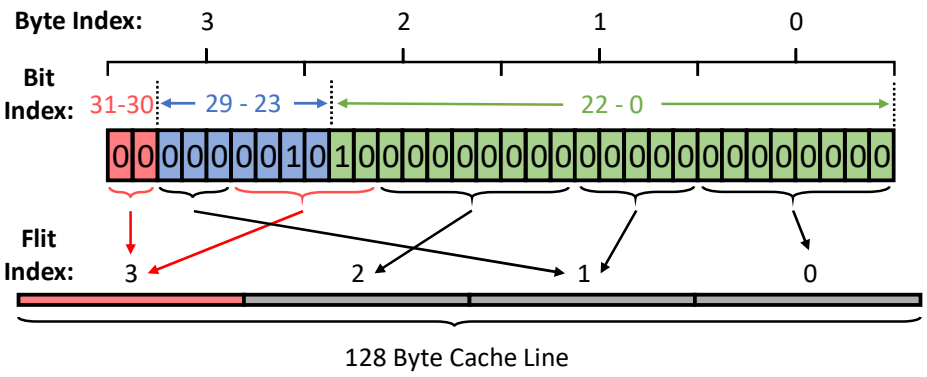


Figure 6.5: Flit mapping strategy enabled by SFP.

can be done efficiently without programmer effort. Due to these advantages, SFP enables new memory system design opportunities.

6.2.4 A Flexible Memory System

The flexible feature of SFP enables several unique memory system design opportunities. These novel designs do not incur large changes to the existing memory system but can bring large energy and performance benefits. Prior work [39] has discussed the usage of shorter formats at the cores. However, the memory system design enabled by SFP can provide benefits even without using these schemes at the cores.

First, a memory mapping strategy can be used together with SFP to eliminate the need to fetch certain parts of the cache line out of memory bank. As shown in Figure 6.5, within an SFP32 data, all bits involved in the SFP8 when converted can be mapped into a single flit 3. Similarly, all bits involved in the SFP16 can be mapped into flit 3 and flit 2. Therefore, only the required flit needs to be fetched from the memory when using shorter formats, implicitly completing the conversion. Second, a comparator can be used to dynamically bound the error of the data. Due to the simplicity of the format conversion enabled by SFP, a simple comparator will be sufficient to provide a bound for truncation error, which has low latency and hardware overhead. Third, a tagging scheme can be used to mark the desired cache line for truncation. A single cache line can have

as low as 1 bit to indicate whether it can be a target for using shorter formats or not. Therefore, it is possible for us to store the metadata for all or part of the cache lines. This metadata can be used to indicate if a cache line have high number of ones, bit toggling, or truncation error, etc.. Fourth, minimal changes are required for the cache to support SFP with shorter length, therefore, significantly improving its capacity. Similar to the memory, the cache only requires one copy of the data in order to generate SFP formats of other lengths. Therefore, its capacity can be effectively 3 times larger if it stores values as SFP8.

In summary, these techniques enabled by SFP is able to help us design a more efficient memory system. In the future, we will further explore the possibilities of this novel memory system design, and provide the implementation details for Flexmem.

6.3 Conclusions

In this chapter, we presented initial results on our current work, Flexmem. We did a detailed analysis of floating-point format and discussed the reason why value truncation is useful for it. We then proposed a novel floating-point format, symmetrical floating-point (SFP). SFP has several advantages over the previous floating-point format design. Most importantly, it enables a novel memory system design, which significantly improves performance and energy-efficiency. We conclude that Flexmem can be used towards developing architectural support for flexible data precisions and is a useful tool to enhance the scalability of GPUs.

Chapter 7

Conclusion and Future Work

7.1 Summary of Dissertation Contributions

GPUs are designed to provide high compute throughput via high thread-level parallelism. For each generation of GPUs, the number of cores continuously grow, providing higher peak throughput. To support the continuous scaling of GPU cores, it is crucial to develop new generation of GPU memories that can achieve higher theoretical bandwidth within a limited power budget. Thus, we answer the aforementioned three questions (Chapter 1) by summarizing the contributions of this dissertation.

