ENABLING EFFICIENT RESOURCE UTILIZATION ON MULTITASKING THROUGHPUT PROCESSORS

by

Jason Jong Kyu Park

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2016

Doctoral Committee:

Professor Scott Mahlke, Chair
Assistant Professor Ronald G. Dreslinski, Jr.
Assistant Professor Yongjun Park, Hongik University
Professor Kevin P. Pipe
Associate Professor Thomas F. Wenisch

© Jason Jong Kyu Park 2016
All Rights Reserved

To my family

ACKNOWLEDGEMENTS

My sincere gratitude goes to my advisor, Professor Scott Mahlke for his constant encouragement and support during my Ph.D. He was one of the best mentors I had in my entire life, and I would suggest anyone to do Ph.D. with him. I also would like to thank Prof. Ronald G. Dreslinski, Prof. Yongjun Park, Prof. Kevin Pipe, and Prof. Thomas F. Wenisch for their invaluable comments and suggestions that helped me to improve and polish this dissertation.

My adventure for graduate research would have been so lonely without Compilers Creating Custom Processors (CCCP) members. Yongjun Park collaborated in all the projects, and provided significant help in making my thesis more concrete. Janghaeng Lee and I discussed a lot of research areas spanning across computer architecture and compilers. Fellow GPU guys, Ankit Sethia, Anoushe Jamshidi, and John Kloosterman, shared frustration and joyfulness of doing GPU research. My everyday life in the office was fun thanks to other current members and alumni of CCCP: Shantanu Gupta, Hyunchul Park, Hyoun Kyu Cho, Gaurav Chadha, Daya S. Khudia, Andrew Lukefahr, Shruti Padmanabha, Babak Zamirai, Jiecao Yu, Sunghyun Park, Shikai Li, and Jonathan Bailey. They also broadened my perspective with their own research. I also would like to thank other CELAB (formerly ACAL) members including Neha Agarwal, Shaizeen Aga, and Doowon Lee for discussing

research interests.

My graduate student life in Ann Arbor would have been miserable without my Korean friends. Janghaeng Lee, and Eugene Kim were involved in all the sports activities, and all the dinner I enjoyed in Ann Arbor. Won Choi and other fellow MESA friends, too many to name them all, were my drinking buddies for the first two years. My later years were shared with friends from playing soccer, tennis, rock climbing, and badminton.

My lifelong friends since high school were always behind me when I needed them. The past 15 years, and my refreshing visits to Korea during Ph.D. would have been dull without them. Although they would not understand nor want to understand anything from my dissertation, they deserve a paragraph for my thanks.

Last but not least, my utmost gratitude goes to my family, parents, my sister, my grandparents, and my wife. My grandparents continously showed me the importance of consistency and motivation. My father, Pyongwan Park, always supported my decisions, and my mother, Sungwon Lee, helped me shape my own path by letting me study freely. I also would like to thank other family members including my sister, Shin Young Park, for their unconditional love and support. Finally, I appreciate the love, comfort, and encouragement from my beloved wife, Jungim Min.

TABLE OF CONTENTS

DEDICATION . . . ii

ACKNOWLEDGEMENTS . . . iii

LIST OF FIGURES . . . viii

LIST OF TABLES . . . xii

ABSTRACT . . . xiii

CHAPTER

I. Introduction . . . 1

1.1 Challenges . . . 4
1.2 Contributions . . . 6
1.2.1 *ELF* . . . 6
1.2.2 *Chimera* . . . 7
1.2.3 *GPU Maestro* . . . 7

II. Background . . . 9

2.1 Terminology . . . 9
2.2 GPU Architecture and Execution Model . . . 10
2.3 Multitasking GPUs . . . 11

III. ELF: Maximizing Memory-Level Parallelism for GPUs with Coordinated Warp and Fetch Scheduling . . . 15

3.1 Introduction . . . 15
3.2 Motivation . . . 18
3.2.1 Maximizing Memory-level Parallelism . . . 18
3.2.2 Memory Conflicts and Fetch Stalls . . . 20
3.3 Architecture . . . 21

3.3.1 Finding Program Points 22
3.3.2 Priority Calculator 25
3.3.3 Fetch Scheduling in *ELF* 26
3.3.4 *ELF* with Cache Access Re-execution 27
3.3.5 *ELF* with Instruction Prefetch 29
3.4 Results 30
3.4.1 *ELF* Performance 32
3.4.2 Hardware Overhead 36
3.4.3 Comparison to Prior Works 37
3.5 Related Work to *ELF* 39
3.5.1 Memory-Level Parallelism 39
3.5.2 GPU Scheduling 40
3.6 *ELF* Conclusions 42

IV. Chimera: Collaborative Preemption for Multitasking on a Shared GPU 44

4.1 Introduction 44
4.2 Motivation 47
4.2.1 Spatial Multitasking 48
4.2.2 Prior Preemption Techniques 48
4.2.3 SM Flushing 50
4.2.4 Tradeoff 50
4.2.5 Collaborative Preemption 52
4.3 Architecture 53
4.3.1 GPU Scheduler with Preemptive Multitasking 55
4.3.2 Cost Estimation 56
4.3.3 Preemption Selection 58
4.3.4 SM Flushing 59
4.4 Results 61
4.4.1 Periodic Task with Deadline 63
4.4.2 Impact of Preemption Latency Constraint 65
4.4.3 Relaxed Idempotence Condition in SM Flushing 66
4.4.4 Case Study 67
4.5 Related Work to *Chimera* 71
4.6 *Chimera* Conclusions 73

V. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs 75

5.1 Introduction 75
5.2 Background 79
5.2.1 Multikernel Metrics 79
5.3 Motivation and Challenges 80
5.3.1 Spatial vs. Simultaneous Multikernel 80
5.3.2 Multitasking GPU Performance 82

5.3.3 Interference and Dynamism 83
5.3.4 SMK Challenges 85
5.3.5 Challenge 3: Starvation 88
5.4 GPU Maestro Design 89
5.4.1 Dynamic Resource Partitioning 90
5.4.2 2-Way Resource Allocation 93
5.4.3 Kernel-aware Warp Scheduling 95
5.5 Results 95
5.5.1 Resource Partitioning Performance 97
5.5.2 Kernel-aware Scheduling Performance 99
5.5.3 Repartitioning Analysis 101
5.5.4 Comparison to Prior Works 103
5.6 Related Work to *GPU Maestro* 105
5.6.1 Simultaneous Multithreading 105
5.6.2 GPU Multitasking 105
5.7 *GPU Maestro* Conclusions 107

VI. Conclusion 108

BIBLIOGRAPHY 112

LIST OF FIGURES

Figure

1.1	Comparison of CPU and GPU peak single precision floating point performance [19].	2
1.2	Comparison of CPU and GPU energy efficiency.	3
2.1	GPU architecture and execution model.	10
2.2	A more detailed illustration of an SM. W[i] denotes a warp.	11
2.3	GPU architecture with multitasking support.	12
3.1	Execution timeline of (a) greedy-then-oldest (GTO) warp scheduling, and (b) ELF warp scheduling for the given kernel program, where warps are at different execution points.	18
3.2	Distribution of stalls in the GTO scheduler. Memory conflict stalls occur when memory resources are saturated. Memory dependency stalls occur when instructions are waiting for the memory requests to come back. Fetch stalls happen when warps are waiting for instruction fetch. Other stalls include conflict and dependency stalls in other functional units.	19
3.3	Overall architecture for *ELF*. There are two components in *ELF*: a priority calculator, and a warp priority table (WPT). The priority calculator calculates the priority of a warp using the program points generated by the compiler. The warp priority table stores the results from the priority calculator, and directs the warp scheduler to issue accordingly.	21
3.4	Extending LSU with NewCAR. Structurally, NewCAR is almost identical to CAR [75]. The key difference is more relaxed conditions on when and how the NewCAR queue is controlled.	29
3.5	Allowed reordering of memory requests when a warp has (a) a load, and (b) a store waiting in the NewCAR queue. W[0] denotes an example warp that has a memory request in the NewCAR queue, and W[N] denotes any other warp. Memory requests from other warps can always bypass the memory request from the example warp. Loads from the example warp can bypass the loads from itself, but not the stores. Stores from the example warp cannot bypass any loads or stores from itself.	29
3.6	Performance improvement of NewCAR, instruction prefetch, *ELF*, and *ELF*++ over the baseline GTO. *ELF*++ combines *ELF* with NewCAR and instruction prefetch.	33

3.7 Reduction of priority recalculation in *ELF* compared to the naive priority recalculation. . . . 33
3.8 Sensitivity of *ELF*++ to (a) the number of available program points, and (b) the threshold for instruction prefetch. A geometric mean of IPC improvement for all the benchmarks is presented. . . . 35
3.9 Comparison of *ELF*++ with prior works. GTO is the baseline. . . . 36
3.10 Comparing *ELF*++ to prior works when different cache configurations are used. GTO is the baseline. . . . 38
4.1 Estimated preemption latency for each preemption technique. For draining, a uniform random distribution on the preemption point across thread block execution is assumed. For flushing, zero preemption latency is assumed. . . . 48
4.2 Estimated throughput overhead for each preemption technique when thread blocks running on an SM are assumed to be in sync. For flushing, a uniform random distribution on the preemption point across thread block execution is assumed. . . . 48
4.3 Theoretical cost of each preemption technique when preempting a thread block at a given amount of execution progress. Context switching has constant cost across the execution, draining has lower cost as a thread block is near the end of execution, and flushing has lower cost as a thread block is closer to the beginning of execution. . . . 52
4.4 GPU scheduler with preemptive multitasking. The scheduler is two-level: the kernel scheduler assigns SMs to each kernel that may involve preemption decisions, and the thread block scheduler executes the decision by dispatching or preempting thread blocks from each SM. SMs can feedback the schedulers when an event that can change the scheduling decision occurs. . . . 54
4.5 The percentage of preemptions that violate the deadline of a periodic, real-time task when GPGPU benchmarks are run together. The preemption latency constraint is 15 μs. . . . 63
4.6 Throughput overhead of each preemption technique when GPGPU benchmarks are run with a periodic, real-time task. The preemption latency constraint is 15 μs. Effective throughput is used to avoid giving unfair advantage to the preemption techniques that frequently miss the deadline. 63
4.7 Impact of varying preemption latency constraint on (a) the percentage of deadline violations, (b) throughput overhead, and (c) distribution of each preemption technique used in *Chimera*. . . . 65
4.8 The percentage of preemptions that violate a 15μs preemption latency constraint when SM flushing uses strict or relaxed idempotence condition. 67
4.9 ANTT improvement over the non-preemptive FCFS when LUD is concurrently executed with another benchmark. . . . 68
4.10 STP improvement over the non-preemptive FCFS when LUD is concurrently executed with another benchmark. . . . 68

5.1 Normalized STP of SMK when fixed number of threads are assigned to each kernel within an SM for representative examples. STP of spatial multitasking is used as 1. (a) Even SMK is as good as the best performing SMK, (b) large performance gap exists between the best performing SMK and even SMK, and (c) spatial multitasking performs better than any SMK. 81

5.2 A trace of the instructions per cycle (IPC) within an SM using a 50k instruction window when running SRAD and BFS together with SMK (top), and when running them independently for the same interval (bottom). Circles indicate the same interval, where the IPC trace shows different behavior between running alone, and SMK. 84

5.3 A trace of STP improvement over non-shared execution for Oracle and Even thread block partition within an SM using 50k instruction window when running SRAD and BFS together. Indicated sub-intervals are when Oracle chooses different partition from Even. 85

5.4 An illustration of resource fragmentation problem for shared memory when (a) a kernel terminates, and (b) a kernel is preempted. 86

5.5 A timeline of register fragmentation when running FDTD/LUD on an SM with SMK. 86

5.6 An IPC trace within an SM using a 50k instruction window when running LC/BS. LC is launched before BS. Sub-intervals, where BS starves, are shown. 88

5.7 An architectural overview of *GPU Maestro*. PCntr refers to existing performance counters on an SM. 90

5.8 An illustration of dynamic resource partitioning process in *GPU Maestro* when two kernels are running on the GPU. *GPU Maestro* makes repartitioning decisions at the end of every epoch, which may not result in repartitioning if the previous preferred partition is the best. Follower SMs follows the repartitioning decision including spatial multitasking, which gives the best performance. Note that trial and follower SMs can change to minimize the preemption overhead from repartitioning. Dedicated SMs turn into follower SMs when a steady state is reached. 91

5.9 (a) The proposed 2-way allocation, where a kernel schedules thread blocks in the opposite direction, and (b) the fixed preemption priority order imposed by the 2-way allocation. 93

5.10 ANTT increase and STP improvement of Spatial, SMK, and *GPU Maestro* over non-shared execution. Spatial partitions resources evenly at the SM granularity. SMK partitions resources evenly within the SMs. A geometric mean of all combinations of co-running kernels from each category is shown. ANTT is a lower-is-better metric, and STP is a higher-is-better metric. 98

5.11 ANTT increase and STP improvement of various kernel-aware scheduling techniques on *GPU Maestro* over non-shared execution. A geometric mean of all combinations of co-running kernels from each category is shown. All the kernel-aware scheduling techniques are implemented on top of GTO warp scheduling. None does not do any kernel-aware scheduling. ANTT is a lower-is-better metric, and STP is a higher-is-better metric. 99
5.12 (Left) The percentage of repartitioning decisions to use spatial multitasking, SMK with even partitioning, and SMK with other non-even partitioning, and (Right) repartitioning overhead in *GPU Maestro*. An average of all combinations of co-running kernels from each category is shown. . . 101
5.13 Thread block repartitioning timeline for ST/TPACF and LBM/TPACF. . . 102
5.14 ANTT increase and STP improvement of Spatial, SMK-(P+W), WS, and *GPU Maestro* over non-shared execution. A geometric mean of all combinations of co-running kernels from each category is shown. ANTT is a lower-is-better metric, and STP is a higher-is-better metric. 103
5.15 STP improvement of Spatial, SMK-(P+W), WS, and *GPU Maestro* over non-shared execution for selected benchmark pairs. 104

LIST OF TABLES

Table

3.1 System configuration 30
3.2 Benchmark specifications to evaluate *ELF*. PP refers to the number of program points before the merge. 32
3.3 Hardware overhead per SM from additional structures. 36
4.1 System configuration 60
4.2 Benchmark Specification. IDPT denotes idempotence, and the percentage of idempotent region in a thread block execution for non-idempotent kernels is also shown. 61
5.1 Resource trends in an SM on GPUs. 80
5.2 System configuration. 96
5.3 Benchmark specification. 97

ABSTRACT

ENABLING EFFICIENT RESOURCE UTILIZATION ON MULTITASKING THROUGHPUT PROCESSORS

by

Jason Jong Kyu Park

Chair: Scott Mahlke

Graphics processing units (GPUs) are increasingly adopted in modern computer systems beyond their traditional role of processing graphics to accelerating data-parallel applications. The single instruction multiple thread (SIMT) programming model enabled programmers to easily offload data-parallel kernels to the GPUs, and achieve large performance improvements and energy efficiency. As a result, many supercomputers, cloud services, and data centers are utilizing GPUs as general-purpose throughput processors.

In these modern computer systems, achieving high resource utilization becomes important, especially in the shared environment like cloud services, and data centers. To enable efficient resource utilization of multitasking GPUs, new unique challenges like enabling an efficient preemptive multitasking and resolving shared resource contention have to be solved. Many traditional policies such as context switching or fine-grained multitasking can incur a large overhead if blindly applied. To address these challenges with minimal

overhead, a framework that exploits both static characteristics of the SIMT programming model and runtime properties of GPU execution is required.

This thesis proposes a framework with hardware/software extensions to enable efficient resource utilization on multitasking GPUs. The framework identifies characteristics coming from the SIMT programming model or determines the best policy based on multiple candidates either at compile time or using runtime software. The framework is adaptive and dynamic because it gathers runtime statistics using hardware performance counters to guide the decisions in the runtime software, and implements individual mechanisms in hardware with low overhead.

The framework consists of three components. The framework uses compiler hints to improve utilization by an average of 11.9% when running memory-intensive kernels alone, which can provide synergistic improvement when multitasking is enabled. The framework further improves the system throughput by an average of 12.2% with a collaborative preemption mechanism for efficient preemptive multitasking, which can handle both latency-sensitive and throughput-oriented applications. This framework implements a dynamic resource management scheme to maximize the utilization by exploiting both spatial multitasking, which partitions resources at the streaming multiprocessor (SM) granularity, and simultaneous multikernel, which partitions resources within the SM. Combining all the techniques in the framework, 45.0% better system throughput was achieved over single kernel execution.

CHAPTER I

Introduction

GPUs have been traditionally used to process graphics, which require large memory bandwidth and computing capability. With the introduction of the single instruction multiple thread (SIMT) programming model such as CUDA [48] and OpenCL [34], the computing capability of GPUs became available to data-parallel applications. In the SIMT programming model, a large number of threads are launched onto a GPU, where they perform the same control-flow and data-flow execution on different data. To reduce the overhead from synchronization, threads are further grouped into a thread block, which is the basic unit of scheduling and can be synchronized with an explicit barrier.

Recent GPUs show about 10x peak performance for single precision floating point operations, and 4x energy efficiency compared to recent CPUs, as shown in Figure 1.1, and Figure 1.2. As a result, GPUs are becoming general-purpose throughput processors, and prevalent from mobile devices to data centers. For example, many supercomputers in the TOP500 [2] and the Green500 [1] are already composed of GPUs due to their high performance and energy efficiency when running data-parallel applications. GPUs are also readily available in cloud computing services like Amazon Web Services [4]. In these

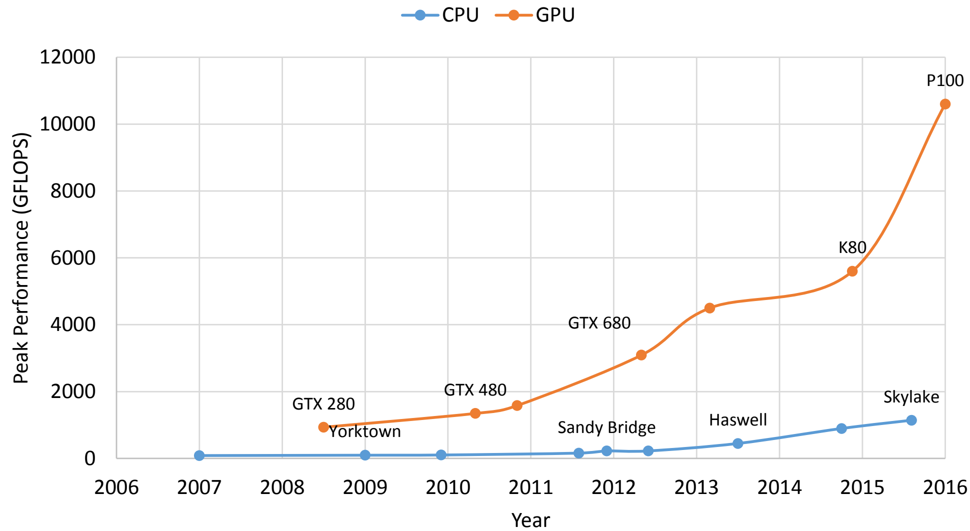


Figure 1.1: Comparison of CPU and GPU peak single precision floating point performance [19].

systems, GPUs are used beyond their traditional role of processing graphics to more diverse applications, including image/video processing, computer vision, machine learning, physical simulations, and big data analytics.

As GPUs are widely adopted in modern computer systems, achieving high resource utilization becomes an important problem to address. For example, new challenges like thread block scheduling [33, 40], and branch/memory divergence [18, 42, 36] were addressed to improve resource utilization for GPUs. Some of the research ideas are borrowed from CPUs, however, GPUs present new challenges due to both the differences in the programming model and the microarchitecture. For example, instruction issue [45, 68, 75], prefetching [39, 74], cache bypassing [8, 92], and dynamic voltage and frequency scaling [76] were modified to meet the GPUs' needs. Achieving high resource utilization for single kernel execution on GPUs still remains an important problem to address.

The fast adoption of GPUs in supercomputing, cloud services, and data centers poses

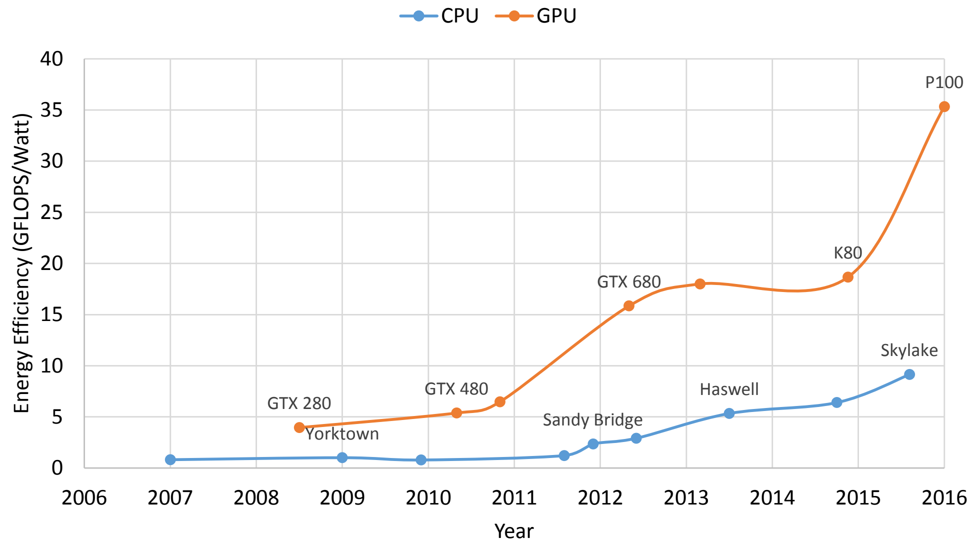


Figure 1.2: Comparison of CPU and GPU energy efficiency.

another dimension of problems: resource sharing becomes critical to achieving efficient resource utilization, similar to that shown for CPUs [88]. In these environments, multitasking CPUs are used by default, where time-sharing of the processor between multiple programs increases CPU utilization significantly, while maintaining the illusion that each program has its own CPU for execution. Cooperative multitasking, where applications voluntarily gave up the CPU to execute another application, was once widely used by many operating systems, however, it eventually gave its way to preemptive multitasking as the decision can be made dynamically without asking programmers to explicitly write multitasking code. In multitasking CPUs, preemptive multitasking supported by context switching has been standard for many years. Similarly, to support preemptive multitasking on GPUs, a new approach, which addresses both hardware and software extensions, is required for efficient preemption as GPUs have large states including registers and data in the scratchpad memory.

GPUs have multiple streaming multiprocessors (SM) on a single chip, which can be

thought of as GPU cores similar to the CPU cores. With preemptive multitasking support, GPUs can choose to divide resources among kernels in two ways: spatial multitasking [3], which partitions at SM granularity, or simultaneous multikernel (SMK) [90], which partitions within an SM. Spatial multitasking can be thought of as chip multi-processors (CMPs) [53] on CPUs, and SMK can be thought of as simultaneous multithreading (SMT) [86] on CPUs. Due to these similarities, shared resources like functional units, caches, memory controllers, DRAM bus, and on-chip networks are also points of contention on GPUs, which has been extensively stuided on CPUs [97]. On top of these problems, there are new challenges for multitasking GPUs. For example, a resource fragmentation problem can be significant on multitasking GPUs, where a thread block cannot be scheduled because resources are available in small chunks although there are enough resources on an SM in total. To enable efficient multitasking GPUs, these new challenges have to be identified and solved.

1.1 Challenges

There are three challenges with efficient resource utilizations on multitasking GPUs. While similar challenges have been solved on CPUs, however, the same strategies cannot be directly applied to the GPUs due to the differences in the programming model and the architecture.

Efficient single kernel execution: CPUs have improved the single thread performance primarily using out-of-order execution. GPUs have deliberately taken another approach to achieve higher performance per watt efficiency. On GPUs, the issue unit switches between

thousands of threads to hide the latency of the individual operations. However, a memory latency is still not hidden completely because the latency is in the range of hundreds of cycles, which is difficult to hide with warps in the range of ten. This situation often occurs for memory-intensive applications with intense memory operations. Another approach to improve performance for memory-intensive kernels is required.

Efficient preemptive multitasking: Traditionally, CPUs utilized context switching to enable preemptive multitasking, however, the same strategy can incur larger overhead for GPUs due to the large states in a GPU kernel. This is due to the GPU programming model, where thousands of threads are running together on a GPU core. Each thread on GPUs can have more state than a CPU thread, and this is multiplied by the number of threads. Another mechanism is required to enable efficient preemptive multitasking on GPUs.

Efficient resource management for multitasking GPUs: CMP adds more CPU cores to run multiple threads concurrently. Simultaneous multithreading improved resource utilization within a CPU core by sharing the processor between multiple threads in a more fine-grained fashion. On GPUs, these ideas are called spatial multitasking, which partitions resources at the SM granularity, and SMK, which runs multiple kernels concurrently on a single SM. SMK can further improve resource utilization over spatial multitasking. For example, compute-intensive kernels can utilize computational functional units on an SM while memory-intensive kernels wait for the memory requests to come back. However, spatial multitasking may provide better system performance when SMK results in higher contention within the SM. Moreover, enabling SMK on GPUs requires additional hardware extensions, and poses new problems like resource fragmentation. An efficient multitasking GPU will only be available if these problems are addressed.

1.2 Contributions

This thesis designs and evaluates a framework with hardware/software extensions for efficient resource utilization on multitasking GPUs. The hardware extensions are designed to be simple, so that the hardware overhead can be minimized. Complicated functionalities are offloaded to the software extension, which includes both compiler and runtime. The hardware extensions consist of various performance counters to monitor the runtime properties, and the logic to carry out the decisions made by the software extensions. The software extensions statically or dynamically analyze the execution state, and notify the hardware extensions of what to do. As a whole, the framework can efficiently support single kernel execution, preemptive multitasking, and simultaneous multikernel on GPUs.

