Data Resource Management in Throughput Processors

by

John S. Kloosterman

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
2018

Doctoral Committee:

Professor Scott Mahlke, chair
Professor Trevor Mudge
Professor Kevin Pipe
Assistant Professor Lingjia Tang

John S. Kloosterman

jklooste@umich.edu

ORCID iD: 0000-0001-8180-1237

© John S. Kloosterman 2018

ACKNOWLEDGMENTS

Completing this dissertation would not have been possible without the support of many people.

I would like to thank Scott Mahlke for making this research possible as my advisor; his insight made an impact on every part of this dissertation. Alongside his research mentorship, he has also helped me learn the other skills that make research successful – presenting my work in a way that is understandable and convincing, as well as managing the different stakeholders in the work. I also had the privilege of working with Trevor Mudge throughout my time at Michigan. Lingjia Tang and Kevin Pipe have provided valuable insights as committee members that improved this work.

I am proud to have been a part of the CCCP research group. As I started graduate school, Ankit Sethia showed me how to be an effective graduate student during our meetings together with Scott. Mehrzad Samadi, Hyoun Kyu Cho, Janghaeng Lee, Daya Khudia, Gaurav Chadha, Jason Park, Shruti Padmanabha, and Andrew Lukefahr were a welcoming group of people who I had the privilege of joining and working alongside. I enjoyed collaborating with Anoushe Jamshidi on our GPU research; many of the ideas in Chapter 4 came out of our conversations. Jiecao Yu, Babak Zamirai, Jon Bailey, Shikai Li, Sunghyun Park, Salar Latifi, Hossein Golestani, Ze Zhang, and Pedram Zamirai have been a great group to work alongside these past years. Jonathan Beaumont was also a pleasure to work with on the power models used in this dissertation.

I am grateful to my mentors through the different stages of my career so far. Joel Adams gave me my first experience doing research with GPUs as an undergraduate, helped me through the graduate school admissions process, and has continued being there throughout my Ph.D. As I prepare for life after graduate school, Drew DeOrio has helped me through the job search process and given me the nudges I needed to finish writing.

I could not be here without the support of my parents, John and Kim. From teaching me how

to use DOS at four years old, to using the farm truck to pick up loads of free computer parts, to giving me all that space in the basement to tinker, they have always been encouraging and supportive. They have been patient through all the times I have had to stay in the States to work on research.

Finally, I could not have made it through graduate school without the support of my wife Liz. She has been there through all the exams, paper deadlines, conferences, and job interviews that would otherwise have been too much for me. She'll be glad I can finally get some rest.

TABLE OF CONTENTS

Acknowledgments . . . . **ii**

List of Figures . . . . **vii**

List of Tables . . . . **xi**

Abstract . . . . **xii**

Chapter

1 Introduction **1**

- 1.1 Data Management Inefficiencies 4
  - 1.1.1 Memory Divergence and Cache Thrashing 5
  - 1.1.2 Register File Energy Overhead 5
  - 1.1.3 Inter-Application Contention 6
- 1.2 Contributions 6
  - 1.2.1 Increasing Memory Throughput 6
  - 1.2.2 Reducing Register File Energy and Storage 7
  - 1.2.3 Controlling Interference in Shared GPUs 7

2 Background **9**

- 2.1 SM Design 9
- 2.2 Memory System Design 10
- 2.3 Programming Model 11
- 2.4 Design Convergence in Desktop, Data Center, and Mobile 12

3 Inter-Warp Memory Request Merging and Prioritization **13**

- 3.1 Introduction 13
- 3.2 Background and Motivation 15
  - 3.2.1 Background 15
  - 3.2.2 Oversubscription of L1 Bandwidth 18
  - 3.2.3 Increasing Coalescing Window Size 20
- 3.3 WarpPool Design 21
  - 3.3.1 Overview 21
  - 3.3.2 Instruction Queues 23
  - 3.3.3 Intra-Warp Coalescers 24
  - 3.3.4 Inter-Warp Coalescing Queues 24