1. How can we efficiently and fairly mange the memory resources for multiple co-running applications in the GPU? With the continuous scaling of GPUs, GPU multi-programming has become an inevitable trend to improve the occupancy of GPUs. However, one major challenge of this is the difficulty of avoiding the contentions between multiple applications on the shared resources. We analyzed the problem of shared resource contention between multiple concurrently executing GPGPU applications and showed that there is an ample scope for TLP management techniques for improving system throughput and fairness in GPUs. Therefore, in the first research, we propose pattern-based searching (PBS) that cuts down a significant amount of the searching overhead of the optimal TLP combinations. To facilitate this, we propose a new metric, *effective bandwidth (EB)*,

which more accurately measures the effective shared resource usage for each application by considering its private and shared cache miss rates and memory bandwidth consumption. This research shows that the pattern-based searching for TLP can significantly improve the system throughput and fairness in GPUs compared to previously proposed state-of-the-art mechanisms. We also believe that the presented analysis and the insights can be extended to other systems (e.g., chip-multiprocessors, systems-on-chip with accelerator IPs, server processors) where contention in shared caches and memory resources are performance-critical factors.

2. How can we reduce the data movement in the memory hierarchy and improve the throughput of the GPU? As the number of GPU cores continue to grow, increasing data movements will be imposed on the GPU memory. In order to alleviate such burdens, value approximation techniques recently have received attention. In the second research, we propose Address-Stride Assisted Approximate Value Predictor (ASAP), which utilizes the address stride and value stride correlation in many realistic inputs and predicts the values only if it detects strides in their corresponding addresses. ASAP is designed to identify the value stride pattern(s) in a highly multi-threaded environment where thousands of memory requests can be on-the-fly and their access order is highly dependent on GPU-specific features such as warp scheduling and coalescing. This research shows that ASAP can significantly improve value prediction accuracy even at a high value of prediction coverage, leading to significant performance and data movement benefits. Compared to prior works, ASAP also incurs lower hardware overhead. We believe that this work can open up interesting research avenues that consider other readily available information locally at the core (e.g., address stride information) to improve the accuracy of value prediction.

3. How can we improve the energy efficiency of the GPU memory? The high energy consumption of GPU memory is another reason that limits the growth of its peak bandwidth, due to the limited memory power budget. We observe that several GPGPU applications suffer from poor row buffer reuse, which contributes to a significant proportion

of GPU memory energy consumption. In the third research, we propose the lazy memory scheduler which leverages the latency and error tolerance of GPGPU applications. The lazy memory scheduler works in two modes: delayed or approximate. In the delayed mode, it carefully delays the scheduling of memory requests to allow more of them to accumulate at the memory pending queue. Such a mechanism increases the visibility of the memory scheduler thereby improving the chances of finding more requests that can be served by reusing the data in the row buffer. In the approximate mode, it carefully identifies a small fraction of requests with low row buffer locality and does not issue them to the DRAM banks. Instead, a simple but effective value predictor can be used to approximate the values for such requests. This research shows that both these modes are effective in improving the row buffer locality while reserving the system throughput. Moreover, they can work synergistically and improve the effectiveness of each other when employed together. We hope that this work can open up new research directions that consider the interactions between scheduling, error resilience, and latency tolerance techniques at different levels of the memory hierarchy.

7.2 Future Work

Architectural Support for Flexible Data Precisions. We propose a new memory architecture which supports low overhead data precision conversion locally at the memory. This work targets to dynamically managing the precision of data in the GPU memory such that the system throughput and memory energy efficiency can be improved with no or limited application quality loss.

Bibliography

[1] 1st Workshop on Hardware/Software Techniques for Minimizing Data Movement. *http://insight-archlab.github.io/minmove.html*.

[2] NVIDIA GTX 780-Ti. *http://www.nvidia.com/gtx-700-graphics-cards/gtx-780ti/*.

[3] The Green500 List - June 2015. *http://www.green500.org/lists/green201506*.

[4] Top500 Supercomputer Sites - June 2015. *http://www.top500.org/lists/2015/06/*.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[6] V. Adhinarayanan, I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W. c. Feng. Measuring and modeling on-chip interconnect power on real hardware. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, Sept 2016.

[7] V. Adhinarayanan, I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W. Feng. Measuring and modeling on-chip interconnect power on real hardware. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, 2016.