1.2.1 *ELF*

GPUs rely on fast context switching between thousands of threads to hide long latency operations, however, they still stall due to memory operations. To minimize the stalls, memory operations should be overlapped with other operations as much as possible to maximize memory-level parallelism (MLP). In order to deal with memory-intensive applications on GPUs, we propose Earliest Load First (*ELF*) warp scheduling [60], which maximizes the MLP by giving higher priority to the warps that have the fewest instructions to the next memory load. *ELF* leverages compiler techniques to identify and transfer necessary information to compute the number of remaining instructions to the next memory load, and suggests a simple hardware scheduler that can prioritize warps accordingly. With *ELF*, memory-intensive applications on GPUs can transparently achieve better per-

formance. Chapter III discusses *ELF* in more detail.

1.2.2 *Chimera*

While single kernel execution can be explored for better performance, multi-kernel execution can provide even more efficient resource utilization. Multitasking GPUs are inevitable. However, supporting preemptive multitasking on GPUs through context switching can incur a higher overhead compared to CPUs, where the context of an SM can be as large as 256kB of register file and 96kB of on-chip scratch-pad memory on the recent Maxwell architecture [50]. To overcome these challenges, *Chimera* [59], a collaborative preemption approach for GPUs that can precisely control the preemption overhead, is proposed. *Chimera* first introduces SM flushing, a GPU-specific preemption technique that is enhanced to exploit the semantics of thread blocks in the GPU programming model and the concept of idempotence to achieve low preemption latency. Furthermore, *Chimera* exploits the fact that preempted thread blocks do not have to use the same preemption technique because thread blocks are executed independent of each other. By recognizing the different tradeoffs of three preemption techniques, namely, context switching, draining, and flushing, *Chimera* can preempt each thread block with the most efficient preemption technique by dynamically estimating the preemption costs. Chapter IV discusses *Chimera* in more detail.

1.2.3 *GPU Maestro*

Depending on the application mixes, either running multiple kernels at an SM granularity on the shared GPUs or running multiple kernels within an SM can be more ad-

vantageous. Predicting which resource partition will perform the best is difficult because of the dynamism within a kernel and interference between kernels. *GPU Maestro*, which performs dynamic resource management for efficient utilization of multitasking GPUs, is proposed to solve these problems. *GPU Maestro* addresses three challenges: (1) *GPU Maestro* implements a lightweight dynamic scheduling framework that utilizes a direct measurement with existing performance counters rather than a model-based prediction to find the best performing resource partition, (2) *GPU Maestro* solves resource fragmentation problem on SMK GPUs using 2-way resource allocation, and (3) *GPU Maestro* adopts a simple kernel-aware warp scheduler to avoid starvation of one kernel. Chapter V discusses *GPU Maestro* in more detail.

CHAPTER II

Background

This chapter introduces the GPU terminology, programming and execution models, and the architecture. Nvidia's terminology is used throughout the paper.

2.1 Terminology

The GPU programming model is based on a single instruction multiple thread (SIMT) model to explicitly express the parallelism in the *kernel* code, which is the parallel code section that runs on the GPUs. In the SIMT model, a programmer writes a code for a *thread*. In the GPU hardware, the threads are executed in a group called a *warp*. A warp is not an exposed concept to the GPU programming model, but rather a micro-architectural decision to process threads efficiently. In Nvidia's GPU architecture, a warp consists of 32 threads. The programmer also specifies a group of threads called a *thread block*. The threads within the same thread block can be synchronized with an explicit barrier operation, and have access to a common, fast, on-chip scratch-pad memory called *shared memory*. Finally, a *grid*, which is a group of thread blocks, will be grouped to form a kernel.

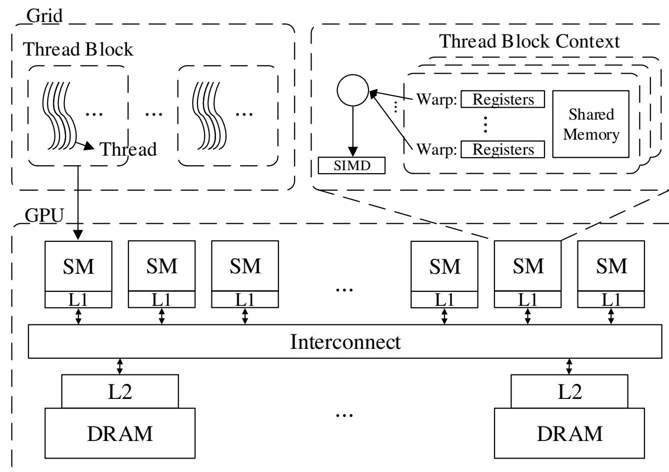


Figure 2.1: GPU architecture and execution model.

2.2 GPU Architecture and Execution Model

Figure 2.1 illustrates a GPU architecture and its execution model. The top left box shows a kernel with the notion of a GPU programming model. The bottom box depicts the GPU architecture with a memory hierarchy. The top right box represents a GPU execution model with the contexts of running thread blocks. In a GPU, each streaming multiprocessor (SM) has a private L1 data cache, a read-only texture cache, and a read-only constant cache. The memory subsystem of the GPU consists of multiple memory partitions. Each memory partition contains a shared L2 cache bank and a memory controller.

The GPU execution model relies on the notion of a thread block. Thread blocks from a kernel can run in arbitrary order because thread block executions are independent from each other. When a kernel is launched, each thread block is scheduled to one of the SMs. Depending on the resource constraints, the number of thread blocks that can run simultaneously on an SM may vary. When a thread block is dispatched to an SM, the thread block is split into groups of 32 threads called *warps*, where threads within a warp operate on a single common instruction. Each warp has its own register contents, and shares the state

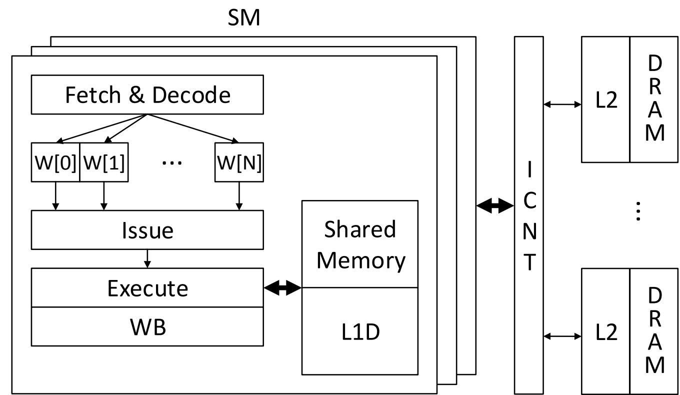


Figure 2.2: A more detailed illustration of an SM. W[i] denotes a warp.

of a shared memory if they are in the same thread block. Note that SMs do not share any states among themselves.

Figure 2.2 illustrates the detailed execution pipeline of an SM. Instruction fetch, decode, issue, and execute are performed at a warp granularity. Note that the figure neglects a read-only texture cache and a read-only constant cache in each SM for simplicity as did in Figure 2.1. In an SM, fetch and issue pipelines are shared by all the warps. Because there is no dependency between the execution of the warps except for the synchronization within thread blocks, deciding which warp should fetch or issue an instruction among ready warps is an important problem in GPUs. Within an SM, there are typically multiple warp schedulers, which issue from independent subsets of warps.

2.3 Multitasking GPUs

Figure 2.3 illustrates a baseline GPU architecture with multitasking support. At the top, a command processor is shown, which receives the commands from the CPU and controls the GPU accordingly, and updates memory-mapped registers when the command is done. In the middle, the thread block scheduler is depicted with control information

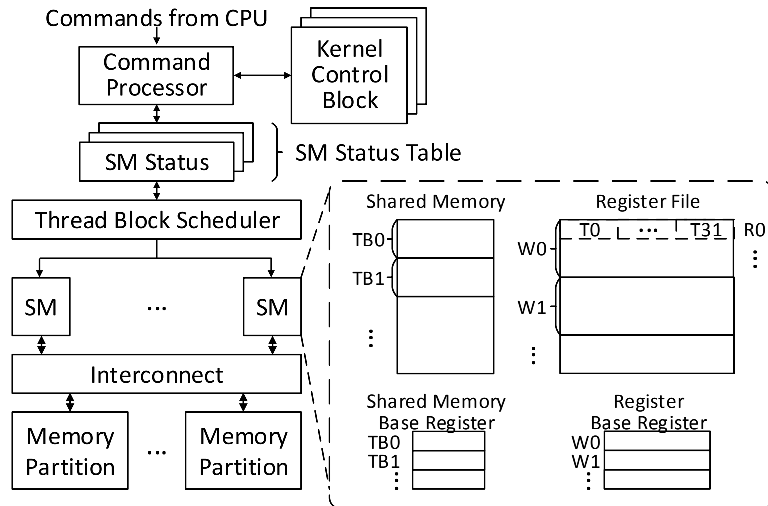


Figure 2.3: GPU architecture with multitasking support.

structures. At the bottom left, SMs and memory partitions are shown with an interconnect. At the bottom right, shared memory and register file allocation within an SM is shown. Both shared memory and register file are multi-banked structures although they are shown as one structure for simplicity. The thread block scheduler determines how many thread blocks from each kernel will be launched on each SM. A thread block is preempted if the thread block scheduler decides to reduce the number of thread blocks running from one kernel to allow thread blocks from other kernels to run. The thread block scheduler also launches a thread block with the chosen partition when an SM has enough resources to accept a new thread block. It has to remember the preempted thread blocks so that it can relaunch those thread blocks later.

When a kernel is launched, the related information will be stored at a kernel control block (KCB), whose name is borrowed from process control block (PCB) in the operating system [78]. Similar to PCB, the KCB stores the kernel-specific information, which is necessary to launch and run a kernel on the GPU. The KCB contains which CPU pro-

cess launched the kernel, information on memory management structures (e.g. page table), the grid size, the number of executed thread blocks, the resource requirement of a single thread block, and runtime information such as instructions per cycle (IPC) and average thread block execution cycles. The runtime information can be used to enable low overhead preemption [59]. To support multitasking on GPUs, the KCB is extended to multiple entries. Note that current generation GPUs are likely to have similar structures already if they support Hyper-Q, which allows multiple independent kernels from the same process to concurrently run on the GPU.

An SM status table (SMST) stores the current status of SMs. An SM status includes kernels executing on the SM, the state of each thread block (free, running, or preempting), the mapping between thread blocks and kernel, and remaining resources on the SM such as registers, shared memory, the number of threads that can be launched concurrently on an SM, and the number of thread blocks that can be launched concurrently on an SM.

As shown by the figure, the shared memory and register file are shared among the thread blocks within an SM. Because shared memory is shared at a thread block granularity, each thread block allocates a consecutive region in the shared memory. Because thread blocks execute the same code to compute the shared memory address, physical address for shared memory has to be differentiated between thread blocks. The shared memory base register (SBR) contains the base shared memory address for each thread block. When a thread block accesses shared memory at runtime, the address is computed by adding the virtual address with the SBR. Similarly, consecutive registers are allocated to each warp. Note that a physical register actually contains the same register index for 32 threads to reduce the number of ports because a warp consists of 32 threads. When a warp is indexing register 0,

the physical register index is computed by adding the register index 0 with a warp's register base register (RBR).

Because there are multiple SMs on a GPU with no shared state among them, the easiest way to schedule multiple kernels on a GPU is to run each kernel on different subsets of SMs. Spatial multitasking [3] noticed this property, and explored possible SM partitioning policies. Other prior works have shown that spatial multitasking is orthogonal to preemptive multitasking, and can be used together to provide further benefits [83, 59]. Simultaneous multikernel (SMK) [90] or intra-SM slicing [93] proposed to further partition within the SMs. This thesis assumes that the baseline GPU is able to support all these features with required extensions. On top of the baseline GPU, this thesis proposes a framework with hardware/software extensions to further enable efficient resource utilization on multitasking GPUs.

CHAPTER III

ELF: Maximizing Memory-Level Parallelism for GPUs with Coordinated Warp and Fetch Scheduling

3.1 Introduction

The trend of using GPUs as throughput processors in modern computer systems is constantly increasing as their computing capability and energy efficiency exceed that of CPUs. GPU programming models such as OpenCL [34] or CUDA [48] launch thousands of threads together to the hundreds of processing units on the GPU. By context switching between this large number of threads quickly, GPUs can hide long latency operations and achieve high throughput.

As more diverse applications adopt GPUs to exploit their computing capability, many have reported the difficulty of achieving the peak performance [71, 72]. Many recent works attempted to tackle this problem [45, 68, 29, 33, 28, 75]. CCWS [68] modified warp scheduling to reduce L1 cache contention. DYNCTA [33] noted that reducing the number of concurrently running thread blocks can reduce memory contention. MRPB [28] showed the importance of prioritizing memory requests from the same warp. These works show

that cache/memory contention is one of the main reasons why GPUs are not achieving peak performance.

Cache/memory contention often makes GPUs stall because memory operations have the highest latency. GPUs schedule a single warp at a time to leverage data-level parallelism. When a warp is blocked due to an operation that is dependent on a memory operation, the warp is swapped for another warp until the memory request comes back. If all the warps are blocked, GPUs can no longer hide the memory latencies. To reduce the impact of these unhidden memory latencies, it is strongly suggested that memory operations are overlapped with each other as much as possible. In other words, memory-level parallelism (MLP) has to be maximized.

In this chapter, Earliest Load First (*ELF*) scheduling is proposed, which maximizes the MLP by issuing memory operations as soon as possible. To maximize MLP, *ELF* gives higher priority to the warps with fewer remaining instructions to reach the next memory operation. The highest priority warp will continue to issue instructions until it issues the next memory operation. *ELF* leverages compiler techniques to identify program points that are needed to calculate the priority, and annotate that information in the binary to notify the hardware.

In order to ensure that the highest priority warp continuously has instructions to issue, the fetch unit also has to fetch according to the issue priorities. Otherwise, a warp scheduler has to suffer because a lower priority warp will have to issue. By using the coordinated warp priority between fetch and warp scheduling, the decision from warp scheduling becomes more effective as the fetch unit supplies instructions to the warps that are trying to issue with higher priority. Moreover, instruction prefetch is employed to reduce fetch stalls that

can prohibit the highest priority warp from making progress.

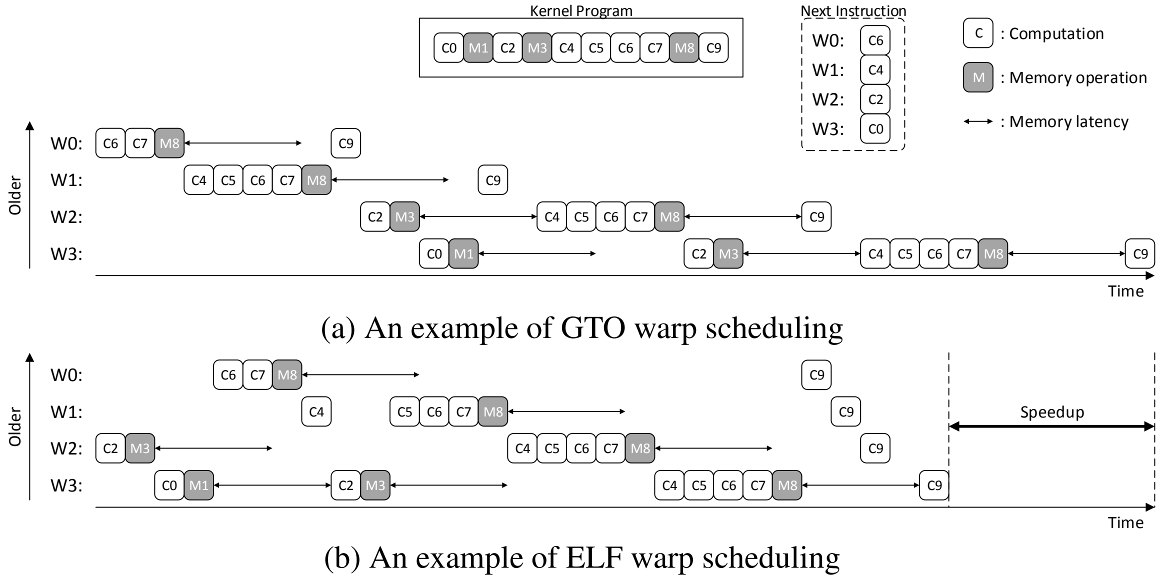
ELF can be limited by memory resource saturation when trying to issue a memory request. However, with multiple independent warps concurrently running on GPUs, the memory resource saturation from one warp does not necessarily mean that other warps will experience the memory resource saturation as well because other warps may see hit-under-miss or avoid associativity stalls. Without any solutions to allow *ELF* to issue memory requests, even if the previous memory request is blocked due to memory resource saturation, *ELF* may lose its effectiveness because it cannot further exploit MLP.

This chapter makes the following contributions:

- This chapter proposes *ELF*, a warp scheduling technique that maximizes the MLP by prioritizing warps with fewer remaining instructions to the next memory operation. Furthermore, this chapter also shows that the interplay between the warp scheduler and fetch unit is important in *ELF*.

- This chapter introduces a compiler technique that analyzes the program and passes the necessary information for priority calculation to the GPU hardware. With hardware/software co-design, the overhead of priority calculation can be minimized.

- This chapter evaluates the use of *ELF* with other orthogonal techniques, which help to avoid situations when the actual scheduling cannot follow the expected scheduling from *ELF*. This chapter also discusses an extended version of cache access re-execution (NewCAR) to handle memory conflicts, and an instruction prefetch to reduce stalls from the fetch unit.



(a) An example of GTO warp scheduling

(b) An example of ELF warp scheduling

Figure 3.1: Execution timeline of (a) greedy-then-oldest (GTO) warp scheduling, and (b) ELF warp scheduling for the given kernel program, where warps are at different execution points.

3.2 Motivation

This section motivates *ELF* with examples. Although the problem of issuing warps (or warp scheduling) has been studied extensively in the past [45, 68, 69, 29, 75], fetch scheduling has received less attention [38]. In this chapter, both problems are addressed.

3.2.1 Maximizing Memory-level Parallelism

Figure 3.1 illustrates an example execution timeline when greedy-then-oldest (GTO) scheduling [68] and *ELF* scheduling are applied. For simplicity, all the computations are assumed to have a single cycle latency, and all memory operations have a four cycle latency. Each instruction is also assumed to be dependent on the previous instruction. The top box denotes a kernel program with 10 instructions. The top right box shows the next instruction to be executed for each warp. Unless strict round-robin scheduling is used, it is likely that

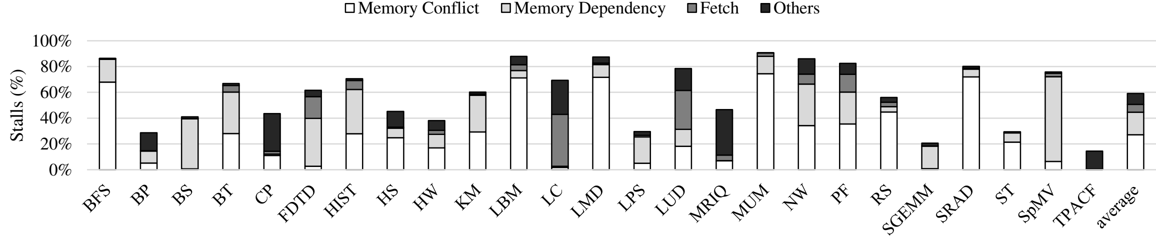


Figure 3.2: Distribution of stalls in the GTO scheduler. Memory conflict stalls occur when memory resources are saturated. Memory dependency stalls occur when instructions are waiting for the memory requests to come back. Fetch stalls happen when warps are waiting for instruction fetch. Other stalls include conflict and dependency stalls in other functional units.

warps are executing different instructions.

In Figure 3.1 (a), the GTO scheduler selects an instruction from the warp which was issued in the previous cycle. If the warp cannot progress because of a long latency memory operation, GTO selects the oldest warp among the ready warps. Because GTO prioritizes older warps, it cannot tolerate long memory latencies from younger warps. On the other hand, *ELF* in Figure 3.1 (b) tries to issue memory instructions as early as possible. Because memory instructions are issued quickly, their long latencies are overlapped more with themselves or other computations. As a result, *ELF* can achieve a better aggregate throughput with higher MLP.

The intuition behind *ELF* scheduling is simple. GPUs become idle when there are no ready instructions, which typically happens when warps are waiting for long latency memory operations to finish. If these memory operations can be scheduled earlier than the computations from other warps, their latency can be overlapped with these computations and other memory operations. For example, W2 and W3 send out their memory requests earlier in Figure 3.1 (b), which makes them ready for their compute-intensive phase (four

consecutive computations) earlier. The illustrative example in Figure 3.1 shows the importance of issuing long latency memory operations as early as possible.

3.2.2 Memory Conflicts and Fetch Stalls

If GPUs have infinite memory resources, issuing memory operations as soon as possible will always maximize the MLP. In reality, GPUs have limited memory resources, which can result in memory conflicts that prohibit the MLP to be maximized.

Figure 3.2 illustrates the distribution of stalls with GTO, which is percentage of cycles when a warp scheduler could not issue an instruction. The stalls are categorized into four classes: memory conflict stalls, memory dependency stalls, fetch stalls, and other stalls. Memory conflict stalls occur when a warp scheduler could not issue an instruction because of memory resource conflicts. These stalls occur when the load/store unit cannot accept any further instructions. Memory dependency stalls occur when a warp scheduler could not issue an instruction because at least one of the source operands in the instruction are waiting for the memory requests to come back. Fetch stalls occur when a warp scheduler is waiting for instruction fetch. Other stalls include conflict and dependency stalls from other functional units. For example, a dependency stall can occur for special function units (SFUs) because they can take multiple cycles.

As shown in the figure, there are 27.2% memory conflict stalls, 17.5% memory dependency stalls, 6.0% fetch stalls, and 8.4% other stalls on average. The results show that memory conflict stalls are already a dominant source of stalls, and can be a limiting factor when trying to maximize the MLP as in Figure 3.1 (b). Therefore, a strategy of maximizing the MLP should be accompanied by a technique, which can reduce the memory

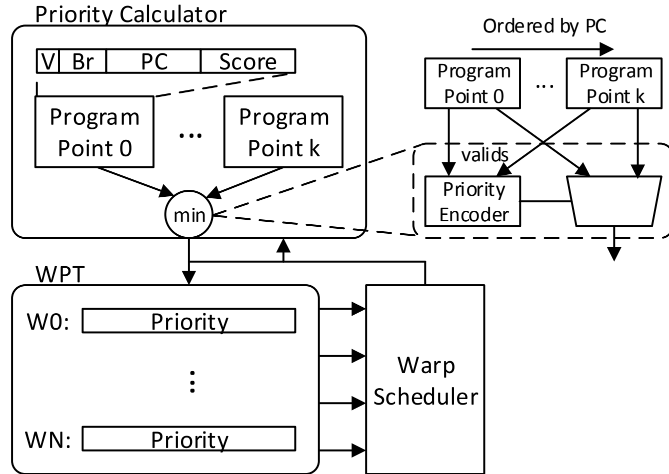


Figure 3.3: Overall architecture for *ELF*. There are two components in *ELF*: a priority calculator, and a warp priority table (WPT). The priority calculator calculates the priority of a warp using the program points generated by the compiler. The warp priority table stores the results from the priority calculator, and directs the warp scheduler to issue accordingly.

conflicts. Another important observation is that fetch stalls can take a large portion of stalls for benchmarks such as FDTD, LC, and LUD. In such benchmarks, a strategy of maximizing the MLP is again limited by the fetch unit. Therefore, a technique that can reduce fetch stalls should also be coupled with maximizing the MLP.

3.3 Architecture

ELF is a warp scheduling technique that utilizes both compiler and hardware to maximize MLP by prioritizing a warp that has the earliest memory load. *ELF* relies on the program points that are necessary to compute the priority. A *program point* is defined as an instruction that can change the distance to the next memory load. By definition, memory loads are program points. Branch instructions are also program points because they alter the control flow of the program. The remaining instructions are not program points.

Figure 3.3 illustrates the overall architecture for *ELF*. At the top, a priority calculator is shown, where program points and their related values are computed from the compiler side, and conveyed to the hardware through the binary. On the right side of the top box, an implementation of minimum functionality with a priority encoder is shown, which will be discussed in detail in Section 3.3.2. At the bottom, the warp priority table (WPT) is shown, which keeps the priority for each warp. In *ELF*, a priority equals the number of remaining instructions to the next memory load, hence, lower value in the priority means higher priority to issue an instruction. The priorities from the WPT are referenced by the warp scheduler when it issues.

3.3.1 Finding Program Points

ELF generates the list of program points from the compiler. In *ELF*, each program point is either a memory load operation or a branch. Among memory loads, shared memory loads, constant memory loads, and parameter loads are not considered because they are likely to be a cache hit, which will have similar latency as ordinary computations. In *ELF*, backward branches are only considered to reduce the number of total program points, however, forward branches could also be considered. These program points are notified to the GPU hardware, and further used by the priority calculator to compute the priority for each warp. Two hardware parameters are given to the compiler beforehand: the maximum number of program points is restricted by the number of available program point slots in the GPU, and the maximum score is limited by the bitwidth of the score field in the program point slot.

Each program point in *ELF* is associated with a data called score. For a branch program

point, it stores the minimum distance to the next memory load considering all the possible paths after the branch. For a load program point, score is meaningless and zero by default. *ELF* utilizes the score field of memory loads to merge program points as described by Algorithm 2 when the kernel has more program points than the available program points in the hardware.

Algorithm 1 shows how the program points are generated by the compiler. The algorithm starts by finding memory loads and branches to form initial program points (lines 1-9). Iteratively, *ELF* computes the score of branches by taking the minimum of the scores from the taken path and the fall-through path (lines 10-24). The score of each path is the addition of the number of instructions to the next program point in each path and the next program point's score (lines 36-45). If next program point does not exist, the maximum score is returned (line 46). After the program points are resolved, meaningless program points are first removed (lines 25-31). A branch program point is meaningless if it has the maximum score or the score of the fall-through path. Lastly, the program points are merged if they exceed the available slots in the GPU (lines 32-34).