3.3.5 Request Selector . . . 25
3.3.6 Metadata Tracker . . . 26
3.3.7 Writeback . . . 28
3.3.8 Stores and Memory Consistency . . . 28
3.3.9 Resource Configuration . . . 29
3.3.10 Verilog Implementation . . . 30
3.4 Evaluation . . . 32
3.4.1 Methodology . . . 32
3.4.2 Results . . . 33
3.4.3 Case Study . . . 36
3.5 Related Work . . . 37
3.6 Conclusion . . . 38

4 Register File Storage and Energy Reduction . . . 40

4.1 Introduction . . . 40
4.2 RF Replacement Challenges . . . 43
4.2.1 Capacity Allocation . . . 43
4.2.2 Memory Side Bandwidth . . . 45
4.2.3 L1 Cache Capacity . . . 46
4.3 Design Overview . . . 46
4.4 Compiler Code Generation . . . 48
4.4.1 Region Creation . . . 48
4.4.2 Region Creation Algorithm . . . 50
4.4.3 Register Lifetime . . . 50
4.4.4 Control Flow and Register Liveness . . . 52
4.5 Hardware Design . . . 53
4.5.1 Capacity Managers (CMs) . . . 55
4.5.2 Operand Staging Units (OSUs) . . . 55
4.5.3 Compressor . . . 58
4.5.4 Metadata Encoding . . . 59
4.6 Evaluation . . . 60
4.6.1 Methodology . . . 60
4.6.2 Area and Power . . . 61
4.6.3 Energy Savings . . . 62
4.6.4 Performance . . . 64
4.6.5 Register Preload Location, L1 Bandwidth . . . 65
4.6.6 Region Sizes . . . 66
4.7 Related Work . . . 68
4.8 Conclusion . . . 70

5 Multi-Kernel Resource Management . . . 74

5.1 Introduction . . . 74
5.2 Background and Motivation . . . 77
5.2.1 GPU Architecture and Multitasking . . . 77
5.2.2 Disadvantages of Temporal and Spatial Partitioning . . . 79

5.2.3 Interference under SMK . . . 79
5.2.4 Opportunities to Control Interference . . . 81
5.3 Overview . . . 82
5.3.1 Online Performance Prediction . . . 83
5.3.2 Dynamic Resource Allocation . . . 83
5.3.3 Hardware Components . . . 84
5.4 Online Performance Prediction . . . 85
5.4.1 Determining Profile Length . . . 86
5.4.2 Detecting Phase Boundaries . . . 87
5.5 Performance Controllers . . . 89
5.5.1 Controlling Warps and Memory Requests . . . 90
5.5.2 Controlling Thread Blocks and Preemption . . . 92
5.5.3 Avoiding Throughput Loss . . . 94
5.6 Evaluation . . . 95
5.6.1 Methodology . . . 95
5.6.2 Hardware Implementation . . . 96
5.6.3 Performance Targets and Throughputs . . . 97
5.6.4 Performance Predictor Accuracy . . . 99
5.6.5 Performance Targets Achieved . . . 100
5.6.6 Cloud Operator Revenue . . . 101
5.7 Related Work . . . 102
5.8 Conclusion . . . 104
6 Conclusion and Future Work . . . 105
6.1 Summary . . . 105
6.2 Future Work . . . 107
Bibliography . . . 109

LIST OF FIGURES

1.1 Percentage of cycles any instruction was issued for a set of workloads from Parboil, Rodinia, and the NVIDIA SDK. 2
1.2 Percentage of cycles the load/store unit was stalled, which indicates when throughput is limited by the global memory system. 2
1.3 Major GPU components involved in data movement and storage. This thesis focuses on three critical data management components: global memory merging and caching in shader cores, register storage, and allocation of resources between multiple co-running workloads. 4

2.1 Design of a GPU core, called a streaming multiprocessor (SM). GPUs are optimized for throughput rather than individual thread latency and execute 32-wide vector instructions. 10
2.2 GPU system design, where SMs and memory partitions are on opposite sides of an interconnect. The GPU communicates with a host system through PCI Express data transfers to and from DRAM. 11