[8] J.T. Adriaens, K. Compton, Nam Sung Kim, and M.J. Schulte. The Case for GPGPU Spatial Multitasking. In *HPCA*, 2012.

[9] Advanced Micro Devices Inc. AMD Graphics Cores Next (GCN) Architecture, 2012.

[10] Neha Agarwal, David Nellans, Mark Stephenson, Mike OConnor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. *SIGARCH Comput. Archit. News*, 43(1):607618, March 2015.

[11] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526 – 2533, 2009. 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures.

[12] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[13] Benjamin Block, Peter Virnau, and Tobias Preis. Multi-gpu accelerated multi-spin monte carlo simulations of the 2d ising model. *Computer Physics Communications*, 181(9):1549 – 1556, 2010.

[14] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. *ACM SIGPLAN Notices*, 48(10):33–52, 2013.

[15] Kevin K Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*, 2016.

[16] Niladrish Chatterjee, Mike OConnor, Donghyuk Lee, Daniel R Johnson, Stephen W Keckler, Minsoo Rhu, and William J Dally. Architecting an Energy-Efficient DRAM System for GPUs. In *HPCA*, 2017.

[17] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[18] A. E. Cohen and K. K. Parhi. Gpu accelerated elliptic curve cryptography in gf(2m). In *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, pages 57–60, Aug 2010.

[19] Vinodh Cuppu and Bruce Jacob. Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-system Performance? In *ISCA*, 2001.

[20] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *GPGPU*, 2010.

[21] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. Application-to-core Mapping Policies to Reduce Memory Interference in Multi-core Systems. In *PACT*, 2012.

[22] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. Application-aware Prioritization Mechanisms for on-chip Networks. In *MICRO*, 2009.

[23] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. Aergia: Exploiting Packet Latency Slack in on-chip Networks. In *ISCA*, 2010.

[24] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *ASPLOS*, 2010.

[25] Richard J Eickemeyer and Stamatis Vassiliadis. A Load-Instruction Unit for Pipelined Processors. *IBM Journal of Research and Development*, 37(4):547–564, 1993.

[26] Stijn Eyerman and Lieven Eeckhout. The Benefit of SMT in the Multi-core Era: Flexibility Towards Degrees of Thread-level Parallelism. In *ASPLOS*, 2014.

[27] J. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse. Gpucv: A framework for image processing acceleration with graphics processors. In *2006 IEEE International Conference on Multimedia and Expo*, pages 585–588, July 2006.

[28] Freddy Gabbay. Speculative Execution Based on Value Prediction. Technical Report 1080, Technion - Israel Institute of Technology, 1996.

[29] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján. HAPPY: Hybrid Address-based Page Policy in DRAMs. In *Proceedings of the Second International Symposium on Memory Systems*, 2016.

[30] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike OConnor, and Onur Mutlu.

What your dram power models are not telling you: Lessons from a detailed experimental study. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), December 2018.

[31] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. Gpu-acceleration of block ciphers in the openssl cryptographic library. In *Information Security*, Dieter Gollmann and Felix C. Freiling, editors, pages 338–353, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] Nilanjan Goswami, Bingyi Cao, and Tao Li. Power-performance Co-optimization of Throughput Core Architecture Using Resistive Memory. In *HPCA*, 2013.

[33] GPGPU-Sim v3.2.1. Address mapping.

[34] GPGPU-Sim v3.2.1. GTX 480 Configuration.

[35] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained Resource Sharing for Concurrent GPGPU Kernels. In *HotPar*, 2012.

[36] Nagendra Gulur, Mahesh Mehendale, Raman Manikantan, and Ramaswamy Govindarajan. ANATOMY: An Analytical Model of Memory System Performance. In *SIGMETRICS*, 2014.

[37] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. Many-Core vs. Many-Thread Machines: Stay Away from the Valley. *CAL*, January 2009.

[38] Wim Heirman, Trevor E Carlson, Kenzo Van Craeynest, Ibrahim Hur, Aamer Jaleel, and Lieven Eeckhout. Undersubscribed Threading on Clustered Cache Architectures. In *HPCA*, 2014.

[39] N. Ho and W. Wong. Exploiting half precision arithmetic in nvidia gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[40] T. M. Hollis. Data bus inversion in high-speed memory applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(4):300–304, 2009.