Algorithm 2 shows how to merge program points. First, two program points are found, where the distance between the two is the minimum (lines 1-15). When merging program points, the program point that merges its previous program point has to be a memory load (line 3). Then, the score field of the later program point is updated with the distance to the previous program point (line 16). Note that the merged program point's score is subtracted if it is a branch (line 11). Finally, the merged program point is removed from the program points, which reduces the number of program points by 1 (line 17).

Algorithm 1 Finding Program Points

```
    findProgramPoints(Kernel):
    Output: ProgramPoints[1..MaxProgramPoints]
 1: for each inst in Kernel do
 2:     if inst is MemoryLoad then
 3:         inst.score = 0                         ▷ Lower score is higher priority
 4:         ProgramPoints.push(inst)
 5:     else if inst is Branch then
 6:         inst.score = MaxScore
 7:         ProgramPoints.push(inst)
 8:     end if
 9: end for
10: changed = true
11: while changed do
12:     changed = false
13:     for each inst in ProgramPoints do
14:         if inst is Branch then
15:             nextScore = getScore(inst.nextInst) + 1
16:             targetScore = getScore(inst.targetInst) + 1
17:             minScore = min(nextScore, targetScore)
18:             if minScore <inst.score then
19:                 inst.score = minScore
20:                 changed = true
21:             end if
22:         end if
23:     end for
24: end while
25: for each inst in ProgramPoints do
26:     if inst is Branch then
27:         if inst.score == MaxScore or inst.score == getScore(inst.nextInst) + 1 then
28:             ProgramPoints.remove(inst)
29:         end if
30:     end if
31: end for
32: while ProgramPoints.size() >MaxProgramPoints do
33:     ProgramPoints.merge()
34: end while
35: return ProgramPoints[1..MaxProgramPoints]

    getScore(inst):
    Output: score
36: currInst = inst
37: score = 0
38: while currInst.valid do
39:     if currInst in ProgramPoints then
40:         score = min(score + currInst.score, MaxScore)
41:         return score
42:     end if
43:     currInst = currInst.nextInst
44:     score += 1
45: end while
46: return MaxScore
```

Algorithm 2 Merging Program Points

```
    merge(ProgramPoints):
 1: minDistance = MaxScore
 2: for each inst in ProgramPoints do
 3:     if inst is MemoryLoad then
 4:         prevPoint = inst.getPrevProgramPoint()
 5:         currDistance = getScore(prevPoint.nextInst) + 1
 6:         if currDistance <minDistance then
 7:             minDistance = currDistance
 8:             memInst = inst
 9:             prevInst = prevPoint
10:             if prevPoint is Branch then
11:                 minDistance -= prevPoint.score
12:             end if
13:         end if
14:     end if
15: end for
16: memInst.score = minDistance
17: ProgramPoints.remove(prevInst)
```

3.3.2 Priority Calculator

The priority calculator loads the program points embedded in the binary when a kernel is launched onto an SM. As shown by the top box in Figure 3.3, each program point has a valid bit, a branch bit, PC, and a score. Using the PC and the score of a program point with the PC of an instruction, a distance to a memory load can be calculated. A priority, which is the minimum distance to the next memory load instruction, can be calculated by taking the minimum of the distances to all the memory loads. As shown in Figure 3.3, a priority encoder can practically implement the minimum functionality when program points are ordered by PC because it is guaranteed that the closest program point will give the minimum.

Given the instruction and a memory program point, it is possible to compute the number of instructions between the instruction and the memory load using the PC. Given the

instruction and a branch program point, the distance to the branch is first calculated using the PC, and then the score field is added to give the distance to the memory load after the branch. Note that a memory program point may also have a non-zero score if other program points have been merged to that program point. In such a case, the calculated priority is subtracted by the score if it is larger than the score to account for the merged program point.

Naively computing priority every time when a warp issues an instruction can have significant overhead. Because priority is the number of remaining instructions to the next memory load operation, it can be decreased by one most of the time. To exploit this property, the priority calculator is triggered only in two cases: when a memory load operation is issued, or when a branch has altered the priority. For the second case, *ELF* extends the instruction format of each branch with a recalculation bit. The recalculation bit indicates whether the priority is altered when the branch is taken. If it is set, a taken branch triggers the priority calculator. Otherwise, a branch triggers the priority calculator if it is not taken. The compiler sets the recalculation bit when the taken path has fewer remaining instructions to the next program point than the fall-through path. Priorities of not issued warps are not changed hence no computation is required.

3.3.3 Fetch Scheduling in *ELF*

Fetch scheduling can play an important role as the fetch unit is shared by the warps, as discussed in Section 2.2. Instruction fetches also use the same priority as the warp schedulers to prioritize warps. In Fermi, there are two warp schedulers, where each of them schedules among an independent subset of the warps, while a single fetch unit exists.

To match the issue width, the fetch unit requests two instructions for one warp in a cycle. To handle two distinct priority orders from the warp schedulers without starving one warp scheduler, *ELF* constructs unified priority order by interleaving the priority orders from each warp scheduler.

3.3.4 *ELF* with Cache Access Re-execution

ELF may not be able to maximize the MLP when memory conflicts occur. The problem of memory conflicts can be more severe in GPU execution model, where warps share the load-store unit (LSU), because not only the current memory request is blocked but also other memory requests from other warps that may be totally independent can be blocked as a result.

Prior works have proposed solutions that can mitigate the problem of memory conflicts. For example, MRPB [28] reported that GPUs can have an associativity stall because of the allocate-on-miss policy on the L1 cache. In such a case, MSHRs and miss queues cannot be utilized even though they are available. MRPB avoids such a problem with bypassing. Mascar [75] reported the opportunity of hit-under-miss, and exploited the opportunity with the cache access re-execution (CAR). Another problem exists because the LSU is shared among the constant, texture, and L1D cache. For example, when the LSU is blocked by L1D cache, memory requests that can be served by constant or texture cache are also blocked. Although the mentioned problems seem different, they can be solved at the same time with any of the previous solutions.

Instead of devising a completely new way to overcome these problems, *ELF* adopts the CAR with 32 entry re-execution queue from Mascar [75] with extensions. The NewCAR

is structurally similar to the CAR as shown in Figure 3.4. However, NewCAR operates with more relaxed conditions compared to CAR, which can reduce more memory conflict stalls. First, multiple memory requests per warp are allowed in NewCAR. This is intra-warp optimization, which allows a warp to progress more even with the memory conflicts. Second, memory requests in the queue can be processed out-of-order while preserving the weak memory consistency semantics. This is both intra and inter-warp optimization, which allows more freedom compared to CAR.

Figure 3.4 illustrates the extension of LSU with NewCAR. A NewCAR queue is attached to the LSU, where a memory request can enter the queue from the LSU. A memory request is inserted into the queue if one of the two conditions is met: 1) a memory request was sent to one of the caches and not accepted, or 2) a memory request cannot bypass the queue due to the memory consistency semantics. NewCAR always gives priority to the memory request from the LSU. A memory request from the queue is processed if one of the two conditions is met: 1) the queue is full, or 2) LSU is idle. Note that when the queue is full, the LSU is prohibited from issuing a new memory request until the queue has an empty slot.

Figure 3.5 shows which memory request reorderings are allowed when there is a prior memory request from a warp in the NewCAR queue. Because memory requests are independent between the warps, memory requests can always bypass another warp's memory requests. If a warp already has a load in the queue, it can issue a new load request ahead of the prior load but not a store request. If a warp has a store in the queue, it cannot issue any new memory request before the store has been processed. Whenever the memory request is not allowed to bypass the queue due to the violation of memory consistency semantics,

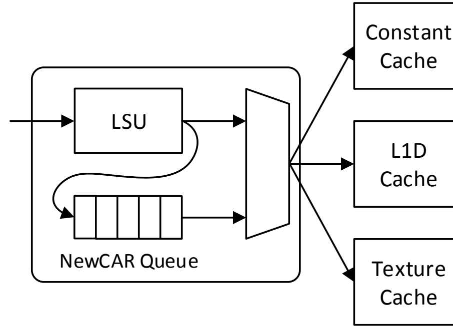


Figure 3.4: Extending LSU with NewCAR. Structurally, NewCAR is almost identical to CAR [75]. The key difference is more relaxed conditions on when and how the NewCAR queue is controlled.

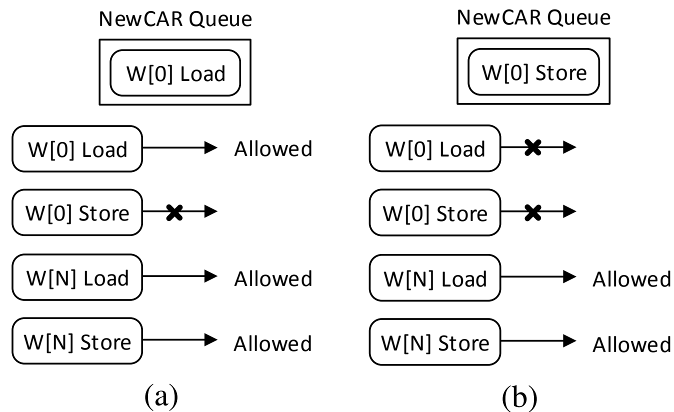


Figure 3.5: Allowed reordering of memory requests when a warp has (a) a load, and (b) a store waiting in the NewCAR queue. W[0] denotes an example warp that has a memory request in the NewCAR queue, and W[N] denotes any other warp. Memory requests from other warps can always bypass the memory request from the example warp. Loads from the example warp can bypass the loads from itself, but not the stores. Stores from the example warp cannot bypass any loads or stores from itself.

it goes straight into the NewCAR queue.

3.3.5 *ELF* with Instruction Prefetch

Fetch stalls can stop high priority warps from issuing instructions in *ELF* because they need to wait for an L1I cache miss as discussed in Section 3.2.2. To reduce such waits, a simple next line prefetcher [79] is employed for the L1I. However, naively using

System	Parameters
SM	15 SMs, 1400 MHz, 32 SIMT width 32768 registers per SM 1536 maximum threads per SM 8 maximum thread blocks per SM 48 kB shared memory
Memory Subsystem	2 kB/4-way/128B L1I per SM 8 MSHRs per L1I 16 kB/4-way/128B L1D per SM 32 MSHRs per L1D 768kB/16-way/128B L2 32 MSHRs per L2 partition 6 memory partitions FR-FCFS DRAM scheduler 177.4 GB/s bandwidth

Table 3.1: System configuration

a prefetcher on GPUs can increase the memory contention, which can negatively impact the GPU performance as studied by prior work on data prefetchers [39, 74]. To avoid this problem, the occupancy of MSHRs can be monitored to determine whether there is memory contention. The next line prefetcher for the L1I only issues a prefetch request when the total number of occupied MSHRs in both L1I and L1D is less than a threshold. As shown in Section 3.4.1, an instruction prefetch is issued only when the total number of occupied MSHRs is less than 16.

3.4 Results

The GPGPU-Sim v3.2.2 [5] is extended to evaluate *ELF*. GPGPU-Sim only models the GPU, where the host code and the overhead of data transfers between the CPU and the GPU do not affect the simulation results. A Fermi [46] architecture is modeled, which is similar to the Nvidia GTX480. The detailed configuration is listed in Table 3.1. For *ELF*,

32 program point slots were assumed to be available, and the score field is 8-bit, which can store up to a distance of 256 instructions. The compiler part of *ELF* is implemented as a part of the run-time system, which performs all the necessary analysis before the kernel is launched and passes the generated information to the GPGPU-Sim.

A wide range of GPGPU applications from Nvidia SDK [51], GPGPU-Sim [5], Rodinia v2.4 [7], and Parboil [82] benchmark suite are used for evaluation. Table 3.2 lists all the evaluated benchmarks, their labels, and kernels with the number of program points before merge. Trivial kernels from SDK were left out, and a few benchmarks from other suites that took too much time to simulate even with the smallest input.

ELF and other two orthogonal techniques are first explored to illustrate their individual performance improvements over the baseline greedy-then-oldest (GTO) scheduling. *ELF++*, which uses all the techniques together, is also evaluated to show whether the improvements are additive or have synergy when used together.

ELF++ is compared with three prior warp scheduling policies: 2-LV [45], CCWS [68], and DYNCTA [33]. For 2-LV, the version provided with the GPGPU-Sim v3.2.2 is used. For CCWS, we used publicly available version, which is based on a prior GPGPU-Sim version than the one that evaluates *ELF*. The GPU configuration is modified to match the baseline Fermi architecture that resembles the GTX480. To be fair, the GTO in CCWS version was used as the baseline of CCWS. DYNCTA was implemented and verified with the results from the prior published work.

Benchmark	Label	Kernel	PP
Breadth First Search [7]	BFS	Kernel	9
		Kernel2	1
Back Propagation [7]	BP	bpnn_layerforward	2
		bpnn_adjust_weights	12
BlackScholes [51]	BS	BlackScholesGPU	4
B+ Tree [7]	BT	findRangeK	22
		findK	13
Coulombic Potential [82]	CP	cenergy	3
3D Finite-Difference Time-Domain [51]	FDTD	FiniteDifferences	60
Histogram [82]	HIST	prescan_kernel	7
		intermediates_kernel	16
		main_kernel	15
		final_kernel	20
HotSpot [7]	HS	calculate_temp	2
Heart Wall [7]	HW	kernel	94
Kmeans [7]	KM	invert_mapping	2
		kmeansPoint	3
Laplace-Boltzmann Method [82]	LBM	performStrideCollide	20
Leukocyte Tracking [7]	LC	GICOV_kernel	4
		dilate_kernel	3
		IMGVF_kernel	10
LavaMD [7]	LMD	kernel_gpu_cuda	23
3D Laplace [5]	LPS	GPU_laplace3d	5
LU Decomposition [7]	LUD	lud_diagonal	16
		lud_perimeter	48
		lud_internal	3
Magnetic Resonance Imaging [82]	MRIQ	ComputePhiMag	2
		ComputeQ	5
MUMmerGPU [7]	MUM	mummergpuKernel	9
		printKernel	12
Needleman Wunsch [7]	NW	needle_cuda_shared_1	19
		needle_cuda_shared_2	19
Particle Filter [7]	PF	likelihood	22
		sum	2
		normalize_weight	8
		find_index	5
RadixSort [51]	RS	radixSortBlocks	2
		findRadixOffsets	1
		reorderData	4
Matrix Multiply [82]	SGEMM	mysgemmNT	20
Speckle Reducing Anisotropic Diffusion [7]	SRAD	srad_cuda_1	9
		srad_cuda_2	10
Stencil [82]	ST	hybrid_coarsen_x	13
Sparse Matrix Vector Multiply [82]	SpMV	spmv_jds	11
Two Point Angular Corr. Function [82]	TPACF	gen_hists	8

Table 3.2: Benchmark specifications to evaluate *ELF*. PP refers to the number of program points before the merge.

3.4.1 *ELF* Performance

Figure 3.6 shows the individual performance improvement of NewCAR, instruction prefetch, and *ELF* over the baseline GTO as well as *ELF*++, which uses the three techniques together. On average, NewCAR, instruction prefetch, *ELF* and *ELF*++ improve the performance by 2.5%, 4.9%, 4.1%, and 11.9%, respectively. While NewCAR and in-

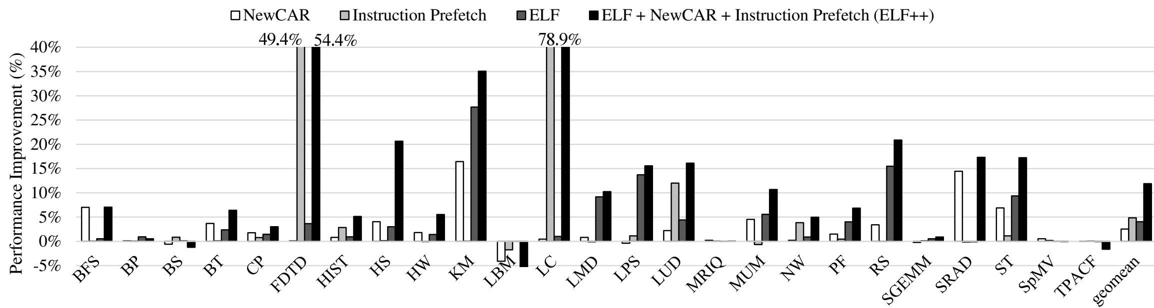


Figure 3.6: Performance improvement of NewCAR, instruction prefetch, *ELF*, and *ELF*++ over the baseline GTO. *ELF*++ combines *ELF* with NewCAR and instruction prefetch.

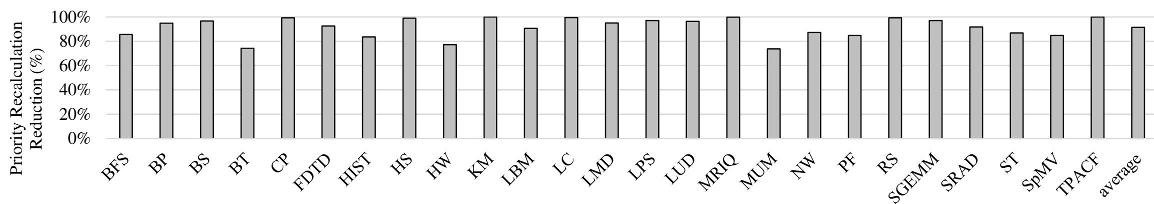


Figure 3.7: Reduction of priority recalculation in *ELF* compared to the naive priority recalculation.

struction prefetch are very effective for a few benchmarks, they do not provide consistent benefit across all the benchmarks. On the other hand, *ELF* is broadly effective among the benchmarks. Also, NewCAR may degrade the performance, as in the case of BS and LBM, because the locality may be lost in the cache when there are many requests waiting in the re-execution queue. FDTD and LC, which show large improvements from instruction prefetch, correspond to the benchmarks with high fetch stalls from Figure 3.2. The result illustrates that the fetch scheduling can be as important as the warp scheduling.

As shown in the figure, *ELF* performs better than the baseline GTO because the MLP is maximized in warp scheduling as well as the fetch scheduling. Note that *ELF* improves performance for not only the memory-intensive benchmarks like BFS and KM [68] but also the compute-intensive benchmarks like CP and HS [33].

When *ELF* is combined with NewCAR and instruction prefetch, they compensate each other. As shown by the results, *ELF++* shows additive improvements from the three techniques, and a synergy of 0.4% additional improvement on average. The synergy mostly comes from *ELF* and NewCAR, where *ELF++* can exploit more MLP with NewCAR because NewCAR reduces memory conflict stalls.

Figure 3.7 illustrates the percentage of reduction in the number of priority recalculation in *ELF* compared to the naive priority recalculation. The number of priority recalculation can be reduced by 91.4% on average, by only invoking the recalculation when necessary. Because a warp progresses by one instruction most of the time, which reduces the distance to the next memory load by one, most of the priority recalculation can be avoided. The reduction is bounded by the ratio of memory loads and branches in the total executed instructions as they are the only source of priority recalculation. For example, BT has less reduction because it has more fraction of memory loads than the other benchmarks.

Figure 3.8 (a) depicts the sensitivity of *ELF++* to the number of available program points on a GPU. On average, performance is improved by 11.9%, 11.9%, 11.9%, and 12.1% when the number of program points is 16, 32, 48, and infinite, respectively. As expected, the performance is dropped when the number of program points changes from infinite to a finite number. Although performance on average slightly improves as the number of program points is decreased, however, individual benchmark shows a random trend. For example, HW, which has the largest number of program points, shows the peak performance when the number of program points is 32. *ELF* chooses to merge the two program points with the least distance between them, but the removed program point may be an important load that should be considered for priority calculation. Although profiling

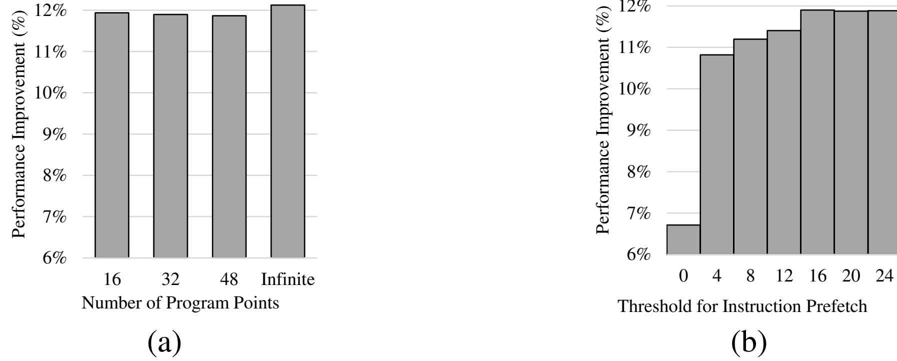


Figure 3.8: Sensitivity of *ELF++* to (a) the number of available program points, and (b) the threshold for instruction prefetch. A geometric mean of IPC improvement for all the benchmarks is presented.

can help to identify the critical program points, but it is left as a future work because the overall difference is small.

Figure 3.8 (b) shows the sensitivity of *ELF++* to the threshold for instruction prefetch, where an instruction is prefetched only when the number of occupied MSHRs in both L1I and L1D is below the threshold. Instruction prefetch is disabled when the threshold is zero. Performance is improved by an average of 6.7%, 10.8%, 11.2%, 11.4%, 11.9%, 11.9%, and 11.9% when the threshold is 0, 4, 8, 12, 16, 20, and 24, respectively. As discussed in Section 3.3.3, blindly issuing instruction prefetch can increase the memory contention, which can degrade the overall performance. Therefore, performance is expected to improve until a certain threshold, where instruction prefetch provides benefit without congesting the memory too much. In the figure, the curve has the maximum improvement at threshold of 16 although the performance degradation is small afterwards. Looking at the performance of individual benchmarks, a subset of benchmarks like MUM, which lose performance with a larger threshold, only shows small performance degradation. Benchmarks like FDTD, LC, and LUD, which obtain the most benefit from instruction prefetch, were not negatively

Structure	Storage per Entry	# Entries	Total
Priority Calculator	42-bit	32	0.16 kB
Warp Priority Table	8-bit	48	0.05 kB
NewCAR	301-bit	32	1.18 kB

Table 3.3: Hardware overhead per SM from additional structures.

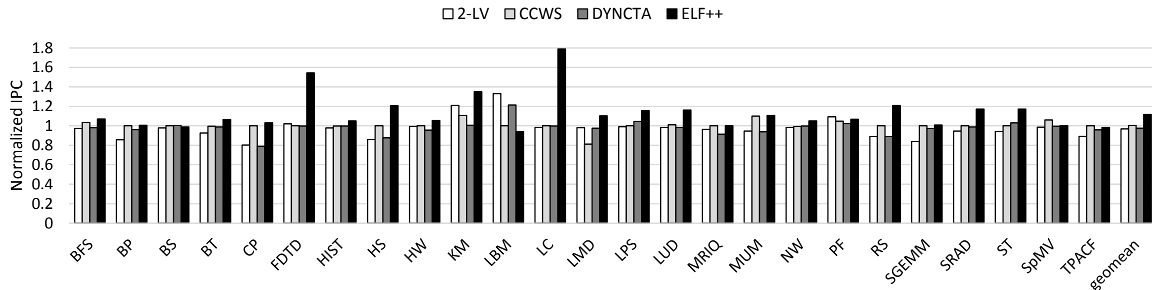


Figure 3.9: Comparison of *ELF++* with prior works. GTO is the baseline.

impacted by the increased threshold. Nevertheless, *ELF++* chooses to use 16 as the threshold in all the experiments because the performance improvement is maximized at the point although the difference is small.

3.4.2 Hardware Overhead

Table 3.3 shows the hardware overhead of *ELF++* per SM. Priority calculator has 32 program points, where each program point requires 1-bit for the valid field, 1-bit for the branch field, 32-bit for the PC, and 8-bit for the score. The WPT requires 8-bit per warp for the priority. GTX480 can have up to 48 warps per SM. NewCAR requires the same overhead as the CAR [75]. In total, *ELF++* only consumes 1.39kB extra storage space per SM.

3.4.3 Comparison to Prior Works

Figure 3.9 compares the performance improvement of *ELF*++ over GTO with three prior works: 2-LV [45], CCWS [68], and DYNCTA [33]. On average, 2-LV, CCWS, DYNCTA and *ELF* ++ improves -3.2%, 0.1%, -2.3%, and 11.9% over the baseline GTO. With the wide range of applications, 2-LV and DYNCTA perform worse than the GTO. For example, BP, CP, HS, and RS show noticeable slowdown in 2-LV and DYNCTA. On the other hand, *ELF*++ always shows either similar or better performance compared to the GTO.

2-LV and DYNCTA perform slightly worse than GTO on average. The only difference between 2-LV and GTO is that 2-LV continuously gives higher priority to the newer group until it issues all the ready instructions while GTO switches back to the older group (thread block) more quickly. Therefore, 2-LV and GTO provide similar performance on average. In DYNCTA, fewer thread blocks are scheduled to an SM to reduce the memory contention. Because warps are scheduled to an SM in a thread block granularity, GTO will schedule the warps in the oldest thread block more frequently than the other warps achieving similar effect to DYNCTA. As a result, DYNCTA performs similar to GTO.

CCWS reduces cache thrashing by limiting the number of warps that can send out new memory requests. Therefore, CCWS provides performance improvement over for cache-sensitive benchmark like KM, PF, and SpMV. However, in general, there are more benchmarks that are not cache-sensitive. As a result, CCWS provides similar performance to GTO on average.

In LBM, 2-LV and DYNCTA performs better than GTO while *ELF*++ performs slightly

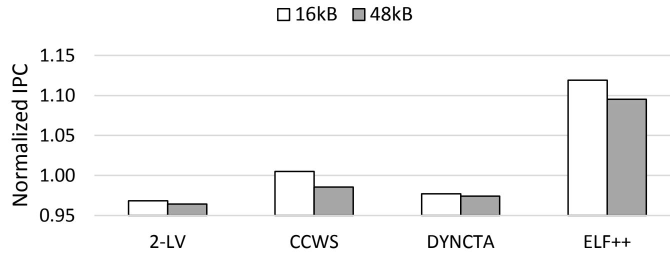


Figure 3.10: Comparing *ELF*++ to prior works when different cache configurations are used. GTO is the baseline.

worse than GTO. LBM has a lot of memory resource conflicts all the time because twenty memory loads are clustered back-to-back at the beginning of the program. These memory loads show row locality on the DRAM side when the warps within a thread block issue the same memory instruction. However, GTO and *ELF*++ will prioritize one warp until it issues all the twenty memory loads, which takes more than half of the MSHRs. As a result, GTO and *ELF*++ do not perform well as they lose the row locality on the DRAM side. In fact, any warp scheduler that round-robins within a thread block performs better than the warp schedulers that prioritize a warp in greedy fashion. For example, LRR scheduler achieves similar performance to 2-LV and DYNCTA for LBM.