3.1 Diagram of GTX 480 memory system. Each SM has a load/store unit with a memory coalescer which sends requests to a private L1 cache. Requests to shared L2 caches and DRAM partitions are sent over an interconnect. 16
3.2 Memory request reduction for the `spmv` benchmark, showing the number of requests remaining at each level of the memory hierarchy as the coalescer and caches convert locality into fewer requests. 16
3.3 Memory throughput-limited workloads overwhelm the coalescing system either though generating many requests with memory divergence or by causing cache resource shortages by saturating memory bandwidth. The kernel number is its sequence in the kernel execution order in the benchmark. 17
3.4 A portion of the execution of the `GEMM` benchmark. The solid line is the number of memory instructions waiting to be scheduled. The background is grey when the L1 cache resources are full. The bars at the bottom show when arithmetic is scheduled. For the cache resource-bound benchmarks, there is little overlap of computation with cache resource stalls. 17

3.5 Many memory throughput-limited kernels show a large degree of inter-warp spatial locality. Each cell corresponds to a kernel, and inside each cell the window size gets larger from left (baseline window of only intra-warp coalescing) to right (window of 128 requests after intra-warp coalescing). As the window size increases the number of requests that must be sent to the cache decreases. 19
3.6 Diagram of the *WarpPool* system. 22
3.7 A diagram of the inter-warp coalescing queues. Requests exiting the intra-warp coalescers are merged with other requests to the same cache line in these queues. 25
3.8 Common mapping patterns from cache line words to threads 27
3.9 Relative occurrence of mapping patterns in benchmarks 28
3.10 Per-SM area breakdown of *WarpPool* components, with a total area of 0.36 mm^2 per SM. (* = SRAM area calculated using CACTI) 31
3.11 Speedup of GPU with banked cache, MRPB, and *WarpPool* over GTX 480 baseline. . 33
3.12 Average number of requests *WarpPool* coalesced into an L1 cache request, compared against the number of requests an 8-bank cache serviced each cycle. 34
3.13 Number of misses per thousand instructions (MPKI) for MRPB and *WarpPool*, normalized to the baseline. 35
3.14 Relative execution time of matrix transpose versions, normalized to naïve global memory version. 36

4.1 Comparison of GPU register energy reduction techniques that change how execution units (EUs) read operands from the register file (RF) 41
4.2 Average register working set in 100 cycle window for GTO and 2-level warp schedulers in baseline 2048 KB register file for benchmarks in Rodinia [17] 43
4.3 Accesses to the register backing store per 100 cycles during the steady state of hotspot for baseline, RF hierarchy [32] with 8-entry scratchpad, and RegLess with 8 entries per warp 45
4.4 Walkthrough 47
4.5 Count of live registers for a portion of `particle_filter`, with low live register points highlighted 49
4.6 Compiler annotations added on regions and instructions 51
4.7 Determining whether a definition is soft. A soft definition of a register might not kill every thread's values. 53
4.8 Block diagram of RegLess components in each SM 53
4.9 Capacity manager (CM) design. CMs track which warps have registers allocated in the OSU and are ready to execute instructions. 56
4.10 Operand staging unit (OSU) design. OSUs store register values and service register read and writes. 56
4.11 Area for RegLess configurations, normalized to 2048-entry baseline RF 61
4.12 Combined static and average dynamic power for RegLess configurations, normalized to baseline RF 61
4.13 Run time vs. GPU energy for RegLess configurations, normalized to baseline. The line marks the Pareto frontier. 62
4.14 Register file energy for RFV [50], RFV [32], and RegLess, normalized to baseline . . 63

4.15 Normalized total GPU energy, including added instruction and memory accesses. The "No RF" entry is the upper bound for energy savings, which uses the baseline performance and a register file that consumes no energy. 63
4.16 Run time (lower is better) for 512-register RegLess design normalized to baseline with full RF. The geomean is compared with RegLess with no compressor, RFV, and RFH. 64
4.17 Location from which registers were preloaded. 0.9% of registers were preloaded from L1 and 0.013% were preloaded from L2 or DRAM. 65
4.18 Average RegLess L1 requests per cycle 66
4.19 Average number of preloads, average number of concurrent live registers, and standard deviation of number of concurrent live registers per region 66