[41] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *ISCA*, 2009.

[42] Sunpyo Hong and Hyesoon Kim. An Integrated GPU Power and Performance Model. In *ISCA*, 2010.

[43] Hynix. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.

[44] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *HPCA*, 2014.

[45] G. Jin, T. Endo, and S. Matsuoka. A parallel optimization method for stencil computation on the domain that is bigger than memory capacity of gpus. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013.

[46] Adwait Jog, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications. In *GPGPU*, 2014.

[47] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Nilardish Chatterjee, Steve Keckler, Mahmut T. Kandemir, and Chita R. Das. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*, 2015.

[48] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Nilardish Chatterjee, Steve Keckler, Mahmut T. Kandemir, and Chita R. Das. MAFIA - Multiple Application Framework in GPU Architectures. *URL: https://github.com/adwaitjog/mafia*, 2015.

[49] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.

[50] Adwait Jog, Onur Kayiran, Nachiappan C. Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.

[51] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T. Kandemir, Onur Mutlu, Ravi Iyer, and Chita R. Das. Exploiting Core Criticality for Enhanced Performance in GPUs. In *SIGMETRICS*, 2016.

[52] A. Karki, C. Palangotu Keshava, S. Mysore Shivakumar, J. Skow, G. Madhukeshwar Hegde, and H. Jeon. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 137–138, 2019.

[53] I. Karlin, A. Bhatele, J. Keasler, B.L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C.H. Still. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *IPDPS*, 2013.

[54] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.

[55] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. μC-States: Fine-grained GPU Datapath Power Management. In *PACT*, 2016.

[56] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. Managing GPU Concurrency in Heterogeneous Architectures. In *MICRO*, 2014.

[57] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, 2011.

[58] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An Online Quality Management System for Approximate Computing. In *ISCA*, 2015.

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

[60] Pradeep V Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. Ampt-ga: Automatic mixed precision floating point tuning for gpu applications. In *Proceedings of the ACM International Conference on Supercomputing*, ICS 19, page 160170, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, and Saurabh Bagchi. Gpumixer: Performance-driven floating-point tuning for gpu scientific applications. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan, editors, pages 227–246, Cham, 2019. Springer International Publishing.

[62] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[63] Janghaeng Lee, Mehrzad Samadi, and Scott A. Mahlke. Orchestrating Multiple Data-Parallel Kernels on Multiple Devices. In *PACT*, 2015.

[64] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.

[65] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. SFU-Driven Transparent Approximation Acceleration on GPUs. In *ICS*, 2016.

[66] Xiuhong Li and Yun Liang. Efficient Kernel Management on GPUs. In *DATE*, 2016.

[67] Mikko H Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *MICRO*, 1996.

[68] Joe Macri. Amd's next generation gpu and high bandwidth memory architecture: Fury. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–26. IEEE, 2015.

[69] Divya Mahajan, Kartik Ramkrishnan, Rudra Jariwala, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Anandhavel Nagendrakumar, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. Axilog: Abstractions for Approximate Hardware Design and Reuse. In *MICRO*, 2015.

[70] S. A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 65–68, Nov 2007.

[71] L. Mattes and S. Kofuji. Overcoming the gpu memory limitation on fdtd through the use of overlapping subgrids. In *2010 International Conference on Microwave and Millimeter Wave Technology*, pages 1536–1539, 2010.

[72] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *ASPLOS*, 2014.

[73] Joshua San Miguel, Mario Badr, and Enright Natalie Jerger. Load Value Approximation. In *MICRO*, 2014.

[74] Young-Suk Moon, Yongkee Kwon, Hong-Sik Kim, Dong-gun Kim, Hyung-dong Hayden Lee, and Kunwoo Park. The Compact Memory Scheduling Maximizing Row Buffer Locality. In *3rd JILP Workshop on Computer Architecture Competitions: Memory Scheduling Championship*, 2012.

[75] Tarun Nakra, Rajiv Gupta, and Mary Lou Soffa. Global Context-Based Value Prediction. In *HPCA*, 1999.

[76] Chitra Natarajan, Bruce Christenson, and Fayé Briggs. A Study of Performance Impact of Memory Controller Features in Multi-processor Server Environment. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, 2004.