Figure 3.10 compares *ELF*++ with prior works when cache configuration is different. GTX480 can be adjusted to have either 16kB L1D or 48kB L1D. As shown by the figure, all the techniques have slightly less improvement compared to the baseline GTO with 48kB L1D. On average, normalized IPC equals to 0.964, 0.986, 0.974, and 1.095 for 2-LV, CCWS, DYNCTA, and *ELF*++ with 48kB L1D. This is because memory requests are likely to have more L1D cache hits with a larger L1D as it can retain more data. Consequently, the average memory latency is reduced with the larger L1D. *ELF*++ improves over GTO and other prior works in both cache configurations because it can issue memory

requests faster than other schedulers, which can hide miss latencies better. As a result, *ELF++* overlaps more long latency memory operations with each other, which reduces the number of memory-related stalls more than other schedulers.

3.5 Related Work to *ELF*

Since the emergence of GPUs as a general-purpose parallel computing platform, a significant number of studies have been proposed to improve GPU performance. Prior works that exploited memory-level parallelism in various platforms are first reviewed, and then GPU-specific ideas that enhance GPU performance with alternative warp or thread block scheduling are summarized.

3.5.1 Memory-Level Parallelism

Memory-level parallelism (MLP) was recognized as one of the key objectives in computer architecture since CPUs' performance overwhelmed the memory performance [21]. Pai and Adve [56] applied code transformations to cluster more read misses within the same instruction window of an out-of-order processor. They also measured the improved MLP by measuring MSHR utilizations. Many other works [81, 41] also used the number of occupied MSHRs or stall cycles due to full MSHRs to measure MLP. Zhou and Conte [95] used value prediction techniques to increase MLP by parallelizing loads that were sequentially dependent. MLP-aware replacement [66] notes that an out-of-order processor can exploit more MLP if the LLC misses occur in parallel. In that sense, it tries to remove isolated misses in the LLC. Chou et al. [10] showed that runahead execution [15, 44], ef-

fective instruction prefetching, and accurate branch prediction can improve MLP, which in turn enhances the overall performance. The challenge of achieving high MLP on GPUs is different from the CPUs because latency hiding on GPUs is done by fast context switch between warps, where inter-warp data locality is scarce.

Some prior works have studied data prefetching or direct memory access (DMA) on GPUs, which essentially improves MLP on GPUs. MT-prefetching [39] prefetches data for other threads rather than for itself. Also, it shows that throttling mechanism may be needed for prefetching when prefetching is harmful because of low accuracy. APOGEE [74] introduces timely prefetching by adjusting the distance of prefetching. APOGEE mainly focuses on improving energy efficiency by reducing the number of thread contexts to hide the memory latency with prefetching. D^2MA [27] achieves MLP on GPUs by transferring data from the global memory to the shared memory behind the scenes. *ELF* also tries to maximize memory-level parallelism, but modifies the warp and fetch scheduler to achieve the goal, which includes instruction prefetching.

3.5.2 GPU Scheduling

Early works have noticed that it is better to prioritize a group of warps rather than giving equal priority to all the warps. Two-level warp scheduling [45] divides warps into groups, where the scheduling algorithm is re-structured into scheduling warps within a group and scheduling between the groups. Two-level warp scheduler only schedules from another group when all the warps within one group are blocked by long latency operations. Gebhart et al. [20] also utilize the two-level scheduling for energy-efficiency.

Some of the works have focused on utilizing warp scheduling to reduce cache con-

tention. CCWS [68] observes that intra-warp locality is dominant in the L1 data cache for GPUs. By monitoring the L1 cache behavior, CCWS only allows a subset of active warps to issue memory requests to exploit the intra-warp locality in the L1 data cache. DAWS [69] also uses the idea of throttling a subset of warps from issuing requests, however, DAWS is also aware of memory divergence, which occurs when memory requests from threads within a warp cannot be coalesced. DAWS predicts a L1 cache footprint for each warp either by profiling or run-time sampling, and uses the information to find the number of warps to be throttled. OWL [29] proposes four schemes, which include thread block aware warp scheduling, locality aware scheduling, bank-level parallelism aware scheduling, and opportunistic prefetching.

Some other works have looked at controlling the number of scheduled thread blocks. DYNCTA [33] shows that issuing maximum number of thread blocks to an SM is not always beneficial because memory-intensive applications create cache contention. By looking at how many cycles are stalled by memory, DYNCTA tries to predict the optimal number of thread blocks to be scheduled. However, the choice of the thresholds is dependent on micro-architecture. LCS [40] notes that GPUs are designed to hide latencies by leveraging the thread-level parallelism (TLP). In that sense, the optimum number of thread blocks to be scheduled on an SM is the number of thread blocks that can hide latencies until one thread block finishes. LCS monitors the number of instructions issued for each thread block during an execution of a single thread block, and calculates the number of thread blocks to be scheduled.

More recent works have focused on prioritizing memory requests from one warp, and explored a possibility of doing useful works when the load/store unit is stalled. MRPB [28]

prioritizes memory accesses from the same warp to preserve locality in the L1 data cache. MRPB also notes that an associativity stall may occur because of the allocate-on-miss policy on the L1 cache. Even if MSHRs and miss queues are available, the memory request cannot be accepted because there is no more way to allocate. In such cases, MRPB bypasses the L1 cache. Mascar [75] schedules the warps with outstanding memory requests first when memory resources are saturated. Mascar also utilizes cache access re-execution to enable hit-under-miss.

ELF shares the basic idea that the performance in GPUs can be improved by carefully scheduling warps. However, *ELF* primarily differs with prior works by giving higher priority to the warps with sooner memory operations. Also, *ELF* shows that reducing memory conflicts and fetch stalls can provide additional improvement.

3.6 *ELF* Conclusions

This chapter presented *ELF*, a GPU scheduling mechanism that maximizes the MLP as fast as possible by scheduling a warp with an earliest memory load first. In order to achieve the goal, *ELF* utilizes both compiler and hardware to give higher priority to the warps that have fewer remaining instructions to the next memory load operation. The importance of the interplay between the warp scheduler and the fetch scheduler was shown for *ELF*: the fetch unit should prioritize according to the issue priorities. Moreover, two cases were identified when *ELF* could improve the performance further by explaining when *ELF* is not fully achieving its goal. First, memory conflicts can block new memory requests. An extended version of cache access re-execution (NewCAR) was introduced to reduce memory

conflicts and thus increase the MLP. Second, fetch stalls can block higher priority warps from making progress because they are waiting for the instructions. A simple next line prefetcher for the L1I cache was sufficient to reduce most of the fetch stalls. Evaluations show that *ELF* can improve the performance by 4.1% over the greedy-then-oldest (GTO) scheduler with only 1.39kB extra storage per SM. When used with other techniques like NewCAR and instruction prefetching, *ELF* can achieve total speedup of 11.9% over the GTO.

CHAPTER IV

Chimera: Collaborative Preemption for Multitasking on a Shared GPU

4.1 Introduction

Many modern computer systems are heterogeneous systems, where GPUs are attached to CPUs for throughput-oriented workloads (or kernels). These systems often have multiple CPUs that share a single GPU. When multiple CPUs offload data-parallel kernels simultaneously onto a shared GPU, multitasking must be supported. Recently, Nvidia's Kepler architecture [47] introduced the Hyper-Q feature to maintain multiple independent kernel queues to concurrently execute independent kernels on a shared GPU. However, this feature is limited to the kernels *within* a single process. Multi-Process Service (MPS) [52] achieves multitasking with a software solution, but is limited to MPI applications. With current generation GPUs, kernels have to wait until a previously running kernel finishes, if multiple processes are trying to share a GPU.

Traditionally, preemptive multitasking on CPUs has been achieved through context switching, which has a reasonable preemption latency and throughput overhead. How-

ever, supporting preemptive multitasking on GPUs through context switching can incur a higher overhead compared to CPUs, where the context of an SM can be as large as 256kB of register file and 48kB of on-chip scratch-pad memory [3, 55, 83]. Not only does a kernel have to endure a long preemption latency, the GPU also wastes execution resources while context switching. Although Tanasic et al. [83] has shown that the average normalized turnaround time can still be improved with high context switching overhead, such overhead wastes the GPU's computing power and may be ineffective for latency-sensitive applications [31, 32, 6].

To overcome these challenges, this chapter proposes *Chimera*, a collaborative preemption approach for GPUs that can precisely control the preemption overhead. *Chimera* can achieve a specified preemption latency while minimizing throughput overhead. Since GPUs consist of multiple SMs, a preemption request can have multiple solutions with diverse overheads by preempting different subsets of SMs with different preemption techniques. Given a preemption request, *Chimera* explores the possible solutions to minimize throughput overhead while conforming to the required preemption latency. *Chimera* achieves the goal by intelligently selecting *which* SMs to preempt and *how* each thread block will be preempted.

Chimera first introduces SM flushing, a GPU-specific preemption technique that is enhanced to exploit the semantics of thread blocks in the GPU programming model and the concept of idempotence to achieve low preemption latency. A kernel is *idempotent* if it generates the same result even if it is restarted in the middle of its execution [35, 14, 17, 43]. *Chimera* further relaxes the idempotence condition to enable flushing for more kernels. A thread block is defined to be *idempotent at the time of preemption* if it produces the same

result up to the preemption point even if it is restarted from the beginning. Thus, the context of a thread block can be safely dropped with the relaxed idempotence condition even if the kernel is non-idempotent. Because non-idempotent execution regions tend to be clustered at the end of execution in GPU kernels, relaxed idempotence is effective for increasing the opportunities for flushing.

In addition to flushing, *Chimera* has two other preemption techniques in its toolbox: context switching, and draining. Context switching [43, 83] stores the context of currently running thread blocks, and preempts an SM with a new kernel. Draining [31, 83] stops issuing new thread blocks to the SM and waits until the SM finishes its currently running thread blocks.

These three preemption techniques exhibit different tradeoffs between preemption latency and throughput overhead. Context switching has an almost constant mid-range preemption latency and throughput overhead. Draining has the least throughput overhead, but preemption latency can be long if the preemption happens near the beginning of the thread block execution. Flushing has almost zero preemption latency, but throughput overhead can be large if the preemption occurs near the end of thread block execution.

Chimera recognizes the different tradeoffs of these three preemption techniques and chooses which SMs to preempt and how each thread block will be preempted. *Chimera* estimates the costs of the three preemption techniques for the candidate SMs, and intelligently selects SMs and corresponding preemption technique by comparing the costs.

This chapter makes the following contributions:

- This chapter introduces SM flushing, a GPU-specific adaptation of a classic preemp-

tion technique that can instantly preempt an SM. This chapter combines the concept of idempotence with the semantics of thread blocks in the GPU programming model to enable efficient SM flushing.

- This chapter shows that relaxing the idempotence condition is essential for SM flushing to achieve its promised preemption latency. Detecting the relaxed idempotence condition can be done in software.

- This chapter shows that the three available preemption techniques for GPUs, namely context switching, draining, and flushing, make different tradeoffs between preemption latency and throughput overhead. Moreover, these tradeoffs change as a thread block makes execution progress on an SM.

- This chapter proposes *Chimera*, a collaborative preemption approach for a shared GPU that achieves a specified preemption latency while minimizing throughput overhead. *Chimera* recognizes tradeoffs of available preemption techniques, and makes an intelligent decision as to which SMs to preempt and how to preempt each thread block.

4.2 Motivation

This section introduces the three preemption techniques used in *Chimera*. This section also motivates the need for collaboration among the preemption techniques.

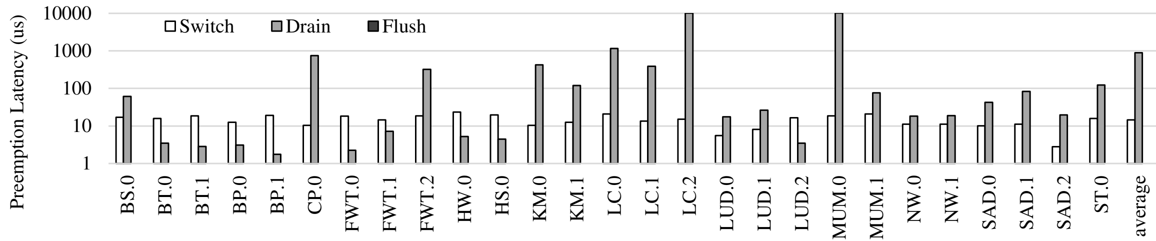


Figure 4.1: Estimated preemption latency for each preemption technique. For draining, a uniform random distribution on the preemption point across thread block execution is assumed. For flushing, zero preemption latency is assumed.

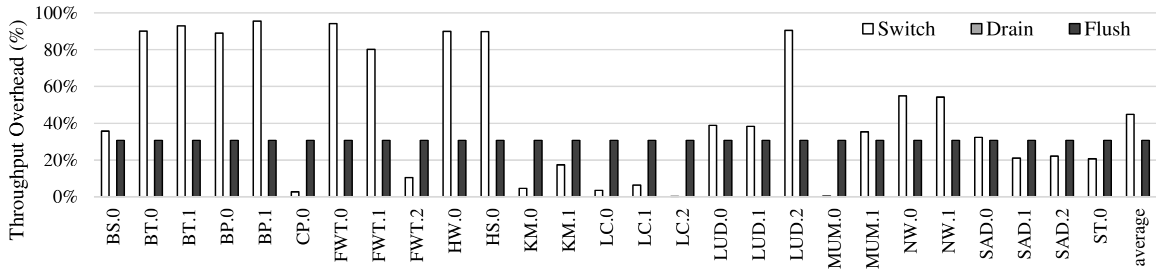


Figure 4.2: Estimated throughput overhead for each preemption technique when thread blocks running on an SM are assumed to be in sync. For flushing, a uniform random distribution on the preemption point across thread block execution is assumed.

4.2.1 Spatial Multitasking

As described in Chapter II, SMs do not share any states among themselves. Spatial multitasking [3] exploits this property to allow GPUs to run multiple kernels on different subsets of SMs. Preemptive multitasking can also exploit the same property by preempting only a subset of SMs to yield to a new kernel. Starvation can also be avoided by scheduling at least one SM to each available kernel.

4.2.2 Prior Preemption Techniques

Supporting preemptive multitasking incurs overheads in terms of latency and throughput. For example, context switching for preemption in CPUs involves saving the context

of the currently running process/thread, running the operating system (OS) scheduler to choose the next process/thread to run, and loading context of the selected process/thread. Preempting a process/thread experiences increased response time due to preemption latency. Also, system throughput is degraded because no progress is made during context switching. The overhead of context switching is proportional to the size of the context.

Modern GPUs can have up to 2048 threads concurrently running on a single SM [47]. Because each thread accesses its own registers, the context size for an SM can grow quickly. Moreover, each SM has its own on-chip scratch-pad memory, which is shared by the threads within a thread block. For modern GPUs, the context of a single SM can be as large as 256kB of register file and 48kB of shared memory [3, 55, 83]. With such a large context, preempting with context switching has high overhead in both preemption latency and wasted throughput.

To avoid the throughput overhead of context switching, SM draining [83] has been proposed. Because thread block executions are independent from each other in the GPU execution model, a thread block does not have to remember its context when it finishes execution. When an SM is preempted with draining, new thread blocks are no longer issued to that SM. When the SM finishes all the running thread blocks, the SM is preempted and can be assigned to another kernel. As the SM is continuously making progress during preemption, throughput overhead of draining is much less than that of context switching.

Draining, however, does not solve the preemption latency problem. Because the preemption latency of draining is dependent on the remaining execution time of thread blocks in the SM, it can be much higher than that of context switching.

4.2.3 SM Flushing

To enable low preemption latency, SM flushing is introduced, which further exploits the independence of thread block execution in the GPU execution model. Flushing drops an execution of a thread block without context saving and re-executes the dropped thread block from the beginning on another SM. Because thread block executions are independent, other thread blocks do not notice whether a thread block has been rerun from the beginning. Flushing reduces the preemption latency to almost zero. However, certain conditions have to be met to ensure the correctness of the thread block execution that was dropped and rerun from the beginning.

A GPU kernel is *idempotent* if it produces the same result regardless of the number of times it is executed [35, 14, 17, 43]. Because there is no interaction between thread blocks, idempotence conditions for a GPU kernel are much simpler than those in general CPU applications. To be idempotent, a kernel should not have any 1) atomic operations, and 2) overwrites to a global memory location that is read in the kernel. In the studied benchmarks, 12 out of 27 kernels were found to be idempotent. The idempotence conditions are listed in Table 4.2. Without enabling flushing in all the kernels, flushing loses its effectiveness because it cannot preempt non-idempotent kernels. The details of relaxing the idempotence conditions, and implementation of flushing are further discussed in Section 4.3.4.

4.2.4 Tradeoff

Context switching, draining, and flushing make different tradeoffs between preemption latency and throughput overhead. Figure 4.1 shows the estimated preemption latency for

each preemption technique. In the figure, the y-axis shows preemption latency on a logarithmic scale, and the x-axis shows the kernels in the benchmarks. If multiple kernels are launched in a benchmark, they are differentiated with numbers after the benchmark name and a dot. All the labels and numbers for benchmarks and kernels are listed in Table 4.2. To estimate the preemption latency of context switching, an SM is assumed to have only its share of global memory bandwidth to save its context. Context size can be calculated from the kernel's resource usage even before the kernel launch. The same method was used in [83] to project the estimated preemption latency for context switching. To estimate the preemption latency for draining, the average time to execute a thread block is first measured through simulation. Assuming uniform random distribution on the preemption point across the execution of a thread block, the preemption latency for draining can be calculated. The preemption latency for flushing idempotent kernels is assumed to be zero.

In Figure 4.1, context switching shows a relatively constant response time in the order of 10 μs, while draining exhibits diverse response time ranging from 0.8 μs to 10212.8 μs. On average, context switching, draining, and flushing require 14.5 μs, 830.4 μs, and 0 μs, respectively, to preempt an SM. They are equivalent to an order of 10,000, 500,000, and 0 cycles, respectively, in current generation GPUs.

Figure 4.2 shows the estimated throughput overhead for each preemption technique. In the figure, the y-axis shows the percentage of throughput overhead for each preemption technique compared to the throughput without preemption. Thread blocks running on an SM are assumed to be in sync. The throughput overhead of context switching is twice the preemption latency divided by the thread block execution time, where the preemption latency is doubled because throughput overhead comes both from context saving and context

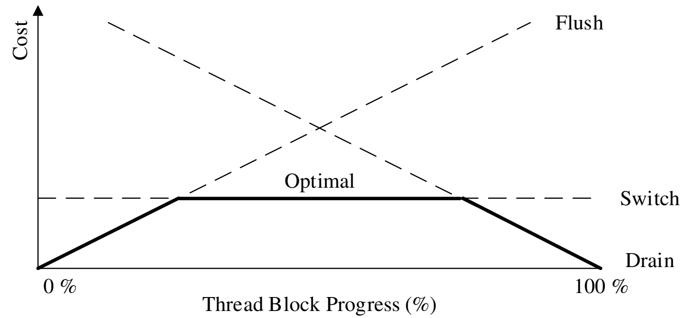


Figure 4.3: Theoretical cost of each preemption technique when preempting a thread block at a given amount of execution progress. Context switching has constant cost across the execution, draining has lower cost as a thread block is near the end of execution, and flushing has lower cost as a thread block is closer to the beginning of execution.

loading. SM draining is assumed to have zero throughput overhead because it continuously does useful work until the thread block finishes. In reality, thread blocks can be out of sync, which will cause draining to incur some throughput overhead. To estimate the throughput overhead of flushing, a uniform random distribution on the preemption point across the execution of a thread block is again assumed. The throughput overhead of flushing is independent of the kernel, and is constant across all the benchmarks. Overall, context switching, draining, and flushing have throughput overhead of 47.7%, 0%, and 30.7%, respectively.

4.2.5 Collaborative Preemption

Different tradeoffs from the three preemption techniques encourage using different preemption techniques for each kernel. In fact, different tradeoffs can be further exploited by using different preemption techniques within one preemption request. Because a preemption request would typically want multiple SMs at the same time, each SM can be preempted with a different preemption technique. Moreover, each thread block in the SM can be preempted with a different preemption technique.

Figure 4.3 depicts the theoretical cost of each preemption technique if a thread block at a given progress is preempted. The cost can be thought of as an aggregate measure of preemption latency and throughput overhead. The cost of context switching is dependent on the context size and the available bandwidth for an SM, which is almost constant across thread block execution. The cost of draining, which is primarily preemption latency, is dependent on the remaining execution time of a thread block. It decreases toward the end of the thread block progress. The cost of flushing, on the other hand, is primarily throughput overhead, which is dependent on the work thrown away by flushing. More work is thrown away as the thread block progresses; hence, the cost increases accordingly.

The figure motivates to preempt with flushing if a thread block is in the early stage of execution, with context switching if a thread block is in the middle stage of execution, and with draining if a thread block is near the end of execution. The exact points at which to switch the preemption decision is based on the cost estimation of each preemption technique.

4.3 Architecture

Chimera is a collaborative preemption with three individual techniques: context switching, draining, and flushing. Context switching is implemented with a software trap routine. Draining is performed by adding logic in a thread block scheduler that stops issuing new thread blocks. Flushing requires reset logic in SMs, which clears all the states and in-flight instructions in the SM. For context switching and flushing, an SM has to send the stopped thread blocks' IDs back to the thread block scheduler so that they can be re-issued to the

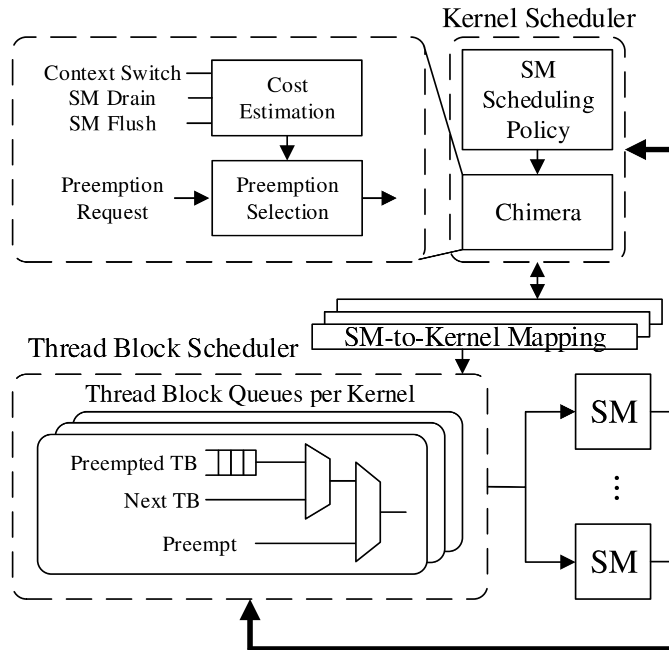


Figure 4.4: GPU scheduler with preemptive multitasking. The scheduler is two-level: the kernel scheduler assigns SMs to each kernel that may involve preemption decisions, and the thread block scheduler executes the decision by dispatching or preempting thread blocks from each SM. SMs can feedback the schedulers when an event that can change the scheduling decision occurs.

other SMs.

Chimera decides which SMs to preempt and which preemption technique to use for each thread block in the SMs, given the number of SMs to preempt. *Chimera* makes the decision based on the upper bound for preemption latency given by the preempting application or kernel. *Chimera* first estimates preemption latency and throughput overhead for each thread block in an SM when it is preempted with each preemption technique. *Chimera* chooses preemption techniques with the least throughput overhead that satisfy the given preemption latency for an SM, which can give the total cost of preemption for each SM. With the calculated costs, *Chimera* selects SMs which can minimize throughput overhead while meeting the required preemption latency.

4.3.1 GPU Scheduler with Preemptive Multitasking

Figure 4.4 illustrates the GPU scheduler with preemptive multitasking when multiple GPU kernels are running concurrently. The scheduler is a two-level scheduler: the kernel scheduler assigns kernels to each SM, which may involve preemption decisions, and the thread block scheduler carries out the decision. The thread block scheduler dispatches a new thread block to an SM, or preempts an SM with the given preemption techniques based on the decisions from the kernel scheduler. The kernel scheduler is a part of an operating system that manages a GPU device, while the thread block scheduler is a hardware module in a GPU, which is an extension of GigaThread engine in Fermi [46] with preemption support.

An SM partitioning policy in the kernel scheduler tells how many SMs each kernel will run on. The policy is orthogonal to the preemption decisions. It may be dependent on a characteristic of a kernel [3] or a priority of a kernel [83]. *Chimera* in the kernel scheduler achieves an SM partitioning policy by making preemption decisions. The kernel scheduler communicates to the thread block scheduler through SM-to-kernel mapping information which contains per-SM information about which kernel to schedule, whether preemption is necessary or not, and which preemption technique to use. The thread block scheduler always prefers to schedule the preempted thread blocks first so that the size of the preempted thread block queue can be limited.

Chimera consists of two parts: estimating costs of preemption for each technique, and selecting SMs to preempt with corresponding preempting techniques. To estimate the costs, *Chimera* gathers statistics for SMs. The statistics are measured using hardware and re-

ported directly to *Chimera*. Based on these statistics, *Chimera* estimates preemption latency in cycles, and throughput in the number of instructions rather than IPC. *Chimera* can directly compare the estimated cost of each preemption technique because they are calculated in the same units.