5.1 Diagram of GPU and SM design. In SMK [141, 146], the warp contexts are split between applications. 77
5.2 Resource demands for GPU workloads (methodology in Section 5.6.1); workloads on the left saturate compute resources, and workloads on the right saturate memory resources. Sharing an SM between complementary workloads increases overall throughput. 78
5.3 Running multiple kernels using SMK results in interference. The SG benchmark is run alongside three other benchmarks, sharing resources evenly. Interference causes throughput loss for one or both workloads. 80
5.4 Timeline of % cycles arithmetic issued and load/store unit stalled, averaged over 100-cycle windows, for a 20,000-cycle interval of BP. A co-running workload is able to issue compute and memory instructions at times of low utilization. 80
5.5 Overview of the Scavenger system 82
5.6 The Scavenger components in each SM, which use performance counters to determine resource allocations. The upper components (in orange) predict the primary workload's performance and detect when it has entered a new phase. The lower components (in blue) adjust the resource allocation to achieve the primary workload target and maximize secondary workload throughput. 84
5.7 Performance of excerpt of HS over time along with the difference between the mean training and validation interval IPCs. To detect an appropriate profile length, Scavenger continues profiling until the difference stays below a threshold. 86
5.8 Feedback control system for active warps and outstanding memory requests. The short-term and long-term difference between the predicted IPC and actual IPC is used to adjust resource allocations using PID controllers. 90
5.9 The thread block controller preempts blocks to balance warp slack between the workloads. (a) Scavenger detects too little slack for primary workload, too much for secondary. (b) Blocks of secondary workload are preempted to make way for primary. (c) Slack is more evenly distributed between workloads. 93
5.10 Secondary workload throughput with Scavenger compared to temporal partitioning, by primary x secondary workload category. Pairs violating the primary kernel performance target are excluded. 97
5.11 Geomean primary workload throughput by pair category and target 97
5.12 Total primary and secondary throughput with Scavenger compared to temporal partitioning. Pairs violating the primary kernel performance target are excluded. 98

5.13 Cumulative % error and % time profiling, averaged across benchmarks vs. maximum difference between training and validation interval IPC while profiling 99
5.14 Percentage of pairs where the primary kernel's performance was below the target with 1% error margin, and the average percentage by which pairs that did not achieve the target were below the target performance 100
5.15 Additional revenue per dollar realizable with Scavenger over leasing GPUs as a unit or using temporal partitioning. 101

LIST OF TABLES

3.1 Per-SM storage and power overhead of *WarpPool* components 30
3.2 Resource configuration evaluated 31

4.1 GPGPU-sim simulation parameters 60
4.2 Average number of static instructions per region and average dynamic cycles per region 67

5.1 Performance predictor parameters 89
5.2 Controller parameters 94
5.3 Simulator configuration 94
5.4 Benchmarks 95

ABSTRACT

Graphics Processing Units (GPUs) are becoming common in data centers for tasks like neural network training and image processing due to their high performance and efficiency. GPUs maintain high throughput by running thousands of threads simultaneously, issuing instructions from ready threads to hide latency in others that are stalled. While this is effective for keeping the arithmetic units busy, the challenge in GPU design is moving the data for computation at the same high rate. Any inefficiency in data movement and storage will compromise the throughput and energy efficiency of the system.

Since energy consumption and cooling make up a large part of the cost of provisioning and running and a data center, making GPUs more suitable for this environment requires removing the bottlenecks and overheads that limit their efficiency. The performance of GPU workloads is often limited by the throughput of the memory resources inside each GPU core, and though many of the power-hungry structures in CPUs are not found in GPU designs, there is overhead for storing each thread's state. When sharing a GPU between workloads, contention for resources also causes interference and slowdown.

This thesis develops techniques to manage and streamline the data movement and storage resources in GPUs in each of these places. The first part of this thesis resolves data movement restrictions inside each GPU core. The GPU memory system is optimized for sequential accesses, but many workloads load data in irregular or transposed patterns that cause a throughput bottleneck even when all loads are cache hits. This work identifies and leverages opportunities to merge requests across threads before sending them to the cache. While requests are waiting for merges, they can be reordered to achieve a higher cache hit rate. These methods yielded a 38% speedup for memory throughput limited workloads.

Another opportunity for optimization is found in the register file. Since it must store the registers for thousands of active threads, it is the largest on-chip data storage structure on a GPU. The second work in this thesis replaces the register file with a smaller, more energy-efficient register buffer. Compiler directives allow the GPU to know ahead of time which registers will be accessed, allowing the hardware to store only the registers that will be imminently accessed in the buffer, with the rest moved to main memory. This technique reduced total GPU energy by 11%.