[77] Bin Nie, Devesh Tiwari, Saurabh Gupta, Evgenia Smirni, and James H Rogers. A Large-scale Study of Soft-errors on GPUs in the Field. In *HPCA*, 2016.

[78] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *MICRO*, 2018.

[79] NVIDIA. CUDA C/C++ SDK Code Samples. *http://developer.nvidia.com/cuda-cc-sdk-code-samples*, 2011.

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

[81] NVIDIA. NVIDIA Tesla V100 GPU Architecture Whitepaper. Technical report, 2018.

[82] Mike OConnor. Highlights of the high-bandwidth memory (hbm) standard. In *Memory Forum Workshop*, 2014.

[83] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems. In *MICRO*, 2017.

[84] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS*, 2013.

[85] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim. Design and performance evaluation of image processing algorithms on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, Jan 2011.

[86] Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *ASPLOS*, 2015.

[87] Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *ASPLOS*, 2017.

[88] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. Flexjava: Language Support for Safe and Modular Approximate Programming. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[89] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 347–348, Aug 2015.

[90] Arthur Perais and André Seznec. Practical Data Value Speculation for Future High-End Processors. In *HPCA*, 2014.

[91] Perais, Arthur and Seznec, André. EOLE: Paving the Way for an Effective Implementation of Value Prediction. In *ISCA*, 2014.

[92] Perais, Arthur and Seznec, André. BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction. In *HPCA*, 2015.

[93] Louis-Noël Pouchet. Polybench: the polyhedral benchmark suite. *http://www.cs.ucla.edu/∼pouchet/software/polybench/*, 2012.

[94] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA*, 2006.

[95] Moinuddin K Qureshi and Yale N Patt. Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*, 2006.

[96] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *MICRO*, 2013.

[97] Scott Rixner. Memory Controller Optimizations for Web Servers. In *MICRO*, 2004.

[98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[99] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.

[100] Diego Rossinelli, Michael Bergdorf, Georges-Henri Cottet, and Petros Koumoutsakos. Gpu accelerated simulations of bluff body flows using vortex particle methods. *Journal of Computational Physics*, 229(9):3316 – 3333, 2010.

[101] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-Tuning Approximation for Graphics Engines. In *MICRO*, 2013.

[102] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. *ACM SIGPLAN Notices*, 46(6):164–174, 2011.

[103] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. The Bunker Cache for Spatio-Value Approximation. In *MICRO*, 2016.

[104] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelganger: A Cache for Approximate Computing. In *MICRO*, 2015.

[105] Edans Flavius O. Sandes and Alba Cristina M.A. de Melo. Cudalign: Using gpu to accelerate the comparison of megabase genomic sequences. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 137–146, New York, NY, USA, 2010. ACM.

[106] Yiannakis Sazeides and James E Smith. Implementations of Context Based Value Predictors. Technical report, Technical Report ECE-97-8, University of Wisconsin-Madison, 1997.

[107] Sazeides, Yiannakis and Smith, James E. The Predictability of Data Values. In *MICRO*, 1997.

[108] Sazeides, Yiannakis and Smith, James E. Modeling Program Predictability. In *ISCA*, 1998.

[109] Bertil Schmidt and Andreas Hildebrandt. Next-generation sequencing: big data meets high performance computing. *Drug Discovery Today*, 22(4):712 – 717, 2017.

[110] Hynix Semiconductor. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.

[111] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up GPU Warps by Reducing Memory Pitstops. In *HPCA*, 2015.

[112] Ankit Sethia and Scott Mahlke. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *MICRO*, 2014.

[113] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS*, 2000.

[114] D. Steinkraus, I. Buck, and P. Y. Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pages 1115–1120 Vol. 2, Aug 2005.

[115] 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. Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, March 2012.

[116] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: Increasing DRAM Efficiency with Locality-aware Data Placement. In *ASPLOS*, 2010.

[117] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense point trajectories by gpu-accelerated large displacement optical flow. In *Computer Vision – ECCV 2010*, Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, pages 438–451, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[118] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.

[119] Xulong Tang, Ashutosh Pattnaik, Huaipan Jiang, Onur Kayiran, Adwait Jog, Sreepathi Pai, Mohamed Ibrahim, Mahmut T Kandemir, and Chita R Das. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *HPCA*, 2017.