4.3.2 Cost Estimation

To distinguish different tradeoffs between different preemption techniques, *Chimera* has to estimate the cost of each preemption technique precisely for each SM. First, *Chimera* measures the total number of executed instructions for each thread block to determine the progress of each thread block. Note that instructions are counted not in thread granularity, but in warp granularity so that control divergence in a warp has minimal effect on the total executed instructions. Second, *Chimera* also measures the progress of each thread block in cycles. *Chimera* can calculate the average instructions-per-cycle (IPC) or cycles-per-instruction (CPI) of a thread block with these two statistics.

The preemption latency of context switching is estimated using the same method as detailed in Section 4.2.4. To estimate the throughput overhead of context switching, the average IPC of the preempted kernel on a single SM is multiplied by twice the preemption latency of context switching. Note that preemption latency is doubled because throughput overhead not only comes from context saving, but also from context loading. The preemption latency of draining is estimated by multiplying the remaining instructions to execute in a thread block by the average CPI of the preempted kernel. The average execution cycles per thread block is not used directly because it has much larger variance compared to the average executed instructions, leading to less accurate estimations. The throughput

Algorithm 3 Preemption Selection

```
    Input: LatLimit, Kernel, NumPreempts                    ▷ From SM Scheduling Policy
    Output: SM_Preemptions[1..NumPreempts]
 1: for each SM in Kernel do
 2:     for each TB in the SM do
 3:         for each Preemption Technique do
 4:             TBCosts.push(EstimateCost(TB, Technique))
 5:         end for
 6:     end for
 7:     TBSorted = SortByThroughputOverhead(TBCosts)
 8:     while !TBSorted.empty() do
 9:         TBCandidate = TBSorted.pop()
10:         if meets_latency(TBCandidate) and TBCandidate.TB not in SingleSMCost then
11:             SingleSMCost.add(TBCandidate)
12:         end if
13:     end while
14:     for each TB not in SingleSMCost do
15:         SingleSMCost.add(EstimateCost(TB, Switch))
16:     end for
17:     SMCosts.push(SingleSMCost)
18: end for
19: SMSorted = SortByThroughputOverhead(SMCosts)
20: for i = 1 to NumPreempts do
21:     while !SMSorted.empty() do
22:         SMCandidate = SMSorted.pop()
23:         if meets_latency(SMCandidate) then
24:             SM_Preemptions[i] = SMCandidate
25:             break
26:         end if
27:     end while
28: end for
29: return SM_Preemptions[1..NumPreempts]
```

overhead of draining is estimated by summing the difference between the number of executed instructions for each thread block and the maximum number of executed instructions among them. Flushing is always assumed to have zero preemption latency. The total number of executed instructions for thread blocks in the SM is used to estimate the throughput overhead of flushing. When the cost cannot be estimated due to the lack of gathered statistics, the maximum value is conservatively used as the estimated cost to avoid selecting affected techniques.

4.3.3 Preemption Selection

Chimera is a collaborative preemption that achieves low overhead multitasking through multiple preemption techniques with different overheads. SM scheduling policy, which is independent of these decisions, provides *Chimera* a preemption latency constraint, a kernel to preempt, and the number of SMs to preempt. Given the inputs, *Chimera* generates combinations of which SM to preempt and how to preempt, while satisfying the latency constraint.

Algorithm 3 illustrates how *Chimera* selects a subset of SMs and techniques to preempt. The algorithm starts by estimating the cost of each preemption technique for the thread blocks in an SM (line 2-6). The costs for the thread blocks are sorted by throughput overhead (at line 7), and a preemption technique for a particular thread block is selected if the preemption latency constraint is met and it is not already selected with another preemption technique (line 8-13). If a thread block cannot meet the constraint with any preemption technique, *Chimera* performs context switching for the thread block (line 14-16). Now, *Chimera* knows the preemption costs for each SM that is running the given kernel. It sorts all the costs by throughput overhead (at line 19). From the list of sorted candidates, *Chimera* finalizes the preemption selection (line 20-28). When finalizing the decision, *Chimera* checks whether the candidate satisfies the preemption latency constraint (at line 23). Because only one candidate exists for an SM, *Chimera* does not have to check whether the candidate SM is already selected.

The time complexity of Algorithm 3 is $O(NTlogT + NlogN)$, where N is the number of SMs that a kernel to preempt is occupying, and T is the number of available thread blocks

in an SM. The first term comes from the first loop (line 1-18), where preemption techniques are selected for particular thread blocks. Two loops (line 2-6 and line 8-13) take the linear time complexity of $O(PT)$, where P is the number of preemption techniques. In *Chimera*, P is a maximum of 3. The third loop (line 14-16) only takes the linear time complexity of $O(T)$. Therefore, sorting (at line 7) defines the time complexity with $O(PTlogPT) = O(TlogT)$. Since the outer loop runs N times, the entire loop (line 1-18) takes $O(NTlogT)$. The second term, which is $O(NlogN)$, comes from sorting for SMs (at line 19). The last loop (line 20-28) only takes the linear time complexity of $O(N)$. In general, N is in the order of 10 for current GPU generations. Furthermore, N will be reduced as more kernels run concurrently on the GPU because each kernel is likely to occupy a lower number of SMs. Also, T is a fixed number (maximum of 16 in Kepler [47]), but is typically less than the maximum (8 is the largest number of thread blocks per SM for simulated benchmarks in Table 4.2). Thus, the impact of the selection algorithm in *Chimera* is negligible to the preemption latency.

4.3.4 SM Flushing

SM flushing can be effective if it can preempt all kernels, whether they are idempotent or not. Flushing may still violate the required preemption latency if it cannot preempt an SM due to non-idempotence. Implementation of flushing is fairly straightforward as an SM already has a circuit that resets or clears itself.

The idempotence condition can be relexed further by looking at thread blocks individually with the notion of time. A GPU thread block is *idempotent at a given time* if it neither 1) has executed any atomic operations yet, nor 2) has overwritten a global memory location

System	Parameters
SM	30 SMs, 1400 MHz, 8 SIMT width 32768 registers per SM 8 maximum thread blocks per SM 48 kB shared memory
Memory Subsystem	6 memory partitions 177.4 GB/s bandwidth

Table 4.1: System configuration

that is read by the thread block. Because atomic operations or global memory overwrites tend to be performed at the end of a thread block execution, a thread block can remain idempotent for most of its execution time even if the kernel itself is non-idempotent.

With the relaxed idempotence condition, SMs have to notify the GPU scheduler when thread blocks have progressed beyond the non-idempotent point. The notification is implemented in software by inserting a store instruction in front of atomic operations or global overwrite operations. The store is made to a pre-defined address, which is non-cacheable. SMs will prepend their ID to the store so that the store address is unique for each SM. As SMs are in-order cores, these inserted stores are guaranteed to take place before the atomic or global overwrite operations. The GPU scheduler looks at these pre-defined addresses to check whether each SM can be preempted with flushing.

As atomic operations are separate hardware instructions, they are trivial to find. Global overwrite operations are found by compiler analysis to distinguish between global writes and global overwrites. While pointer alias analysis is undecidable [24] in general, pointers are used in a more restricted fashion in GPU kernels, which allows the compiler to find global overwrites precisely in most cases.

Benchmark (Label) [Input]	Source	Kernel (Label)	Average Drain Time	Context /TB	TBs /SM	Switching Time	IDPT (Region)
BlackScholes (BS) [4M Options]	Nvidia SDK [51]	BlackScholesGPU (0)	60.9 μs	24 kB	4	17.0 μs	Yes
B+ Tree (BT) [1M Nodes]	Rodinia [7]	findRangeK (0)	3.5 μs	46 kB	2	15.9 μs	No (47%)
		findK (1)	2.8 μs	36 kB	3	18.7 μs	No (46%)
Back Propagation (BP) [128K Nodes]	Rodinia [7]	bpnn_layerforward (0)	3.1 μs	12 kB	6	12.5 μs	No (89%)
		bpnn_adjust_weights (1)	1.8 μs	22 kB	5	19.0 μs	No (52%)
Coulombic Potential (CP) [2K on 256x256 Grid]	Parboil [82]	cenergy (0)	746.9 μs	7 kB	8	10.4 μs	No (63%)
Fast Walsh Transform (FWT) [8M]	Nvidia SDK [51]	fwtBatch2Kernel (0)	2.3 μs	21 kB	5	18.2 μs	No (52%)
		fwtBatch1Kernel (1)	7.2 μs	28 kB	3	14.5 μs	No (90%)
		modulateKernel (2)	321.8 μs	18 kB	6	18.7 μs	Yes
Heart Wall Tracking (HW) [656x744 Pixels/Frame]	Rodinia [7]	kernel (0)	5.2 μs	67 kB	2	23.4 μs	No (90%)
HotSpot (HS) [1024x1024 Data Points]	Rodinia [7]	calculate_temp (0)	4.5 μs	38 kB	3	19.7 μs	Yes
Kmeans (KM) [0.5M Data, 34 Features]	Rodinia [7]	invert_mapping (0)	424.3 μs	10 kB	6	10.4 μs	Yes
		kmeansPoint (1)	118.8 μs	12 kB	6	12.5 μs	Yes
Leukocyte Tracking (LC) [640x480 Pixels/Frame]	Rodinia [7]	GICOV_kernel (0)	1162.0 μs	17 kB	7	20.9 μs	Yes
		dilate_kernel (1)	391.7 μs	9 kB	8	13.5 μs	Yes
		IMGVF_kernel (2)	10173.2 μs	87 kB	1	15.2 μs	No (99%)
LU Decomposition (LUD) [512x512 Data Points]	Rodinia [7]	lud_diagonal (0)	17.4 μs	4 kB	8	5.6 μs	No (97%)
		lud_perimeter (1)	26.2 μs	5 kB	8	8.1 μs	No (94%)
		lud_internal (2)	3.5 μs	16 kB	6	16.6 μs	No (80%)
MUMmer (MUM) [50000 25-char. Queries]	Rodinia [7]	mummergpuKernel (0)	10212.8 μs	18 kB	6	18.7 μs	Yes
		printKernel (1)	76.4 μs	24 kB	5	20.8 μs	Yes
Needleman-Wunsch (NW) [4096x4096 Data Points]	Rodinia [7]	needle_cuda_shared_1 (0)	18.2 μs	8 kB	8	11.1 μs	No (96%)
		needle_cuda_shared_2 (1)	18.7 μs	8 kB	8	11.1 μs	No (96%)
SAD [1920x1072 Pixels]	Parboil [82]	mb_sad_calc (0)	42.3 μs	7 kB	8	10.1 μs	Yes
		larger_sad_calc_8 (1)	82.9 μs	8 kB	8	11.1 μs	Yes
		larger_sad_calc_16 (2)	19.7 μs	2 kB	8	2.8 μs	Yes
Stencil (ST) [512x512x64 Grid]	Parboil [82]	block2D_hybrid_coarsen_x (0)	122.3 μs	11 kB	8	15.9 μs	Yes

Table 4.2: Benchmark Specification. IDPT denotes idempotence, and the percentage of idempotent region in a thread block execution for non-idempotent kernels is also shown.

4.4 Results

The GPGPU-Sim v3.2.2 [5] is extended to evaluate *Chimera*. GPGPU-Sim only models the GPU, while the host code runs natively on CPUs. GPGPU-Sim does not model the overhead of data transfers between the CPU and GPU either. A Fermi [46] architecture with 30 SMs is modeled. All the system configuration parameters are summarized in Table 4.1. Context switching is implemented by halting an SM for the estimated context switch time instead of using a software trap routine. The result for context switching will be rather optimistic in the sense that the memory bandwidth consumed by context switching will affect

other SMs to slow down in reality and vice versa, whereas the evaluated implementation does not account for the effect.

For all the preemption techniques, the same SM scheduling policy is used, which is similar to the mix of **Smart Even** and **Rounds** in spatial multitasking [3]. SMs are distributed evenly across the kernels except when the kernel requires less SMs than the even split. A kernel can request less SMs than the even split for two reasons: if a kernel is size-bound, where the grid size or the number of thread blocks for the kernel cannot fully occupy its portion of SMs at its launch, or if the remaining number of thread blocks is insufficient to fully occupy the given number of SMs near the end of execution.

A wide range of GPGPU applications from Nvidia Computing SDK [51], Rodinia [7], and Parboil [82] benchmark suite are evaluated. Table 4.2 lists all the evaluated benchmarks, their inputs, and their kernels with their characteristics. The estimated average drain time, the size of the context for one thread block, the maximal number of concurrent thread blocks per SM, the estimated context switch time, and idempotence of the kernel are shown as well. The importance of relaxed idempotence condition in flushing is partly shown by the percentage of idempotent region in a thread block execution for non-idempotent kernels. Note that idempotent kernels have 100% idempotent region, and are not shown in the table. A subset of benchmarks from each benchmark suite with diverse characteristics in terms of the context switching time, draining time, and the idempotence condition were selected for evaluation.

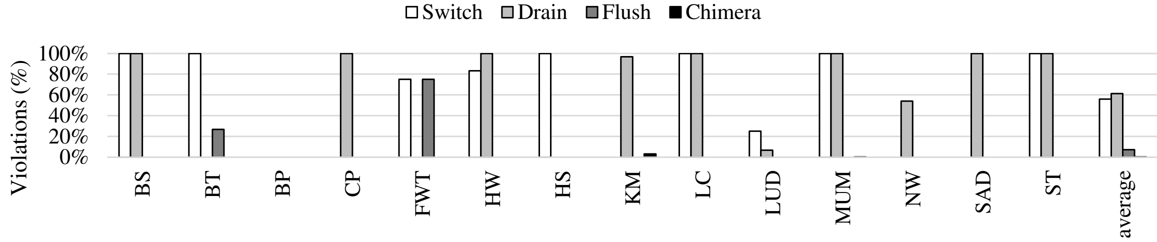


Figure 4.5: The percentage of preemptions that violate the deadline of a periodic, real-time task when GPGPU benchmarks are run together. The preemption latency constraint is 15 μs.

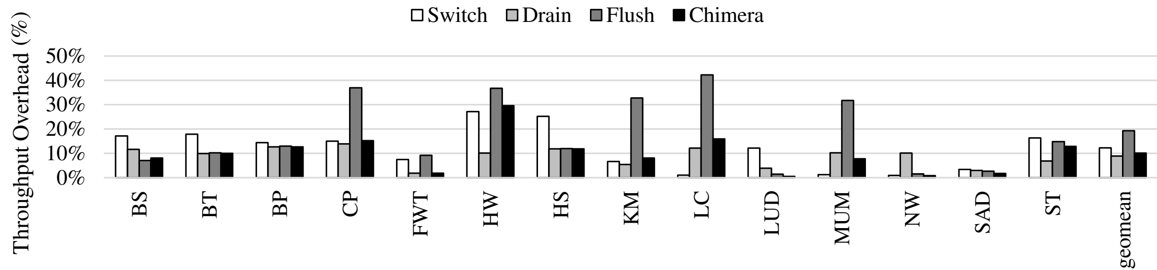


Figure 4.6: Throughput overhead of each preemption technique when GPGPU benchmarks are run with a periodic, real-time task. The preemption latency constraint is 15 μs. Effective throughput is used to avoid giving unfair advantage to the preemption techniques that frequently miss the deadline.

4.4.1 Periodic Task with Deadline

Each GPGPU benchmark is first concurrently run with a synthetic GPU benchmark, which mimics a periodic, real-time task that has a hard deadline. The synthetic GPU benchmark is launched every 1ms, preempting half of the SMs, and executed for 200μs. The deadline for the synthetic benchmark is the execution time plus the required preemption latency. The synthetic benchmark is killed if it misses the deadline. The simulation is run until a GPGPU benchmark executes 1 billion instructions or finishes its execution.

Figure 4.5 illustrates the percentage of preemptions that violate the deadline of the synthetic benchmark when the preemption latency constraint is set to 15μs. On average, context switching, draining, flushing, and *Chimera* miss the deadline for 56.0%, 61.3%,

7.3%, and 0.2% of preemptions, respectively. Flushing, despite its zero preemption latency, violates the deadline for BT and FWT because these benchmarks have small idempotent regions with short thread block execution time. In such cases, flushing is more likely to miss the deadline because thread blocks have higher chances to be in a non-idempotent region even with the relaxed idempotence condition. On the other hand, *Chimera* misses the dealine in 0.2% cases only. These misses are primarily due to the incorrect estimation of draining latency. However, the error is in the range of few hundred cycles ($< 1\mu$s), and can be avoided by providing a margin to preemption latency constraint against the deadline.

Figure 4.6 shows the overhead on throughput for each preemption technique in the same scenario. If the deadline of the synthetic benchmark is missed, the throughput additionally gained by running GPGPU benchmark more during that period is ignored so that the measured overhead is fair among the preemption techniques. Also, the throughput of the synthetic benchmark is neglected on purpose, to isolate throughput overhead of each preemption technique. Overall, context switching, draining, flushing, and *Chimera* have throughput overhead of 12.2%, 8.9%, 19.3%, and 10.1%, respectively. Draining does not have zero throughput overhead as explained in Section 4.2.4 because the assumption that thread blocks are running in sync is not true in practice. However, it still achieves the minimum throughput overhead compared to switching, and flushing. *Chimera* shows similar throughput overhead with significantly fewer deadline misses. FWT, LUD, and NW show least throughput overhead for *Chimera* because they include kernels that either have short execution time (SMs will be quickly freed) or occupy less than half of all available SMs from the beginning. In such case, preemption can be performed with low overhead since a new kernel can be launched to the idle SMs.

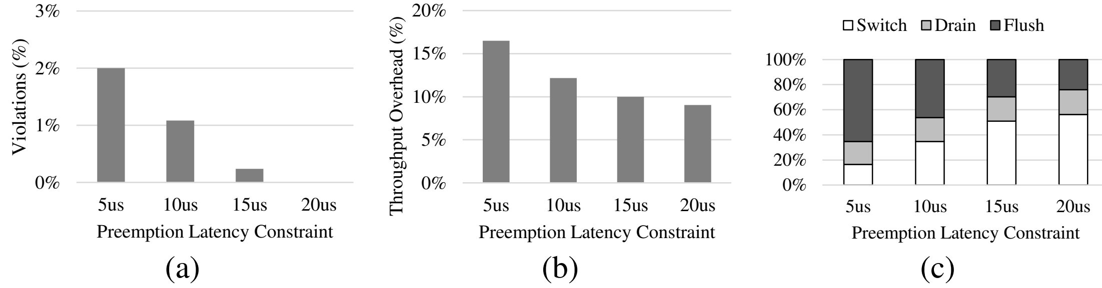


Figure 4.7: Impact of varying preemption latency constraint on (a) the percentage of deadline violations, (b) throughput overhead, and (c) distribution of each preemption technique used in *Chimera*.

As shown by the figures, *Chimera* can almost always meet the preemption latency constraint while individual preemption techniques cannot. *Chimera* achieves this goal while maintaining the low throughput overhead of draining. In fact, *Chimera* can have lower throughput overhead than individual preemption techniques, as in LUD, by collaboratively utilizing all the techniques.

4.4.2 Impact of Preemption Latency Constraint

Chimera is a collaborative preemption with controlled overhead. Figure 4.7 demonstrates the impact of the preemption latency constraint when it is varied from 5μs to 20μs. The same multi-programmed workloads are used as in Section 4.4.1.

Figure 4.7 (a) shows the percentage of preemptions that *Chimera* violates the deadline when the preemption latency constraint is varied from 5μs to 20μs. When the preemption latency is 5μs, 10μs, 15μs, and 20μs, violations happen for 2.00%, 1.08%, 0.24%, and 0.00% of preemptions, respectively. As explained with Figure 4.5, flushing may fail to meet the deadline if a kernel is non-idempotent, and has short thread block execution time. Since flushing is what *Chimera* relies on to achieve low preemption latency, *Chimera* also

suffers from the same problem when the preemption latency constraint is extremely low, as in the case of 5μs.

Figure 4.7 (b) shows throughput overhead of *Chimera* when the preemption latency constraint is varied from 5μs to 20μs. Again, effective throughput is used to measure throughput overhead to avoid giving unfair advantage to the preemption techniques that miss the deadline. *Chimera* has 16.5%, 12.2%, 10.0%, and 9.0% throughput overhead when the preemption latency constraint is 5μs, 10μs, 15μs, and 20μs, respectively. As shown in the figure, *Chimera* can reduce more throughput overhead when the preemption latency constraint is increased. If only one preemption technique is utilized, the loose deadline cannot be exploited.

Figure 4.7 (c) shows the distribution of each preemption technique used in *Chimera* when the preemption latency constraint is varied from 5μs to 20μs. As the preemption latency constraint is reduced, *Chimera* exploits flushing more because flushing is the only preemption technique that provides low preemption latency at the expense of throughput overhead. About 19% of preemptions constantly utilize draining because thread blocks near the end of their execution always exist even for low preemption latency constraints. Context switching has constant preemption latency regardless of the constraint, hence, its utilization quickly drops as the preemption latency constraint is reduced.

4.4.3 Relaxed Idempotence Condition in SM Flushing

Figure 4.8 illustrates the effectiveness of relaxing the idempotence condition for flushing. Strict refers to the original idempotence condition, and relaxed refers to the relaxed idempotence condition. The percentage of preemptions that violate a 15μs preemption la-

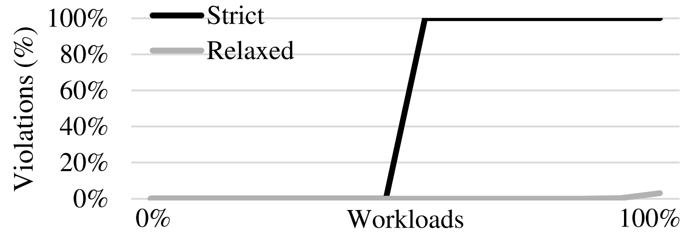


Figure 4.8: The percentage of preemptions that violate a 15μs preemption latency constraint when SM flushing uses strict or relaxed idempotence condition.

tency constraint is shown for all the workloads used in Section 4.4.1. On average, flushing violates the deadline for 50.0% and 0.2% of the total preemptions with strict and relaxed idempotence condition, respectively.

When a strict idempotence condition is used, the kernel idempotence decides whether an SM can be preempted with flushing or not. With the relaxed idempotence condition, thread blocks in such kernel can still be preempted with flushing if they have not reached the non-idempotent region. Without the relaxed idempotence condition, flushing cannot achieve its promised low preemption latency because non-idempotent kernels cannot be preempted. The violations for the strict idempotence condition will be the same regardless of the preemption latency constraint. The results show that it is mandatory for flushing to have the relaxed idempotence condition to provide instant preemption.

4.4.4 Case Study

This section further investigates *Chimera* when a combination of GPGPU benchmarks without hard deadline is concurrently executed. Each multi-programmed workload is a combination of LUD with one of the benchmarks in Table 4.2. LUD is chosen because it launches multiple kernels that require different number of SMs, which results in numerous

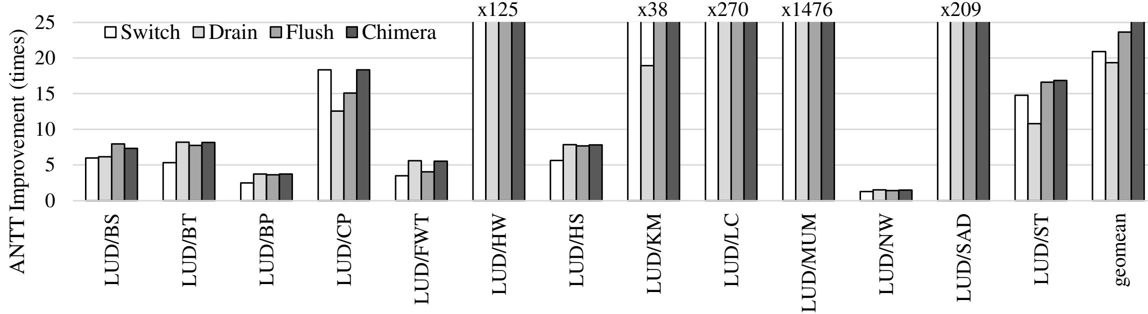


Figure 4.9: ANTT improvement over the non-preemptive FCFS when LUD is concurrently executed with another benchmark.

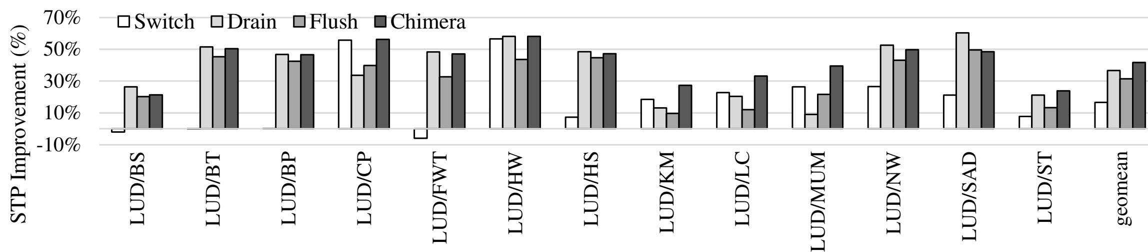


Figure 4.10: STP improvement over the non-preemptive FCFS when LUD is concurrently executed with another benchmark.

preemption requests. GPGPU benchmarks do not have hard deadline hence preemption latency constraint is chosen to be 30μs, which is the maximum possible preemption latency for context switching in the current configuration.

Each simulation runs until *all* benchmarks either execute 1 billion instructions or finish its execution. Among the benchmarks, FWT, HW, KM, LC, MUM, SAD, and ST run more than 1 billion instructions. When one benchmark finishes earlier than the others, it is restarted from the beginning to prohibit the last remaining benchmark from running on its own. The reported results are gathered only for the first 1 billion instructions or first execution whichever is reached first. All the benchmarks are started simultaneously at the beginning. This is a typical setting for simulating multi-programmed workloads [84, 66, 87, 3, 83]. For the baseline, non-preemptive scheduling is used, where each kernel has

to wait until previous kernel has finished its execution. Kernels are launched following first-come, first-serve (FCFS) policy.