Finally, in a data center, many different applications will be launching GPU jobs, and just as multiple processes can share the same CPU to increase its utilization, running multiple workloads on the same GPU can increase its overall throughput. However, co-runners interfere with each other in unpredictable ways, especially when sharing memory resources. The final part of this thesis controls this interference, allowing a GPU to be shared between two tiers of workloads: one tier with a high performance target and another suitable for batch jobs without deadlines. At a 90% performance target, this technique increased GPU throughput by 9.3%.

GPUs' high efficiency and performance makes them a valuable accelerator in the data center. The contributions in this thesis further increase their efficiency by removing data movement and storage overheads and unlock additional performance by enabling resources to be shared between workloads while controlling interference.

CHAPTER 1

Introduction

Graphics Processing Units (GPUs), although originally designed for accelerating 3D graphics in desktop computers, have proven effective at other tasks that require high computational throughput such as simulations, artificial intelligence, and image processing. Because of this, general purpose GPUs are found not just in desktop PCs but in mobile devices and servers as well. GPUs are optimized for throughput, using a highly multithreaded architecture that switches between threads to hide stalls; this design allows them to be more efficient than CPUs at many tasks. For comparison, one current GPU, NVIDIA's Tesla P100, can achieve 10 single-precision teraflops at 33 gigaflops/watt [44], compared to a large 24-core CPU, Intel's Xeon E7-8870, that achieves 96 gigaflops at 7 gigaflops/watt [48].

In data centers, GPUs are used to complete tasks that need more performance than CPUs can offer. GPUs are used in Google and Facebook data centers for artificial intelligence applications, such as training neural networks [25, 72]. Databases accelerated by GPUs enable interactive analysis of very large data sets [91, 10]. GPUs are not only useful for large companies – public cloud providers are responding to demand by offering virtual machines connected to GPUs. Amazon offers instances with GPUs [11], and Google's public cloud uses a PCIe switch system to connect GPUs to any of its virtual machines [42].

The design constraints for a data center GPU differ from those for a desktop graphics card. Energy efficiency is more important, as a large fraction of data center costs come from power consumption and cooling [12]. To help achieve this, recent data center GPUs include accelerators

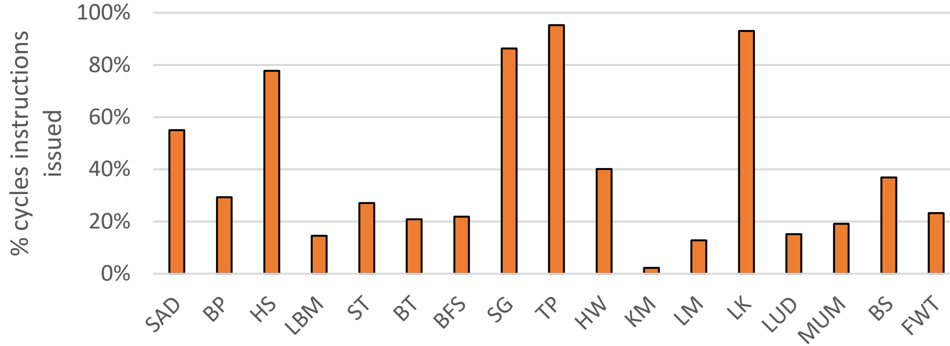


Figure 1.1: Percentage of cycles any instruction was issued for a set of workloads from Parboil, Rodinia, and the NVIDIA SDK.

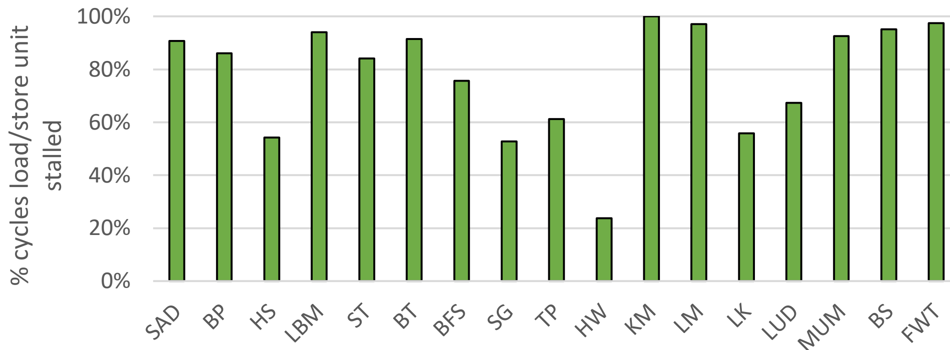


Figure 1.2: Percentage of cycles the load/store unit was stalled, which indicates when throughput is limited by the global memory system.