[120] Renju Thomas and Manoj Franklin. Using Dataflow Based Context for Accurate Value Prediction. In *PACT*, 2001.

[121] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding GPU Errors on Large-scale HPC Systems and the Implications for System Design and Operation. In *HPCA*, 2015.

[122] Jelena Trajkovic, Alexander V Veidenbaum, and Arun Kejariwal. Improving SDRAM Access Energy Efficiency for Low-power Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):24, 2008.

[123] Cole Trapnell and Michael C. Schatz. Optimizing data intensive gpgpu computations for dna sequence alignment. *Parallel Computing*, 35(8):429 – 440, 2009.

[124] A. Tumeo and O. Villa. Accelerating dna analysis applications on gpu clusters. In *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pages 71–76, June 2010.

[125] Yash Ukidave, Xiangyu Li, and David R. Kaeli. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *IPDPS*, 2016.

[126] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1131–1142, New York, NY, USA, 2014. ACM.

[127] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. Approxilyzer: Towards a Systematic Framework for Instruction-Level Approximate Computing and Its Application to Hardware Resiliency. In *MICRO*, 2016.

[128] Thiruvengadam Vijayaraghavany, Yasuko Eckert, Gabriel H Loh, Michael J Schulte, Mike Ignatowski, Bradford M Beckmann, William C Brantley, Joseph L Greathouse, Wei Huang, Arun Karunanithi, et al. Design and Analysis of an APU for Exascale Computing. In *HPCA*, 2017.

[129] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Onur Mutlu, Chita Das, Mahmut T. Kandemir, Todd Mowry, and Rachata Ausavarungnirun. Enabling Efficient Data Compression in GPUs. In *ISCA*, 2015.

[130] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *HPCA*, 2018.

[131] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs. In *ISCA*, 2015.

[132] Lingyuan Wang, Miaoqing Huang, and T. El-Ghazawi. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *HPCS*, 2011.

[133] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-grained Sharing. In *HPCA*, 2016.

[134] Daniel Wong, Nam Sung Kim, and Murali Annavaram. Approximating Warps with Intra-Warp Operand Value Similarity. In *HPCA*, 2016.

[135] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *ISCA*, 2016.

[136] Juekuan Yang, Yujuan Wang, and Yunfei Chen. Gpu accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2):799 – 804, 2007.

[137] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. Axbench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test*, 34(2):60–68, 2017.

[138] Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, et al. Axilog: Language support for approximate hardware design. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 812–817. EDA Consortium, 2015.

[139] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. RFVP: Rollback-free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):62, 2016.

[140] Hangin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. Row Buffer Locality-aware Data Placement in Hybrid Memories. In *ICCD*, 2011.

[141] Seyed Majid Zahedi and Benjamin C Lee. REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors. In *ASPLOS*, 2014.

[142] Tao Zhang, Ke Chen, Cong Xu, Guangyu Sun, Tao Wang, and Yuan Xie. Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation. In *ISCA*, 2014.

[143] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *MICRO*, 2000.

[144] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. Breaking Address Mapping Symmetry at Multi-levels of Memory Hierarchy to Reduce DRAM Row-buffer Conflicts. *The Journal of Instruction-Level Parallelism*, 3:29–63, 2001.

[145] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive Cache and Concurrency Allocation on GPGPUs. *CAL*, 14(2):90–93, 2015.

[146] William K. Zuravleff and Timothy Robinson. Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order. (U.S. Patent Number 5,630,096), September 1997.

VITA

Haonan Wang

Haonan Wang is a Ph.D. Candidate in the Department of Computer Science at The College of William and Mary. He received his BS degree of Computer Science at East China University of Science and Technology. He received his MS degree of Computer Science at The College of William and Mary. His Ph.D. advisor is Adwait Jog. Haonan's main research direction is GPU Architecture and he is interested in architecture related topics such as memory system architecture, approximate computing, hardware scheduling, machine learning and GPU security. His research has been published in major computer architecture conferences (HPCA, DSN, ICS). He has received travel awards from major computer architecture conferences (HPCA, MICRO, DSN). He worked as an intern at Pacific Northwest National Laboratory in the spring and summer of 2018.