To compare the performance of preemption techniques, the metrics suggested by Eyerman et al. [16] are used. Average normalized turnaround time (ANTT) quantifies the user-perceived slowdown due to multitasking using Equation 5.1, where N denotes the number of kernels, CPI_i^{multi} is the CPI when a kernel is executed in the multi-programmed workload, and CPI_i^{single} is the CPI when the kernel is executed on its own. System throughput (STP) measures the progress of the system under multitasking using Equation 5.2, where parameters are the same as in ANTT.

$$ANTT = \frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i^{multi}}{CPI_i^{single}} \tag{4.1}$$

$$STP = \sum_{i=1}^{N}\frac{CPI_i^{single}}{CPI_i^{multi}} \tag{4.2}$$

Figure 4.9 presents the ANTT improvement of each preemption technique over the non-preemptive FCFS. On average, context switch, draining, flushing, and *Chimera* improves the ANTT by 20.9x, 19.3x, 23.6x, and 25.4x, respectively. HW, KM, LC, MUM, and SAD have a kernel, whose execution time is extremely long. Preemptive multitasking improves ANTT drastically over non-preemptive FCFS because non-preemptive FCFS has to wait until these kernels finish their execution. Among single preemption techniques, flushing has the most ANTT improvement as it has the least preemption latency. *Chimera* can improve the ANTT more than flushing because it can preempt non-idempotent thread blocks using other preemption techniques.

Figure 4.10 shows the STP improvement of each preemption technique over the non-preemptive FCFS. Overall, context switch, draining, flushing, and *Chimera* improves the STP by 16.5%, 36.6%, 31.4%, and 41.7%, respectively. Since LUD does not occupy all the available SMs, STP is significantly improved despite the throughput overhead of preemption techniques. Here, spatial multitasking is effectively improving STP. In LUD/SAD, *Chimera* improves STP much less than the top single preemption technique, which is draining. *Chimera* shows such behavior when the cost estimation is not accurate enough for context switching, and draining. Cost estimation is not precise if thread blocks have large variations in the execution time or in the CPI. Again, *Chimera* achieves the most STP improvement over any single preemption technique because *Chimera* can choose alternative low overhead preemption technique if one preemption technique incurs larger overhead compared to the others while a single preemption technique is forced to endure the overhead. There is larger difference in STP compared to throughput overhead in the Section 4.4.1 because preemptions occur more frequently than 1ms interval in the simulated multi-programmed workloads.

Chimera can improve ANTT and STP for GPGPU benchmarks without hard deadlines as it utilizes adequate preemption technique when preemption request occurs. When all the combinations of GPGPU benchmarks are used, *Chimera* improves ANTT and STP by 5.5x, and 12.2%, respectively, on average. Other combinations of GPGPU benchmarks have smaller number of preemption requests, which results in smaller improvement compared to LUD combinations.

4.5 Related Work to *Chimera*

Multitasking on GPUs is receiving a lot of attention from the research community as GPUs are becoming common in modern computer systems. First, recent works on enabling multitasking on GPUs are listed. Next, previous studies on reducing the overhead of context switching on CPUs are presented. Lastly, prior research that exploit the notion of idempotence are discussed.

Multitasking on GPUs: First attempts on GPU multitasking have been made on top of current GPUs by providing an illusion of single process to GPU or using cooperative multitasking. Context funneling [89] merges GPU contexts of multiple processes into a shared GPU context so that they can run concurrently on a single GPU. KernelMerge [22] makes GPUs only see a single scheduler kernel instead of individual independent kernels. Ino et al. [26] used cooperative multitasking to allow the concurrent execution of scientific and graphics applications on GPUs.

Some of the recent works study the scheduling policy when multitasking is enabled. Elastic kernels [55] transform kernels to enable fine grain control over the resource usage of kernels so that they can utilize SMs more efficiently. They study their scheme with multitasking by timeslicing a kernel to launch only a range of thread blocks at a time. Lee et al. [40] studies the interaction between thread block scheduling and warp scheduling. They also propose to run multiple kernels on the same SM, but do not present any detailed implementation.

Several works have paid attention to the independence of thread block execution. RGEM [31] splits memory transfers to GPU into smaller chunks so that they can be preempted at the

chunk boundary. PKM [6] partitions a kernel into subkernels, where each subkernel executes a subset of thread blocks. SM draining [83] stops issuing a thread block and waits until all the running thread blocks are finished. Independence of thread block execution is a unique property of GPUs and creates opportunities for efficient preemption specific to GPUs. *Chimera* also utilizes the independence of thread block execution to enable SM flushing.

Spatial multitasking [3] observes that kernels may not fully occupy all the available SMs and shows that kernels can run on different subset of SMs. However, spatial multitasking still requires preemption if one kernel wants to dynamically take SM that is already running another kernel. Tanasic et al. [83] implement context switching to show that it still improves ANTT, however, they do not solve the problems of long preemption latency and large throughput overhead.

Chimera can control the overhead in preemptive multitasking with collaborative preemption. *Chimera* is the only solution so far that can meet a given preemption latency with minimized throughput overhead.

Context switching: Reducing the overhead of context switching has been researched for CPUs as well. One approach is finding fast context switch points, where there are only few live registers so that the amount of context to be stored can be reduced [80, 96]. But they either achieve small amount of gain or require code specific to each switch point for context switching. Another approach is to mark registers with additional bits to annotate whether corresponding register should be stored during the context switch [54]. With tens of thousands of registers in GPUs, the extra storage overhead is not acceptable. ASTI [77] statically sets context switching points during compile time, thus needs to know which

applications will be running concurrently in advance.

All of these studies are limited in their applicability, and cannot be directly used in GPUs for low overhead context switching. In *Chimera*, context switching collaborates with preemption techniques that are specialized for GPUs to achieve low overhead preemption.

Idempotence: Idempotence has been primarily exploited to reduce the overhead of checkpointing in hardware. Reference idempotency [35] optimizes speculative execution by not tracking idempotent references, thus reduces speculative storage. Idempotent processor [14] and iGPU [43] implement low overhead exception support for CPUs and GPUs, respectively. They reconstruct a consistent program state for precise exception by re-executing from the beginning of idempotent region to the point of exception. Relax [13] and Encore [17] recover from soft errors with low overhead by selectively rerunning the idempotent regions rather than checkpointing all the states.

Chimera shares the idea of idempotence and is the first solution to try the notion of idempotence to eliminate preemption latency on GPUs with flushing. Moreover, flushing can be implemented with minimal overhead because the flush logic already exists, and relaxed idempotence condition is detected in software.

4.6 *Chimera* Conclusions

This chapter presented *Chimera*, a collaborative preemption approach on a shared GPU, that enables multitasking with controlled overhead. *Chimera* utilizes two GPU-specific preemption techniques called draining and flushing on top of traditional context switching. Draining exploits the independence of thread block execution to allow low throughput

overhead preemption. Flushing brings the concept of idempotent execution to preemption, which can be combined with the independence of thread block execution to enable low preemption latency. By intelligently selecting a subset of SMs to be preempted as well as the preemption techniques for thread blocks, *Chimera* can meet a given preemption latency constraint with minimal throughput overhead. Evaluations show that *Chimera* violates a $15\mu s$ preemption latency constraint for only 0.2% of the preemption requests. For multiprogrammed workloads, *Chimera* can improve the average normalized turnaround time by 5.5x, which can go up to 25.4x when a large number of preemption requests occur. System throughput can be improved by 12.2%, which can go up to 41.7% when a large number of preemption requests exist. *Chimera* demonstrates that preemptive multitasking on a shared GPU requires a different strategy from a traditional CPU, but is practical to implement.

CHAPTER V

Dynamic Resource Management for Efficient Utilization of Multitasking GPUs

5.1 Introduction

The single instruction multiple thread (SIMT) programming model used by CUDA [49] and OpenCL [34] unlocked the computing capability of GPUs for general-purpose applications. Many supercomputers in the TOP500 List [2] and the Green500 List [1] are already composed of GPUs due to their high performance for data-parallel applications and energy efficiency. GPUs are also readily available in cloud computing services like Amazon Web Services [4]. Supercomputers, cloud services, and data centers with GPUs will benefit significantly with shared GPUs because resource sharing is critical to efficient resource utilization in these environments [88].

To meet such demand, multitasking on GPUs began to receive a wide attention from both academia and industry. Earlier attempts [70, 32, 6] were made on the software level to provide a notion of fairness when multiple processes are trying to share a GPU. More recent studies from academia have explored the possibility of alternative preemption techniques

specific to the GPUs to enable low overhead multitasking [83, 59]. Industry is also moving in the similar direction. Although limited to the kernels within a single process, Hyper-Q [47] in Nvidia's Kepler architecture enables concurrent execution of independent kernels on a single GPU with multiple independent queues. Nvidia also introduced Multi-Process Service [52], which is a software support for MPI applications to multitask on a single GPU.

Spatial multitasking [3], which divides resources at the streaming multiprocessor (SM) granularity, was first studied for partitioning shared GPUs among multiple kernels. Recently, simultaneous multikernel (SMK) [90] or intra-SM slicing [93] have been proposed, which shares an SM between multiple kernels similar to simultaneous multithreading (SMT) on CPUs. However, neither is superior over the other because their performance depends on application mixes.

SMK performs better than spatial multitasking especially when kernels running on the same SM have different characteristics: (1) when resource requirements of kernels are different so that more threads can be launched on an SM with SMK, or (2) when mainly utilized functional units are different so that SMK can interleave instructions from the kernels with small contention. On the other hand, spatial multitasking has advantage over SMK when the co-running kernels interfere with each other significantly especially due to load-store units or L1 cache. In such case, running these kernels separately on different SMs can be more effective by avoiding contention. Note that the contention for computational units usually has less interference because they are abundant on the GPUs while load-store units and L1 cache are more scarce resources. Because spatial multitasking and SMK have their own advantages and disadvantages, a resource partition scheme for multi-

tasking GPUs should be able to exploit both.

Determining the best performing resource partition is further complicated due to the difficulty in predicting the performance, e.g., predicting the performance of single kernel execution with a regression model on GPUs [25, 94] has shown errors in the range of 10%. Moreover, multikernel execution entails more difficulties because of program phases in kernels and the interference between kernels. Without addressing these problems, the benefits from multitasking GPUs are limited.

To that end, this chapter proposes *GPU Maestro*, which addresses these questions to implement a dynamic resource management to efficiently utilize multitasking GPUs. A lightweight dynamic scheduling mechanism is first proposed, which utilizes a direct measurement of existing performance counters on GPUs rather than a model-based prediction with complex regression models. Because complex interactions between kernels are embedded in the direct measurement, *GPU Maestro* avoids the performance prediction errors from the model-based approaches, or when using single kernel execution profiles. The key idea of *GPU Maestro* is that GPUs are composed of multiple SMs, where each SM can be allocated differently and monitored for performance. In each epoch, *GPU Maestro* tests the performance of different configurations, whose results are used to find the best performing resource partition. The selected resource partition will be used in the next epoch, and a small subset of SMs are used to test different partitions again. After few epochs, *GPU Maestro* converges to the best performing resource partition for the given kernel combination.

There are two additional challenges for realizing SMK: (1) a resource fragmentation problem, where a thread block cannot be scheduled because resources are available in

small chunks although there are enough resources on an SM in total, and (3) a starvation problem from existing warp scheduling on SMK GPUs.

GPU Maestro solves the resource fragmentation problem on SMK GPUs using 2-way resource allocation, which forces thread blocks from the same kernel to allocate resources consecutively in the opposite directions similar to how a stack and heap grow in the opposite directions in a process's virtual memory for CPUs. *GPU Maestro* also imposes a fixed preemption priority order on the running thread blocks. With 2-way resource allocation, *GPU Maestro* can avoid resource fragmentation.

GPU Maestro finally studies the interaction between warp scheduling and SMK GPUs. *GPU Maestro* makes an observation that a newly launched kernel can have starvation periods as the greedy-then-oldest (GTO) warp scheduler favors the already running kernel. Various kernel-aware warp scheduling methods are studied to avoid starvation.

This chapter makes following contributions:

- This chapter shows that system performance of multitasking GPUs can vary depending on the application mixes. This chapter illustrates when SMK performs better than spatial multitasking, and vice versa. Furthermore, this chapter shows the difficulties in predicting multitasking performance because of dynamism within a kernel and interference between kernels.
- This chapter proposes *GPU Maestro*, which dynamically manages resource partition on multitasking GPUs to maximize the system performance. This chapter shows a lightweight implementation for *GPU Maestro*, which considers both dynamism and interference by monitoring existing performance counters for different allocations

with a subset of SMs.

- This chapter presents how the resource fragmentation problem can manifest on SMK GPUs. *GPU Maestro* solves resource fragmentation by using 2-way resource allocation, which restricts how a kernel allocates and releases resources when launching or preempting a thread block.

- This chapter studies the interaction between warp scheduling and SMK GPUs, and show that starvation can negatively impact the system. This chapter shows that kernel-aware warp scheduling is critical to avoiding starvation, and a simple round robin scheduling of kernels improves the system performance better than other complex scheduling methods.

5.2 Background

5.2.1 Multikernel Metrics

Throughout the chapter, the multikernel performance is measured using the metrics suggested by Eyerman et al. [16]: average normalized turnaround time (ANTT), and system throughput (STP). ANTT represents the user-perceived response time, while STP portrays the overall progress of the system. The following equations are used to calculate ANTT and STP:

$$ANTT = \frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i^{MK}}{CPI_i^{SK}} \tag{5.1}$$

$$STP = \sum_{i=1}^{N}\frac{CPI_i^{SK}}{CPI_i^{MK}} \tag{5.2}$$

	Fermi	**Kepler**	**Maxwell**	**Pascal**
Threads	1536	2048	2048	2048
Thread Blocks	8	16	32	32
Registers	128kB	256kB	256kB	256kB
Shared Memory	48kB	48kB	64kB	64kB
# of SMs (Tesla Model)	14 (M2050)	15 (K40)	24 (M40)	56 (P100)

Table 5.1: Resource trends in an SM on GPUs.

where N denotes the number of benchmarks, CPI_i^{MK} is the CPI when a benchmark is executed in the multi-programmed workload, and CPI_i^{SK} is the CPI when the benchmark is executed alone.

ANTT is a lower-is-better metric and STP is a higher-is-better metric. ANTT and STP may disagree on which multikernel scheduling is better because each metric reflects a separate aspect of the whole system.

5.3 Motivation and Challenges

This section shows the opportunity and motivation for dynamic resource management on multitasking GPUs. Two additional challenges in realizing SMK GPUs are also described.

5.3.1 Spatial vs. Simultaneous Multikernel

Table 5.1 shows the trends according to the four generations of Nvidia GPUs, where the resources on a single SM have increased as well as the number of SMs. This trend of having more concurrency supports within an SM and across SMs indicates that multitasking GPUs are promising. Concurrency across SMs favors spatial multitasking, while concur-

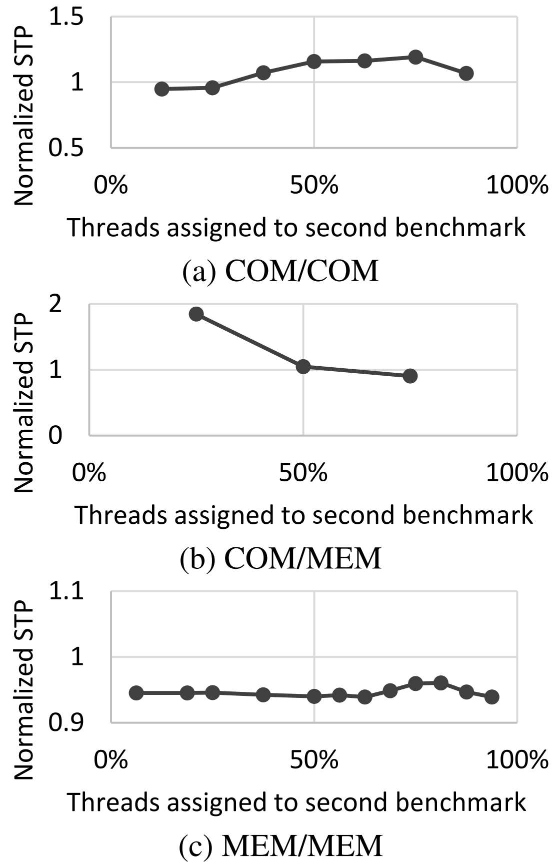


Figure 5.1: Normalized STP of SMK when fixed number of threads are assigned to each kernel within an SM for representative examples. STP of spatial multitasking is used as 1. (a) Even SMK is as good as the best performing SMK, (b) large performance gap exists between the best performing SMK and even SMK, and (c) spatial multitasking performs better than any SMK.

rency within an SM favors SMK. To fully understand the benefit of multitasking GPUs, the performance improvements from different resource partitions for varying application mixes are studied.

Figure 5.1 depicts the normalized STP of SMK on a GTX980 when fixed numbers of threads are assigned to each kernel within an SM. All the STP values are normalized to that of spatial multitasking. Three representative benchmark pairs are shown to discuss opportunities: LC/BP for COM/COM, HS/SC for COM/MEM, and LMD/ST for MEM/MEM. Details of benchmarks are listed in Table 5.3. Figure 5.1 (a) depicts a case where SMK

with even partitioning, which gives equal number of resources to each kernel resulting in 50% on the x-axis, performs better than or similar to other SMK resource partitions as well as spatial multitasking when both benchmarks are compute-intensive (COM). Figure 5.1 (b) illustrates a case where SMK with uneven partitioning benefits much larger than SMK with even partitioning and spatial multitasking. This commonly occurs if one of the benchmark is memory-intensive (MEM) and the other is COM. By giving more thread blocks to COM, they can utilize the idle cycles from MEM. Figure 5.1 (c) shows a case where spatial multitasking performs better than any SMK. This commonly happens if both benchmarks are MEM and the contention for the load/store unit or cache is high.

In general, SMK performs better than spatial multitasking when (1) SMK launches more threads, or (2) co-running kernels have different execution unit utilizations or small interference. For example, it is known that memory-intensive kernels cannot hide the memory latency even with thousands of threads because all the threads are likely to be waiting for the memory as they run the same code [60]. In such cases, SMK can issue instructions from compute-intensive kernels to improve compute resource utilization. On the other hand, spatial multitasking can provide better performance when the interference between kernels within an SM is large enough to degrade the system performance for SMK.

5.3.2 Multitasking GPU Performance

The system performance of a multitasking GPU is greatly impacted by how GPU resources are partitioned among kernels because spatial multitasking and SMK have their own advantages and disadvantages depending on the application mixes. A practical implementation of multitasking GPUs has to address how to find the best performing resource

partition exploiting both spatial multitasking and SMK. However, predicting multitasking performance is difficult especially for SMK because of the complex interactions between kernels.

5.3.3 Interference and Dynamism

To find the best performing resource partition, performance should be estimated for different resource partitions. A regression model is often used to predict the performance of a kernel [25, 94]. However, these models have errors in the range of 10%, which can be too large to provide meaningful estimations for SMK. Moreover, these models assume single kernel execution, which ignores the complex interaction when running multiple kernels on the same SM, and cannot capture the dynamism when doing SMK execution.

Figure 5.2 shows an IPC trace within an SM using a 50k instruction window. From 16M to 80M instruction interval, Figure 5.2 (Top) shows the IPC trace of SRAD and BFS when they are running together on an SM with SMK. Even thread allocation is assumed between SRAD and BFS. Figure 5.2 (Bottom) illustrates the IPC trace for BFS and SRAD when they are executed in isolation. To indicate interference, two sub-intervals are circled, where the IPC trace is substantially different between running alone, and with SMK. In the left circle, BFS has a program phase with high IPC when executed alone, which shows low IPC when running together with SRAD through SMK. On the right circle, the IPC of SRAD is less than half of independent execution due to memory and cache contention with BFS. As illustrated by the figure, SMK performance should not be estimated based on single kernel performance because interference can change the execution behavior significantly.

Figure 5.3 depicts a trace of STP improvement over non-shared execution for Oracle

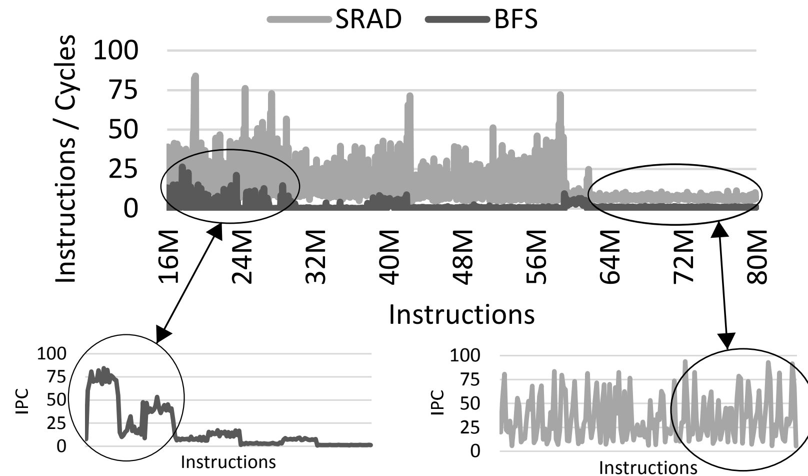


Figure 5.2: A trace of the instructions per cycle (IPC) within an SM using a 50k instruction window when running SRAD and BFS together with SMK (top), and when running them independently for the same interval (bottom). Circles indicate the same interval, where the IPC trace shows different behavior between running alone, and SMK.

and Even within an SM using 50k instruction window when running SRAD and BFS together with SMK. Oracle knows and chooses the best performing thread block partition for each instruction window, and instantly switches between different thread block partitions with zero overhead. Even always distributes threads evenly among SRAD and BFS. Oracle can improve STP by 30.0% over Even across the shown trace. Sub-intervals drawn on the figure illustrate a period when Oracle chooses different thread block partition from Even, which has both phases where more resources are assigned to either SRAD or BFS.

To capture interferences and dynamism, a dynamic scheduling framework that utilizes a direct measurement of the performance is necessary to maximize the performance benefit from SMK.

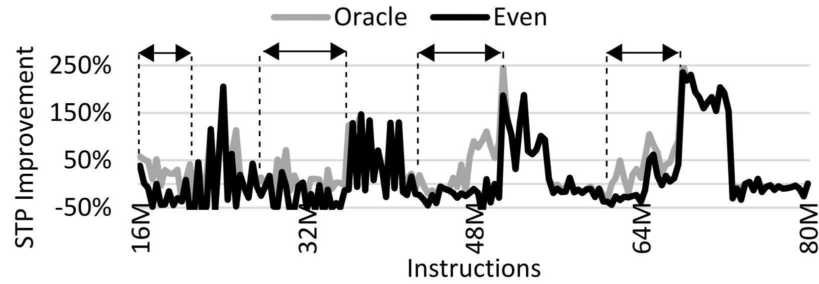


Figure 5.3: A trace of STP improvement over non-shared execution for Oracle and Even thread block partition within an SM using 50k instruction window when running SRAD and BFS together. Indicated sub-intervals are when Oracle chooses different partition from Even.

5.3.4 SMK Challenges

The implementation of SMK GPUs itself has two additional challenges to be addressed: (1) it has to solve resource fragmentation, which can become worse due to more preemptions with dynamic scheduling, and (2) it should avoid the starvation problem, which comes from the interaction between warp scheduling and multikernel execution.

5.3.4.1 Challenge 1: Resource Fragmentation

Shared memory and register file are shared among the thread blocks within an SM. Because shared memory is shared at a thread block granularity, each thread block allocates a consecutive region in the shared memory. Because thread blocks execute the same code to compute the shared memory address, physical address for shared memory has to be differentiated between thread blocks. The shared memory base register (SBR) contains the base shared memory address for each thread block. When a thread block accesses shared memory at runtime, the address is computed by adding the virtual address with the SBR. Similarly, consecutive registers are allocated to each warp. Note that a physical register

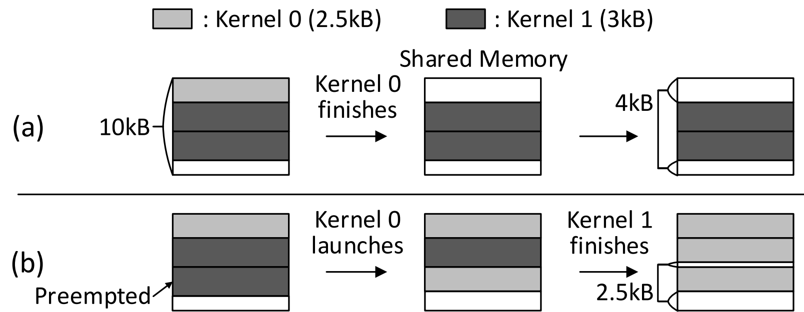


Figure 5.4: An illustration of resource fragmentation problem for shared memory when (a) a kernel terminates, and (b) a kernel is preempted.

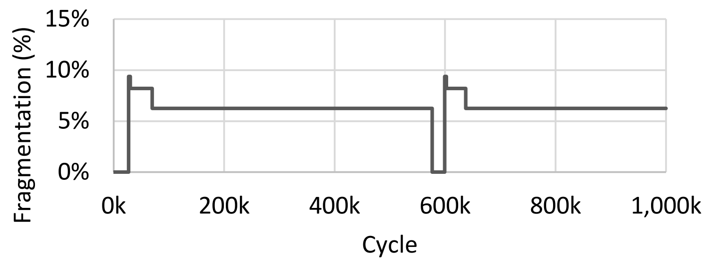


Figure 5.5: A timeline of register fragmentation when running FDTD/LUD on an SM with SMK.

actually contains the same register index for 32 threads to reduce the number of ports because a warp consists of 32 threads. When a warp is indexing register 0, the physical register index is computed by adding the register index 0 with a warp's register base register (RBR).

Figure 5.4 illustrates the resource fragmentation problem in SMK by showing an example with shared memory. A thread block from kernel 0 uses 2.5kB of shared memory, while a thread block from kernel 1 uses 3kB. The total size of the shared memory is 10kB. A resource fragmentation can occur whenever there is resource allocation or release: when a new kernel is launched, a kernel is terminated, or a preemption occurs due to dynamic resource repartitioning. Two examples are illustrated in detail.