for matrix operations and support for lower-precision floating-point calculations [101]. Also, in a data center, multiple applications or users are scheduled on the same physical hardware to increase utilization. Data center GPUs now include ways to share the same hardware between multiple tasks and users: current hardware can expose multiple virtual devices per physical GPU that share hardware with temporal partitioning [45], and other work has examined how to add memory virtualization and protection to GPUs [15].

GPUs in mobile devices must be designed with similar constraints in mind. Energy efficiency is visible to mobile users, as it determines battery life. Performance is also vital – mobile GPUs have the opportunity to enable new types of applications that require heavy computation, but in order for developers to justify the additional programming effort, the performance and efficiency difference must be compelling. Sharing hardware between tasks is also important on mobile GPUs, as users

expect smooth graphics even while other types of computation, such as neural network inference, are being run on the GPU [6].

Better reaching these goals of improved energy efficiency and support for sharing hardware between multiple tasks requires focusing on data movement and storage. GPUs achieve their performance by having many arithmetic units, but to keep them utilized, they must be supported by a memory and register system that has equally high performance. Any data bottleneck in the system leads to underutilization of computation resources, stalls, and lower efficiency. Adding more arithmetic resources will only show performance gains if data movement capacity scales in step. However, even current designs have difficulty keeping up with memory throughput demands. Figure 1.1 shows the percentage of cycles that any instructions were issued on a set of GPU workloads, which is often below 50%. The major bottleneck often is the global memory system, as Figure 1.2 demonstrates by showing the load/store unit, responsible for interfacing with the global memory system, is often stalled.

Alongside this performance bottleneck, there are also energy overheads in the GPU due to data storage. Part of what makes GPUs efficient are strategies, such as grouping instructions into SIMD vectors, that reduce the overheads required in more general-purpose processors like CPUs. Although these strategies work for reducing computation overhead, the GPU execution model that interleaves threads does not allow for similar straightforward ways to reduce data storage overhead in the register file and scratchpad memories, since the data to be computed on must always stay accessible should the instructions using it be scheduled.

This thesis manages data movement and storage resources in a way that improves energy efficiency and makes GPUs more amenable to a shared data center environment. Intelligent management of data resources is possible because GPUs have options between which work to schedule and flexibility in which order to schedule it. Because GPUs run many threads at once, there are options between which which threads to run, allowing the GPU to make priority decisions between threads and workloads at a very fine granularity. There are also few ordering constraints between threads, which gives the freedom to reorder and merge memory requests and start and stop portions