Figure 5.4 (a) shows how resource fragmentation occurs when a kernel terminates.

When kernel 0 finishes all of its thread blocks, 4kB of shared memory is released for other kernels to use. In the given scenario, the released shared memory cannot be utilized by kernel 1 because kernel 1 requires 3kB of consecutive shared memory, while the released 4kB is in chunks of 2.5kB and 1.5kB. Figure 5.4 (b) depicts how shared memory is fragmented when a kernel is preempted partially. During the execution, the bottom thread block from kernel 1 is preempted due to repartitioning. A thread block from kernel 0 is launched instead. Later, kernel 1 eventually finishes, and thread blocks from kernel 0 will fill in. However, it can issue only one thread block rather than two because 2.5kB is fragmented into chunks of 0.5kB and 2kB. Similar resource fragmentation problems can occur for registers as well.

Figure 5.5 shows a timeline of fragmented registers when running FDTD/LUD on the same SM. Resources within an SM are partitioned evenly among the kernels. The percentage of fragmentation refers to the number of under-utilized registers due to fragmentation divided by the total number of registers in an SM. LUD launches three kernels with different resource requirements, and repeatedly relaunches the three kernels in the same order. As a result, similar fragmentation pattern is repeated around 600k cycles. LUD allocates resources first at the beginning. The resource fragmentation mainly occurs because LUD finishes earlier and launches a new kernel with a different resource requirement. The new kernel observes resource fragmentation similar to Figure 5.4 (a) and cannot launch the maximal number of threads that can fit in total available resources.

To avoid resource fragmentation, free resources should be to released resources when a kernel finishes execution, and a thread block to preempt should be chosen with released resources in mind.

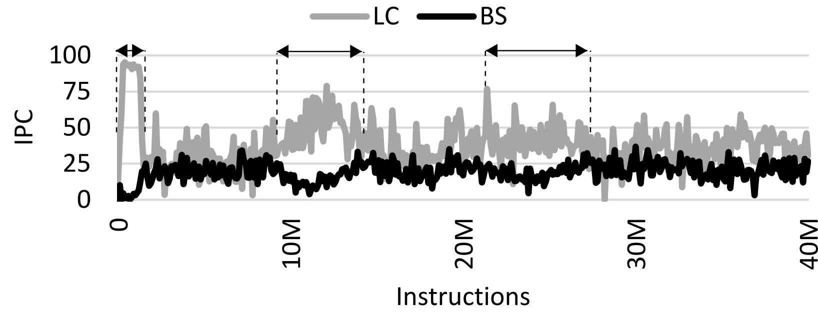


Figure 5.6: An IPC trace within an SM using a 50k instruction window when running LC/BS. LC is launched before BS. Sub-intervals, where BS starves, are shown.

5.3.5 Challenge 3: Starvation

The choice of warp scheduling can greatly impact SMK performance. For example, greedy-then-oldest (GTO) gives higher probability of issuing an instruction to the kernel that issued thread blocks earlier. This can lead to a starvation period for the other kernel. The starvation problem can become worse with unequal resource partitioning because a kernel with fewer warps will have even lower probability of issuing. Figure 5.6 shows an IPC trace within an SM using a 50k instruction window when running LC/BS, where LC is launched before BS. Even is used for thread block partitioning, and GTO is used for warp scheduling. Note that both LC and BS do not show significant phase changes during isolated execution. Because there is no phases nor thread block repartitioning, stable IPCs are expected for both benchmarks. However, as shown in the figure, starvation periods exist, where LC issues more instructions than its average while BS starves with lower IPC than its average. Although the performance gap during the starvation periods becomes smaller as the system progresses, they can take a large portion of execution, e.g., BS progresses by 50% near 30M instructions.

On the other hand, round-robin (RR) scheduling gives equal probability of issuing an

instruction to each warp. STP of SMK under RR is improved by 2.5%, on average, over SMK under GTO when the proposed thread block partitioning in Section 5.4.1 is used. While RR performs better than GTO for SMK in general, SMK under GTO may perform better than SMK under RR for individual workloads because GTO performs better than RR in most cases for single kernel execution [68]. To maximally benefit from SMK, the benefits of GTO should be retained within a single kernel execution, while avoiding starvation problem by allowing the other kernel to make progress.

5.4 GPU Maestro Design

This section introduces *GPU Maestro*, a dynamic resource management for efficient utilization of multitasking GPUs. Figure 5.7 illustrates an architectural overview of the overall system. The top of the figure illustrates 2-way resource allocation, where resources are provided to kernels from opposing ends of the resource pool. The bottom of the figure shows that the dynamic resource management framework collects runtime performance statistics from each SM using existing performance counters, and tells a thread block scheduler to repartition for multitasking GPUs. The thread block scheduler preempts thread blocks to achieve the desired resource partition similar to the prior works [83, 59]. *GPU Maestro* is composed of three components: dynamic thread block partitioning framework, 2-way resource allocation, and kernel-aware warp scheduling mechanism.

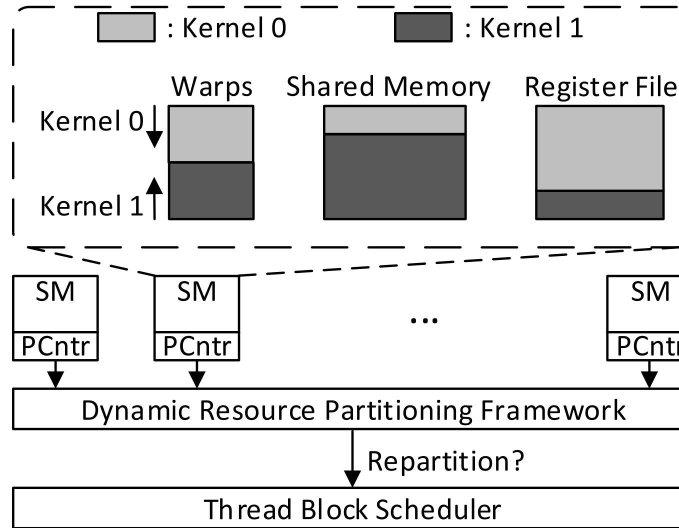


Figure 5.7: An architectural overview of *GPU Maestro*. PCntr refers to existing performance counters on an SM.

5.4.1 Dynamic Resource Partitioning

Figure 5.8 illustrates how *GPU Maestro* performs dynamic resource partitioning when two kernels are running on the GPU. At the top, the timeline of *GPU Maestro* is shown, where repartitioning decisions occur at the end of every epoch using the monitored performance. *GPU Maestro* uses 50k cycles for each epoch, which is long enough to minimize the repartitioning overhead. Note that after repartitioning decisions, the next epoch is not started until the preemptions take place and the desired thread block partition is achieved. On the bottom of the figure, the process of how resource partitioning decisions are made is shown. SMs are divided into three groups: dedicated, trial, and follower. Dedicated SMs run each kernel without SMK, and are used to measure the single kernel performance as well as the performance of spatial multitasking. Trial SMs test resource repartitioning possibilities for SMK GPUs. Follower SMs are partitioned with a resource partitioning that is measured to provide the best performance from the previous epoch.

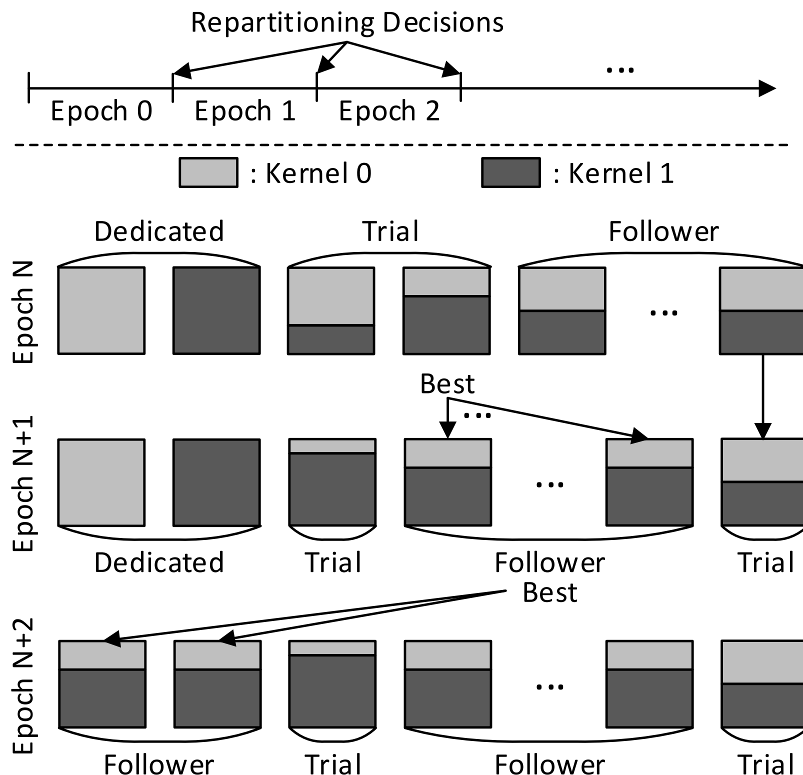


Figure 5.8: An illustration of dynamic resource partitioning process in *GPU Maestro* when two kernels are running on the GPU. *GPU Maestro* makes repartitioning decisions at the end of every epoch, which may not result in repartitioning if the previous preferred partition is the best. Follower SMs follows the repartitioning decision including spatial multitasking, which gives the best performance. Note that trial and follower SMs can change to minimize the preemption overhead from repartitioning. Dedicated SMs turn into follower SMs when a steady state is reached.

GPU Maestro assigns two SMs to trial SMs. To determine the thread block partition for trial SMs, *GPU Maestro* defines a trial kernel, which has larger resource requirements than the other kernel. A kernel has larger resource requirements if it can launch fewer thread blocks when having an SM entirely. The trial kernel assigns one more thread block for one trial SM, and one fewer thread block for the other compared to followers. The other kernel fills up the rest of the resources in the trial SMs. The initial partitioning state for follower SMs is SMK with even partitioning by default. *GPU Maestro* uses a history-based partition

prediction from the previous SMK for the initial state in followers when a running kernel was assigned fewer thread blocks than even partition during previous SMK.

When making repartitioning decisions, *GPU Maestro* tries to minimize the preemption overhead. In the figure, one of the trial SMs becomes a follower and vice versa from epoch N to epoch N+1. When dedicated SMs show stable performance as the SMK partition reaches a steady state, *GPU Maestro* stores the performance of dedicated SMs to be used in upcoming epochs, and turns dedicated SMs into follower SMs to maximize system performance. Dedicated SMs are reassigned when either a new kernel is launched, thread blocks are repartitioned, or once every hundred epochs. The last case ensures that the stored performance is indeed stable.

To determine which resource partition performs the best, *GPU Maestro* first defines the performance objective. As discussed in Section 5.2.1, there could be multiple metrics for multikernel execution. *GPU Maestro* can set one of these metrics for the performance objective. For example, when ANTT is set as the performance objective, *GPU Maestro* computes ANTT for spatial multitasking using dedicated SMs and SMK with different resource partition using trial and follower SMs. Whichever partition with the lowest ANTT will be selected because ANTT is a lower-is-better metric. To compute these metrics, CPIs for both multikernel execution as well as single kernel execution are required. *GPU Maestro* uses the CPI of dedicated SMs to estimate CPI for the single kernel execution.

When spatial multitasking performs better than SMK, *GPU Maestro* adds one SM per kernel from followers for spatial multitasking at each repartitioning epoch rather than turning all followers for spatial multitasking because additional SMs may not provide scalable performance as the memory-intensive benchmarks in Table 5.3 show. This brings addi-

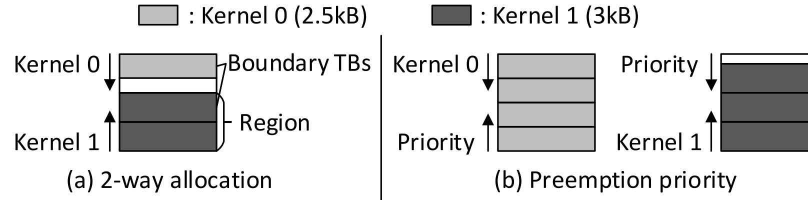


Figure 5.9: (a) The proposed 2-way allocation, where a kernel schedules thread blocks in the opposite direction, and (b) the fixed preemption priority order imposed by the 2-way allocation.

tional benefit to *GPU Maestro* by allowing trial and follower SMs to continuously provide SMK performance with uneven partitioning, which may perform better than spatial multitasking.

The proposed dynamic resource management framework requires 2-bits per SM to indicate whether it is dedicated, trial, or follower. Although existing performance counters are utilized, extra storage is also required to store dedicated SM's performance from previous epoch when dedicated SMs are turned into followers. Comparing and deciding the best performing partition are done in software running on the command processor.

5.4.2 2-Way Resource Allocation

By restricting the direction of allocation, kernels allocate resources consecutively in the steady state, which avoids the resource fragmentation when one of the kernel is finished. Figure 5.9 (a) shows the proposed 2-way allocation, where one-dimensional resources (RF, shared memory, and etc.) are allocated either from top to bottom or from bottom to top. Boundary thread blocks are defined as the thread blocks, where resource allocation meets the other kernel or the free space. A region is defined to be a group of resources, where kernels can safely launch its new thread blocks.

A fixed preemption priority order is also imposed for each kernel. Figure 5.9 (b) illustrates that the preemption priority is the reverse direction of the resource allocation direction. By forcing the preemption priority in the reverse direction, the already launched kernel can release resources adjacent to the free space, which will allow the other kernel to start allocating resources in the opposite direction. Thus, the resource fragmentation coming from both Figure 5.4 (a) and (b) can be avoided, and the other kernel can maximally utilize the released resources. The preemption overhead from the fixed preemption priority can be minimized by using prior work [59].

There is one limitation to 2-way allocation: the number of kernels running concurrently on an SM is limited to two. However, this does not force the shared GPU to run only two kernels because spatial multitasking can be mixed with 2-way allocation SMK to run more than two kernels. For example, when four kernels are launched concurrently, one SM may run first two kernels in SMK while other SMs can run other combinations of two kernels. Combining spatial multitasking with SMK is likely to be better than having SMK to support more than two kernels because the cost of avoiding resource fragmentation can be significant. Note that when *GPU Maestro* does not have to deal with all the concurrent kernels at once. *GPU Maestro* can take multiple steps to find the best performing resource partition when running more than two kernels, where each step is used to find the best partitioning for a single combination of two kernels to reduce the number of required SMs for dedicated and trial sets.

5.4.3 Kernel-aware Warp Scheduling

To avoid the starvation problem, choosing which kernel to issue instructions becomes an important problem in SMK GPUs similar to choosing which thread to fetch in SMT [85]. By having kernel-aware scheduling, each kernel can utilize GTO, which performs better for individual kernels, and still have enough issue slots to avoid the starvation problem. This chapter proposes to use loose round-robin (LRR). In LRR, kernel priority is flipped whenever a lower priority kernel has a ready instruction to issue. LRR gives almost equal opportunity for each kernel to progress. While LRR is the simplest mechanism, it achieves the best performance compared to other more intelligent schemes as shown in Section 5.5.2.

In SMK, there are situations where warp scheduling priority should diverge from the underlying warp scheduling policy. For example, when some of the warps are being preempted with draining [83, 59], these warps should be given higher priority to issue instructions to reduce repartitioning time. Another example is when a kernel has no more thread blocks to issue because it is near the end of execution. Again, the warps within the boundary thread block should be given higher priority to issue instructions because resources only become available to the other kernel when resources are released from the boundary thread block.

5.5 Results

The GPGPU-Sim v3.2.2 [5] is extended to simulate *GPU Maestro*. The Nvidia GTX980 is modeled, which is based on the most recent Maxwell architecture [50]. The system configuration is summarized in Table 5.2. A wide range of GPGPU applications from various

System	Parameters
SM	16 SMs, 1126 MHz, 4 warp schedulers 2K threads, 32 thread blocks 64K registers, 96kB shared memory 2kB L1I, 48kB L1D
Memory Subsystem	2MB L2, 4 memory partitions 224 GB/s bandwidth

Table 5.2: System configuration.

benchmark suites including Nvidia Computing SDK [51], Rodinia [7], and Parboil [82] is used for evaluation. Table 5.3 lists all the evaluated benchmarks, and their types. The types of the benchmarks are statically defined using their profiled performance with different numbers of SMs. COM, which is compute-intensive benchmarks, is defined as the benchmarks with more than 12x speedup when the number of SMs is changed from one to sixteen. MEM, which is memory-intensive benchmarks, is defined as the benchmarks that show less than 12x speedup. Although the GTX980 does not cache global memory accesses in the L1D by default, they are assumed to be cached at L1D using LDG intrinsic because many of the simulated GPGPU benchmarks have intra-warp locality and benefit from the L1D [68]. The performance objective of *GPU Maestro* is set to ANTT, and Chimera [59] is used for preemption.

All possible pairs of GPGPU benchmarks are used to simulate multi-programmed workloads from Table 5.3. The typical evaluation method from prior GPU multitasking works [3, 83, 59] and other CPU cache partitioning works [67, 61] is used. For each multi-programmed workload, all the benchmarks start simultaneously in the beginning. Each benchmark is run until it finishes its execution or 1 billion instructions. When one benchmark completes earlier than the other, it is restarted from the beginning to continuously

Benchmark	Source	Type
Breadth First Search (BFS)	Rodinia [7]	MEM
Back Propagation (BP)	Rodinia [7]	COM
BlackScholes (BS)	Nvidia SDK [51]	MEM
B+ Tree (BT)	Rodinia [7]	MEM
Coulombic Potential (CP)	Parboil [82]	COM
Finite-Difference Time-Domain (FDTD)	Nvidia SDK [51]	MEM
Fast Walsh Transform (FWT)	Nvidia SDK [51]	MEM
HotSpot (HS)	Rodinia [7]	COM
Heart Wall (HW)	Rodinia [7]	COM
Kmeans (KM)	Rodinia [7]	MEM
Laplace-Boltzmann Method (LBM)	Parboil [82]	MEM
Leukocyte Tracking (LC)	Rodinia [7]	COM
LavaMD (LMD)	Rodinia [7]	MEM
LU Decomposition (LUD)	Rodinia [7]	MEM
Magnetic Resonance Imaging (MRIQ)	Parboil [82]	COM
MUMmerGPU (MUM)	Rodinia [7]	MEM
Needleman Wunsch (NW)	Rodinia [7]	MEM
SAD	Parboil [82]	COM
Streamcluster (SC)	Rodinia [7]	MEM
SRAD	Rodinia [7]	MEM
Stencil (ST)	Parboil [82]	MEM
Two Point Angular Corr. Function (TPACF)	Parboil [82]	COM

Table 5.3: Benchmark specification.

stress the system. The reported results are gathered only for the first round of the execution, and the restarted executions are ignored.

5.5.1 Resource Partitioning Performance

In this section, *GPU Maestro* is compared to the baseline Spatial and SMK. Spatial partitions resources evenly among the kernels at the SM granularity, while SMK partitions resources evenly within the SMs. In both baselines, if resources are released from one kernel, the other kernel fills up the remaining resources.

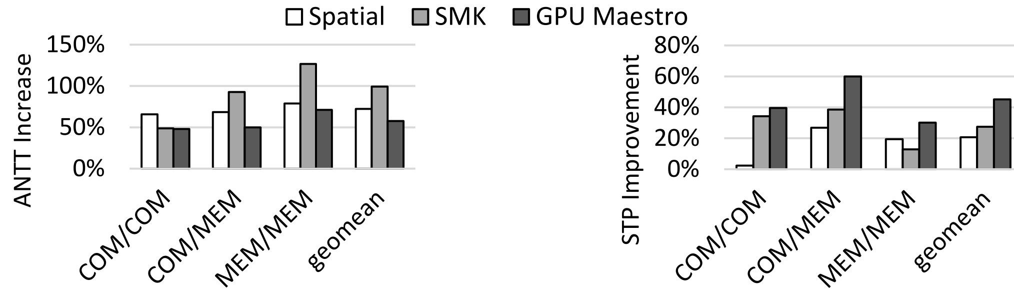


Figure 5.10: ANTT increase and STP improvement of Spatial, SMK, and *GPU Maestro* over non-shared execution. Spatial partitions resources evenly at the SM granularity. SMK partitions resources evenly within the SMs. A geometric mean of all combinations of co-running kernels from each category is shown. ANTT is a lower-is-better metric, and STP is a higher-is-better metric.

Figure 5.10 shows ANTT increase and STP improvement of Spatial, SMK, and *GPU Maestro* over non-shared execution. When sharing the GPU, ANTT, which measures response time, is increased compared to running it on GPU in isolation. On average, Spatial, SMK, and *GPU Maestro* increase ANTT by 72.2%, 99.1%, and 57.6%, respectively, while improving STP by 20.7%, 27.3%, and 45.0%, respectively. In general, SMK is better than Spatial for COM/COM and COM/MEM, but Spatial is better than SMK for MEM/MEM. *GPU Maestro* outperforms both Spatial and SMK because it can exploit both schemes, and also finds the best performing partition within SMK.

In COM/COM pairs, SMK with even partitioning performs similar to any other SMK partitionings as discussed in Section 5.3.1. As a result, the performance gap between SMK and *GPU Maestro* is small for COM/COM. However, the performance gap is much higher for COM/MEM and MEM/MEM. Because *GPU Maestro* can adjust resource partitions such that both SMK with uneven partitionings and Spatial are utilized, it can reduce the effect of interference between kernels. The system performance is improved the most with *GPU Maestro*.

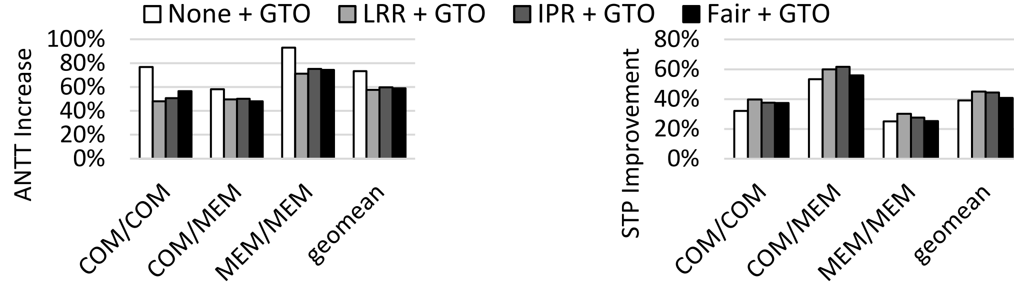


Figure 5.11: ANTT increase and STP improvement of various kernel-aware scheduling techniques on *GPU Maestro* over non-shared execution. A geometric mean of all combinations of co-running kernels from each category is shown. All the kernel-aware scheduling techniques are implemented on top of GTO warp scheduling. None does not do any kernel-aware scheduling. ANTT is a lower-is-better metric, and STP is a higher-is-better metric.

5.5.2 Kernel-aware Scheduling Performance

This section compares three kernel-aware scheduling techniques: LRR, issued-per-ready (IPR), and Fair [90]. All the techniques are implemented on top of GTO warp scheduling. To show the benefit of kernel-aware scheduling, None, which does not do any kernel-aware scheduling, is also shown. All the techniques use *GPU Maestro*'s dynamic thread block partitioning.

IPR: IPR mimics ICOUNT [85] for GPUs by using following metric for each warp scheduler:

$$IPR = \frac{\sum_{cycles} \#\ of\ issued\ warps}{\sum_{cycles} \#\ of\ ready\ warps} \tag{5.3}$$

IPR is cumulated over every cycle. A kernel with lower IPR has higher priority to issue an instruction. This metric avoids starvation because IPR becomes smaller as a kernel waits and does not issue instructions. IPR also favors compute-intensive kernels because they have more ready warps compared to memory-intensive kernels, which makes IPR lower.

Fair: Fair uses following equation to compute quota for each kernel:

$$Quota_k = \frac{C_k}{\sum C_k}, \quad C_k = x\% \times \frac{S_k}{T_k} \tag{5.4}$$

where $x\%$ is the percentage of issued cycles, which is profiled from single kernel execution, and T_k is the number of thread blocks in an SM when the kernel is run in isolation, and S_k is the number of thread blocks allocated in SMK for the kernel. This is equivalent to SMK-W in [90] except for the thread block partitioning.

Figure 5.11 illustrates ANTT increase and STP improvement of None, LRR, IPR, and Fair over non-shared execution. *GPU Maestro*'s dynamic resource partitioning is used for all the kernel-aware scheduling techniques. On average, None, LRR, IPR, and Fair increase ANTT by 73.4%, 57.6%, 59.6%, and 58.9%, respectively, while improving STP by 39.0%, 45.0%, 44.4%, and 40.8%, respectively. In general, all kernel-aware scheduling techniques reduce ANTT and improve STP compared to not having kernel-aware scheduling.

Among LRR, IPR, and Fair, LRR performs the best for both ANTT, and STP. Interestingly, the trend of improvement is the opposite of the implementation complexity. For LRR, a single bit for each kernel is enough to notify whether it has a ready warp to issue. For IPR, counters are needed to record the number of issued warps as well as the number of ready warps, and a comparison logic. For Fair, extra logic is required to compute quota, and the percentage of issued cycles has to be profiled or measured directly on the GPU for each kernel. The main reason for the performance difference between these kernel-aware scheduling techniques goes back to the starvation problem. For example, *GPU Maestro* assigns fewer thread blocks to MEM application when running COM/MEM combinations.

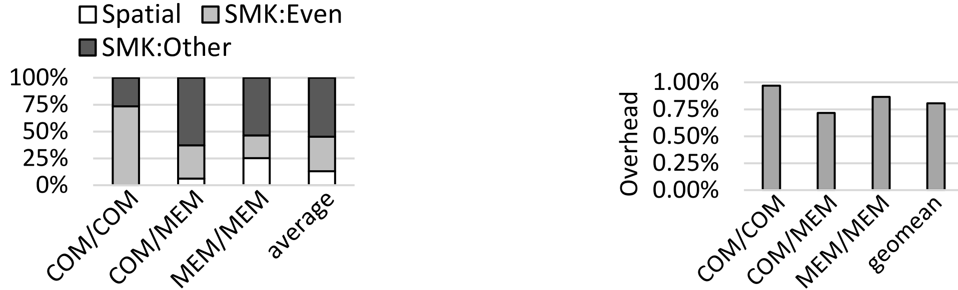


Figure 5.12: (Left) The percentage of repartitioning decisions to use spatial multitasking, SMK with even partitioning, and SMK with other non-even partitioning, and (Right) repartitioning overhead in *GPU Maestro*. An average of all combinations of co-running kernels from each category is shown.

When running COM/MEM combinations in Fair, the quota for MEM application will be small because S_k becomes smaller due to *GPU Maestro*, and $x\%$ is already smaller than the COM application. While Fair still allocates more issue slots to the MEM application compared to not having kernel-aware scheduling, it is more unfair compared to LRR and IPR. As a summary, kernel-aware scheduling is necessary in SMK, and the simplest LRR kernel-aware scheduling in fact gives the best performance both in terms of ANTT and STP.

5.5.3 Repartitioning Analysis

This section analyzes the dynamic thread block repartitioning in detail. Figure 5.12 (Left) illustrates the distribution of repartitioning decisions in *GPU Maestro*. Spatial is when *GPU Maestro* chooses to use spatial multitasking instead of SMK. SMK:Even denotes when *GPU Maestro* chooses Even as the best performing thread block partition on SMK, and SMK:Other refers to the decisions when *GPU Maestro* assigns unequal resources to the kernels on SMK. On average, *GPU Maestro* decides to utilize spatial multitasking for 13.0% of the time, SMK:Even for 32.2%, and SMK:Other for 54.8%. Among

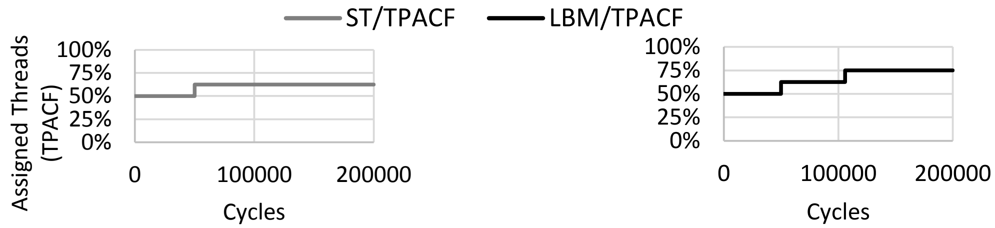


Figure 5.13: Thread block repartitioning timeline for ST/TPACF and LBM/TPACF.

SMK:Other, 29.4% of the time corresponds to having one thread block difference from SMK:Even, and the remaining 70.6% comes from having more unequal resource assignments. Analyzing each category, SMK:Even is utilized the most in COM/COM, while SMK:Other is used the most in other categories. Also, spatial multitasking is used often for MEM/MEM. These results are consistent to the study of motivating examples in Section 5.3.1.

Figure 5.12 (Right) depicts the repartitioning overhead in *GPU Maestro*. To measure this overhead, an ideal case, where thread block repartitioning takes place with zero overhead, was run to measure the STP. The overhead can be thought of as the frequency of thread block repartitioning multiplied by how much system throughput is wasted during the repartitioning. On average, *GPU Maestro* has 0.8% overhead from dynamic thread block repartitioning, which is small. Also, the absolute difference of the overhead is small across the categories.

Figure 5.13 depicts the thread block repartitioning timeline for two example benchmark pairs. Because TPACF is COM and the paired benchmark is MEM, TPACF is assigned more threads. However, the amount of unequal assignment can be different depending on the interaction between the benchmarks on SMK GPUs.

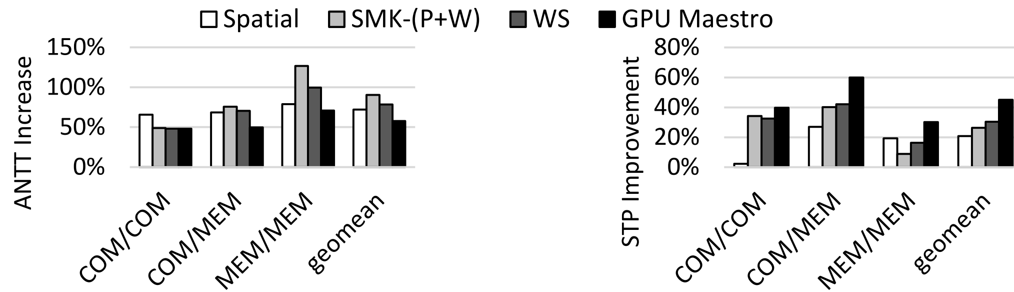


Figure 5.14: ANTT increase and STP improvement of Spatial, SMK-(P+W), WS, and *GPU Maestro* over non-shared execution. A geometric mean of all combinations of co-running kernels from each category is shown. ANTT is a lower-is-better metric, and STP is a higher-is-better metric.

5.5.4 Comparison to Prior Works

This section compares *GPU Maestro* with three prior works: spatial multitasking [3], SMK-(P+W) [90], and Warped-Slicer (WS) [93]. For spatial multitasking at the SM granularity, Smart is used as the SM partitioning heuristic, which partitions SMs evenly among applications. However, it is guaranteed that no application is given more SMs than it can fill up with thread blocks. SMK-(P+W) and WS partition resources within the SM granularity. SMK-(P+W) uses the dominant resource share to partition resources within an SM, and applies fair warp scheduling. WS uses single kernel execution profiles to partition resources within an SM.

Figure 5.14 illustrates ANTT increase and STP improvement of the prior works and *GPU Maestro* over non-shared execution. On average, Spatial, SMK-(P+W), WS, and *GPU Maestro* increases ANTT by 72.2%, 90.3%, 78.4%, and 57.6%, respectively, while improving STP by 20.7%, 26.2%, 30.3%, and 45.0%, respectively. *GPU Maestro* outperforms Spatial, SMK-(P+W), and WS in both ANTT and STP.

For a further in-depth analysis, Figure 5.15 shows STP improvement of the techniques

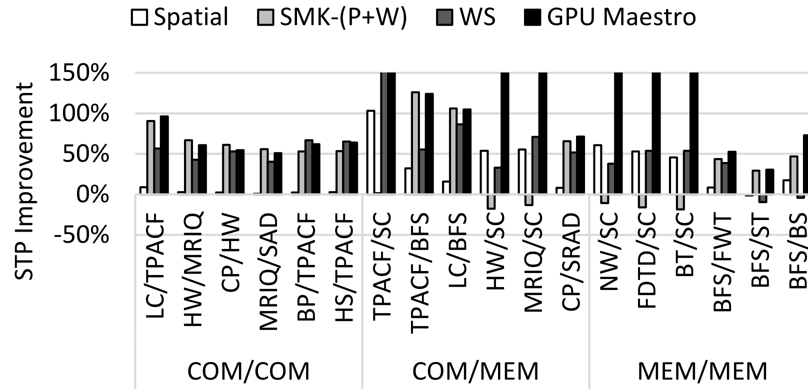


Figure 5.15: STP improvement of Spatial, SMK-(P+W), WS, and *GPU Maestro* over non-shared execution for selected benchmark pairs.

over non-shared execution for selected benchmark pairs. From each category, three pairs, where the performance improvement of SMK with even thread block partitioning over Spatial is the largest, and three pairs with opposite characteristics were selected. For the COM/COM category, SMK always performs better than Spatial hence six pairs with the former characteristic were picked. As shown by the figure, *GPU Maestro* performs similar or better than any prior works. Because *GPU Maestro* can select the best performing resource partition regardless of whether Spatial or SMK is favored, it achieves better performance compared to prior works that focus on one of the scheme. Moreover, *GPU Maestro* can capture interference between kernels, which is not taken into account for WS, which only utilizes single kernel execution profiles. For example, benchmark pairs including BFS, which had the largest interference with the other kernel by slowing it down, do not perform well for WS.

5.6 Related Work to *GPU Maestro*

In this section, the prior works on SMT CPUs are first described. Then, the previous studies on the GPU multitasking are discussed.

5.6.1 Simultaneous Multithreading

SMT [86] was first introduced to maximize throughput of superscalar, out-of-order processors by allowing fine-grained multithreading within a CPU core. SMT was further extended by ICOUNT [85], which explored the instruction fetch and issue priorities between the threads in an SMT core. SMT was successfully adopted by industry [30], which was also renamed as hyperthreading in Intel CPUs [37, 84]. Unlike SMT CPUs, SMK GPUs introduce new problems like thread block partitioning and resource fragmentation. *GPU Maestro* addressed these new problems, and also dealt with the instruction issue priorities by studying kernel-aware warp scheduling.

5.6.2 GPU Multitasking

The first attempts on GPU multitasking came from software approaches by merging kernels at compile time [55] or providing abstractions at the operating system level [70]. These approaches often require modifications to the source code, which may not be applicable in general cases. More recently, hardware preemption mechanisms were studied. Tanasic et al. [83] studied two preemption techniques: traditional context switching, and SM draining. In SM draining, an SM is no longer scheduled with new thread blocks. When the running thread blocks are finished, the SM can be preempted. Chimera [59] further en-

abled fast preemption with SM flushing, and demonstrated that more efficient preemptive multitasking is possible by using different preemption techniques for each thread block. By having hardware supports for multitasking GPUs, existing GPU kernels can be seamlessly take advantage of them.

Elastic kernels [55] controls the resource usage of kernels in more fine-grained manner so that resource utilization on SMs can be improved. However, their evaluation of multitasking is limited to the timeslice of a kernel rather than preemptive multitasking. Persistent threads [23, 91] are software approaches to enable spatial multitasking. However, persistent threads require the programmers to explicitly change the kernels to fit in the framework with increased difficulty for debugging. Moreover, extra registers required by the framework may not be acceptable for register-constrained kernels.

Spatial multitasking [3] proposed to run multiple kernels on shared GPUs at the SM granularity. While spatial multitasking improved system throughput over single kernel execution, it also showed that the performance difference between various SM partitioning schemes were relatively small. The idea of SMK [90] has been explored, however, it only addressed DRF thread block partitioning, which focuses on fair resource allocation. WS [93] improves by using single kernel execution profiles to consider uneven partitioning, however, it did not address the interference between kernels. Moreover, prior works on SMK did not study resource fragmentation problem in depth. *GPU Maestro* addresses these problems, and achieves better performance than these prior works by utilizing both spatial multitasking and SMK.

5.7 *GPU Maestro* Conclusions

This chapter presented *GPU Maestro*, a dynamic resource management for efficient utilization of multitasking GPUs. *GPU Maestro* identified that spatial multitasking and SMK have their own advantages and disadvantages depending on the application mixes. *GPU Maestro* explores which resource partition provides the best performance by testing different partitions with a subset of SMs and directly measuring the performance of these SMs. *GPU Maestro* also identified two challenges in implementing SMK GPUs. First, it showed the existence of resource fragmentation problem, and proposed 2-way resource allocation, which forces kernels to allocate and release resources consecutively in opposing directions. Second, *GPU Maestro* also demonstrated that kernel-aware warp scheduling is critical to fully benefit from SMK, and suggested that simple LRR kernel-aware scheduling performs the best. Evaluations have shown that *GPU Maestro* increases the ANTT by an average of 57.6% while improving the STP by an average of 45.0% over non-shared execution.

CHAPTER VI

Conclusion

Heterogeneous systems are becoming increasingly popular in modern computer systems due to its high energy efficiency and performance. Additional to CPUs, many variants of programmable accelerators have been studied: application-specific processors that are targeting a specific application domain [9, 12, 73], field-programmable gate arrays [65], extended coarse-grained reconfigurable architecture (CGRA) [57, 62, 58], extended SIMD cores [11, 63, 64], or GPUs. Among them, GPUs are the first to be used in general-purpose across mobile devices to data centers.

As GPUs are increasingly being adopted in modern computer systems beyond their traditional role of processing graphics to accelerating data-parallel applications, a scenario of sharing an GPU among multiple applications is becoming viable. For example, cloud computing services like AWS can save the infrastructure costs by having its customers to share GPUs rather than assigning fixed set of GPUs to them. Any field-deployed devices such as autonomous cars or internet of things (IoT), which exploit GPUs for data-parallel processing, need to utilize shared GPUs because they are limited by power and cost budgets.

This thesis proposed a framework with hardware/software extensions on GPUs, which enables efficient resource utilization of multitasking GPUs. Chapter III presented *ELF*, which is a GPU scheduling mechanism for single kernel execution. *ELF* utilizes both compiler and hardware to give higher priority to the warps that have fewer remaining instructions to the next memory load operation. Evaluations show that *ELF* can improve the performance by 4.1% over the greedy-then-oldest (GTO) scheduler with only 1.39kB extra storage per SM. When used with other techniques like NewCAR and instruction prefetching, *ELF* can achieve total speedup of 11.9% over the GTO.

Chapter IV presented *Chimera*, a collaborative preemption to enable efficient preemptive multitasking on GPUs. *Chimera* first proposed a preemption technique called flushing, which combines the concept of idempotent execution to preemption with the independence of thread block execution to enable low preemption latency. The key idea behind Chimera is to preempt each thread block with the most efficient preemption technique by dynamically estimating the preemption costs. Intuitively, Chimera should incur less overhead than any single preemption technique if preemption costs are estimated correctly. Evaluations show that *Chimera* violates a 15μs preemption latency constraint for only 0.2% of the preemption requests. For multi-programmed workloads, *Chimera* can improve the average normalized turnaround time by 5.5x, which can go up to 25.4x when a large number of preemption requests occur. System throughput can be improved by 12.2%, which can go up to 41.7% when a large number of preemption requests exist.

Chapter V presented *GPU Maestro*, a dynamic resource management framework for efficient utilization of multitasking GPUs. *GPU Maestro* showed that unfair thread block partitioning within SMK GPUs can benefit more than fair partitioning. Furthermore, *GPU*

Maestro illustrated when spatial multitasking can perform better than SMK GPUs. Due to interference and dynamism within SMK GPUs, *GPU Maestro* suggested testing different thread block partitioning with a subset of SMs to determine the best performing partition. *GPU Maestro* addresses two additional problems: resource fragmentation problem, and starvation problem. *GPU Maestro* proposed 2-way resource allocation, which forces kernels to allocate and release resources consecutively in opposing directions, to overcome resource fragmentation. *GPU Maestro* also suggested that simple LRR kernel-aware scheduling can avoid the starvation. Evaluations have shown that *GPU Maestro* increases the ANTT by an average of 57.6% while improving the STP by an average of 45.0% over non-shared execution. Compared to the baseline spatial multitasking and SMK approaches, *GPU Maestro* improves average STP by 20.2% and 13.9%, respectively.

This thesis have introduced novel techniques to enable efficient resource utilization on multitasking GPUs. The SIMT programming model has unique properties compared to the programming models on CPUs, and this thesis showed to how to exploit these properties to make multitasking GPUs more efficient. Moreover, the techniques are not limited to the multitasking GPUs, but any large-state acclerators that utilize the SIMT programming model.

While this thesis have addressed major challenges for efficient resource utilization on multitasking GPUs, there are several opportunities still remaining for further investigation that can extend the proposed framework. For example, other shared resources like the memory partitions including the L2 cache, and DRAM controllers can be further examined to reduce contention. Energy efficiency of multitasking GPUs can be also explored, and can be taken into account as one parameter in the proposed framework. To maximize the

benefits from these opportunities, approaches from this thesis, which utilized both software/hardware extensions to exploit unique characteristics of the GPU programming and execution models, can still be promising.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Green500 list, 2015. http://www.green500.org/lists/green201506. 1, 75

[2] Top500 list, 2015. http://www.top500.org/lists/2015/06/. 1, 75

[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012. 4, 14, 45, 48, 49, 55, 62, 68, 72, 76, 96, 103, 106

[4] Amazon. Amazon web services. https://aws.amazon.com/ec2/. 1, 75

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009. 30, 31, 32, 61, 95

[6] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296, 2012. 45, 72, 75

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron.

Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009. 31, 32, 61, 62, 96, 97

[8] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu. Adaptive cache management for energy-efficient GPU computing. In *Proc. of the 47th Annual International Symposium on Microarchitecture*, pages 343–355, 2014. 2

[9] S. Choi, J. J. K. Park, M. Koo, D. Kim, and S.-I. Chae. A 40 mbps H.264/AVC CAVLC decoder using a 64-bit multiple-issue video parsing coprocessor. In *Proc. of the 23rd SOC Conference*, pages 105–108, 2010. 108

[10] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 76–87, June 2004. 39

[11] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 216–227, 2007. 108

[12] G. Dasika, A. Sethia, V. Robby, T. Mudge, and S. Mahlke. Medics: Ultra-portable processing for medical image reconstruction. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 181–192, 2010. 108

[13] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework

for software recovery of hardware faults. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 497–508, June 2010. 73

[14] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 140–151, 2011. 45, 50, 73

[15] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. of the 1998 International Conference on Supercomputing*, pages 68–75, 1997. 39

[16] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008. 69, 79

[17] S. Feng, S. Gupta, A. Ansari, S. Mahlke, and D. August. Encore: Low-cost, fine-grained transient fault recovery. In *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 398–409, 2011. 45, 50, 73

[18] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007. 2

[19] M. Galloy. CPU vs GPU performance, 2013. http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html. viii, 2

[20] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in through-

put processors. In *Proc. of the 38th Annual International Symposium on Computer Architecture*, pages 235–246, 2011. 40

[21] A. Glew. MLP yes! ILP no!, 1998. In ASPLOS Wild and Crazy Idea Session'98. 39

[22] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proc. of the 4th USENIX Workshop on Hot Topics in Parallelism*, page 10, 2012. 71

[23] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *innovative parallel computing(InPar)*, pages 1–14, 2012. 106

[24] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001. 60

[25] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009. 77, 83

[26] F. Ino, A. Ogita, K. Oita, and K. Hagihara. Cooperative multitasking for GPU-accelerated grid systems. *Concurrency and Computation: Practice & Experience*, 24(1):96–107, 2012. 71

[27] D. A. Jamshidi, M. Samadi, and S. Mahlke. D^2MA: Accelerating coarse-grained data transfer for GPUs. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 431–442, 2014. 40

[28] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 272–283, Feb. 2014. 15, 27, 41

[29] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 395–406, Mar. 2013. 15, 18, 41

[30] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multi-threaded processor. *IEEE Micro*, 24(2):40–47, 2004. 105

[31] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66, 2011. 45, 46, 71

[32] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC'11)*, pages 17–30, 2011. 45, 75

[33] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, 2013. 2, 15, 31, 33, 37, 41

[34] KHRONOS. OpenCL - the open standard for parallel programming of heterogeneous systems, 2014. 1, 15, 75

[35] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 2–11, 2001. 45, 50, 73

[36] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke. WarpPool: Sharing requests with inter-warp coalescing for throughput processors. In *Proc. of the 48th Annual International Symposium on Microarchitecture*, pages 433–444, 2015. 2

[37] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003. 105

[38] N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on GPU performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, pages 1–10, 2010. 18

[39] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, pages 213–224, 2010. 2, 30, 40

[40] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *Proc. of the 20th*

International Symposium on High-Performance Computer Architecture, pages 260–271, 2014. 2, 41, 71

[41] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 317–328, 2012. 39

[42] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 235–246, 2010. 2

[43] J. Menon, M. de Kruijf, and K. Sankaralingam. iGPU: Exception support and speculative execution on GPUs. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 72–83, June 2012. 45, 46, 50, 73

[44] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, Feb. 2003. 39

[45] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU performance via large warps and two-level warp scheduling. In *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 308–317, 2011. 2, 15, 18, 31, 37, 40

[46] NVIDIA. Fermi: Nvidias next generation cuda compute architecture,

2009. http://www.nvidia.com/content/PDF/ fermi_white_papers/NVIDIA_Fermi_-Compute_Architect - ure_Whitepaper.pdf. 30, 55, 61

[47] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf. 44, 49, 59, 76

[48] NVIDIA. NVIDIA CUDA C Programming Guide, version 5.5, 2013. 1, 15

[49] NVIDIA. *CUDA C Programming Guide*, 2014. http://docs.nvidia.com/cuda. 75

[50] NVIDIA. NVIDIA GeForce GTX 980: Featuring Maxwell, the most advanced GPU ever made, 2014. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF. 7, 95

[51] NVIDIA. NVIDIA GPU Computing SDK, 2014. http://developer.nvidia.com/gpu-computing-sdk. 31, 32, 61, 62, 96, 97

[52] NVIDIA. Sharing a GPU between MPI processes: Multi-process service (MPS) overview, 2014. http://docs.nvidia.com/deploy/mps/index.html. 44, 76

[53] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, Dec. 1996. 4

[54] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz. The IBM system/370 vector

architecture: Design considerations. *IEEE Transactions on Computers*, 37(5):509–520, 1988. 72

[55] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 407–418, Mar. 2013. 45, 49, 71, 105, 106

[56] V. S. Pai and S. Adve. Code transformations to improve memory parallelism. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 147–155, Nov. 1999. 39

[57] H. Park, K. Fan, S. Mahlke, T. Oh, H. Kim, and H. seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, Oct. 2008. 108

[58] J. J. K. Park, Y. Park, and S. Mahlke. Efficient execution of augmented reality applications on mobile programmable accelerators. In *Proc. of the 2013 International Conference on Field Programmable Logic and Applications*, pages 176–183, 2013. 108

[59] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 593–606, Mar. 2015. 7, 13, 14, 76, 89, 94, 95, 96, 105

[60] J. J. K. Park, Y. Park, and S. Mahlke. ELF: Maximizing memory-level parallelism for GPUs with coordinated warp and fetch scheduling. In *Proceedings of SC15: the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2015. 6, 82

[61] J. J. K. Park, Y. Park, and S. Mahlke. Fine grain cache partitioning using per-instruction working blocks. In *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques*, pages 305–316, Oct. 2015. 96

[62] Y. Park, J. J. K. Park, and S. Mahlke. Efficient performance scaling of future CGRAs for mobile applications. In *Proc. of the 2012 International Conference on Field Programmable Logic and Applications*, pages 335–342, Dec. 2012. 108

[63] Y. Park, J. J. K. Park, H. Park, and S. Mahlke. Libra: Tailoring SIMD execution using heterogeneous hardware and dynamic configurability. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 84–95, 2012. 108

[64] Y. Park, S. Seo, H. Park, H. K. Cho, and S. Mahlke. SIMD defragmenter: Efficient ILP realization on data-parallel architectures. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 363–374, 2012. 108

[65] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale data-

center services. In *Proc. of the 41st Annual International Symposium on Computer Architecture*, pages 13–24, 2014. 108

[66] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 167–178, June 2006. 39, 68

[67] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance runtime mechanism to partition shared caches. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 423–432, 2006. 96

[68] T. G. Rogers, M. OĆonnor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 72–83, 2012. 2, 15, 18, 31, 33, 37, 41, 89, 96

[69] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 99–110, 2013. 18, 41

[70] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011. 75, 105

[71] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proc. of the '12 Conference on Programming Language Design and Implementation*, pages 13–22, 2012. 15

[72] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 440–451, 2012. 15

[73] A. Sethia, G. Dasika, T. Mudge, and S. Mahlke. A customized processor for energy efficient scientific computing. *IEEE Transactions on Computers*, 61(12):1711–1723, 2012. 108

[74] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 73–82, 2013. 2, 30, 40

[75] A. Sethia, D. A. Jamshidi, and S. Mahlke. Mascar: Speeding up GPU warps by reducing memory pitstops. In *Proc. of the 21st International Symposium on High-Performance Computer Architecture*, pages 174–185, Feb. 2015. viii, 2, 15, 18, 27, 29, 36, 42

[76] A. Sethia and S. Mahlke. Equalizer: Dynamic tuning of GPU resources for efficient execution. In *Proc. of the 47th Annual International Symposium on Microarchitecture*, pages 647–658, 2014. 2

[77] S. Shivshankar, S. Vangara, and A. G. Dean. Balancing register pressure and context-switching delays in ASTI systems. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 286–294, Sept. 2005. 72

[78] A. Silberschatz et al. *Operating System Concepts*. John Wiley and Sons, Inc, Indianapolis, IN, 2001. 12

[79] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 588–597, 1992. 29

[80] J. S. Snyder, D. B. Whalley, and T. P. Baker. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems*, 19(1):35–42, 1995. 72

[81] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ilp processors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 380–391, June 1998. 39

[82] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Mar. 2012. 31, 32, 61, 62, 96, 97

[83] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *Proc. of the 41st Annual International Symposium on Computer Architecture*, pages 193–204, 2014. 14, 45, 46, 49, 51, 55, 68, 72, 76, 89, 95, 96, 105

[84] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithread-

ing pentium 4 processor. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 26–34, 2003. 68, 105

[85] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996. 95, 99, 105

[86] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995. 4, 105

[87] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: Fairly measuring multithreaded architectures. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 305–316, 2007. 68

[88] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. of the 10th European Conference on Computer Systems*, 2015. 3, 75

[89] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *2011 International Conference on High Performance Computing and Simulation*, pages 24–32, 2011. 71

[90] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proc.*

of the 22nd International Symposium on High-Performance Computer Architecture, pages 358–369, Mar. 2016. 4, 14, 76, 99, 100, 103, 106

[91] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proc. of the 2015 International Conference on Supercomputing*, pages 119–130, June 2015. 106

[92] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for GPUs. In *Proc. of the 21st International Symposium on High-Performance Computer Architecture*, pages 76–88, Feb. 2015. 2

[93] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *Proc. of the 43rd Annual International Symposium on Computer Architecture*, page To appear, 2016. 14, 76, 103, 106

[94] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, pages 382–393, Feb. 2011. 77, 83

[95] H. Zhou and T. M. Conte. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Transactions on Computers*, 54(7):897–912, 2005. 39

[96] X. Zhou and P. Petrov. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proc. of the 43rd Design Automation Conference*, pages 352–357, 2006. 72

[97] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *acm computing surveys*, 45(1):4:1–4:28, 2012. 4