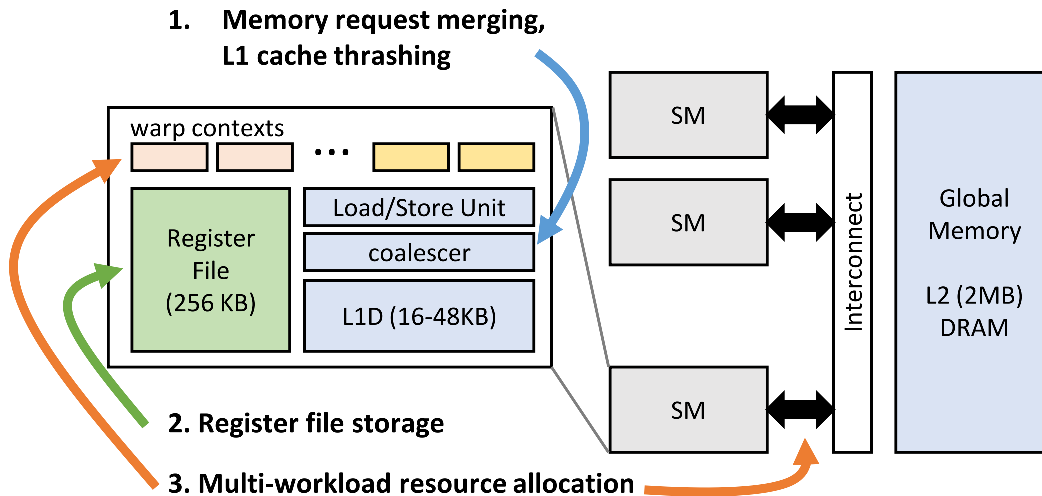


Figure 1.3: Major GPU components involved in data movement and storage. This thesis focuses on three critical data management components: global memory merging and caching in shader cores, register storage, and allocation of resources between multiple co-running workloads.

of workloads to adjust resources. This thesis uses these opportunities to unlock additional energy efficiency and performance for data center and mobile GPUs.

1.1 Data Management Inefficiencies

There are three places where the management of data resources on a GPU is especially critical, shown in Figure 1.3. The first is inside each of the 16 GPU shader cores, which have their own load-store units and L1 cache. Even when running a single workload, the load-store units and L1 cache often do not have sufficient throughput due to memory divergence and cache thrashing. The second is also inside the core, at the register file, which is very large as it is provisioned to hold every register with similar access times. The third is the allocation of thread contexts in the cores between workloads and the number of memory requests each workload can send to the global memory system. This section will look at each of these places in turn.

1.1.1 Memory Divergence and Cache Thrashing

One data management inefficiency comes from the GPU processing cores being unable to issue enough requests to the memory system. GPUs use SIMD vector instructions, including loads and stores, that execute 32 lanes in parallel. This enables each lane in a vector load or store to reference a different address. Often, the addresses in each lane will be consecutive words in the same cache line, in which case loading that single cache line will be enough to complete the vector load. However, when the lanes reference multiple cache lines, called *memory divergence*, multiple requests must be sent to the memory system to satisfy the one load instruction. In the worst case, one load requires 32 requests, creating a throughput bottleneck.

A related inefficiency comes from cache thrashing. Because GPU cores run many threads simultaneously but have a small (32KB) L1 data cache, each thread can only store on the order of bytes in the cache before evicting another thread's data. While techniques to reduce thrashing to improve performance are well-studied in previous work [117, 123, 51], thrashing also blocks requests from leaving the core for the rest of the memory system by filling up MSHRs and using scarce bandwidth to the L2 and DRAM. Therefore, inefficient use of the data cache not only causes increased latency but also lower memory throughput.

1.1.2 Register File Energy Overhead

The register file is the largest data storage structure in a GPU and a source of energy overhead. GPUs' multithreaded architecture means that the register file needs to store the registers for every thread. Because so many threads are active on each core, the register file is very large — 256KB per core in recent designs. It also must support many accesses per cycle, as one vector instruction may have three 128-byte input registers and produce a 128-byte output. This makes the register file expensive to access, consuming up to 15% of total GPU power [77]. Since the register file only provides value to the GPU as a support for computation, this power represents an overhead in the GPU design that should be minimized.

1.1.3 Inter-Application Contention

GPU kernels consist of many threads, but each thread is identical to the others. This means the resource demands for each thread are very similar, and a kernel while executing will tend to saturate one resource and underutilize others. Running multiple workloads simultaneously on the same GPU allows the workloads to saturate different resources, leading to a throughput boost. However, during times when the workloads require the same resources, they interfere with each other. This interference leads to some workloads choking others' performance and other unpredictable effects. In a data center or public cloud environment, controlling this interference will be necessary to harness the throughput boost of sharing the GPU while providing performance targets that are useful to customers.

1.2 Contributions

This thesis addresses these inefficiencies through intelligent management of data resources. Chapter 3 describes a memory merging and prioritization system that increases memory throughput inside a GPU core. Chapter 4 details a register file replacement that requires much less storage and access energy while not impacting performance. Chapter 5 develops a system for controlling the interference between kernels co-running on the same GPU, creating two tiers of service. Together, these pieces attack the sources of energy inefficiency and low utilization in the GPU data movement and storage systems.

1.2.1 Increasing Memory Throughput

Existing GPUs have a memory coalescer which merges requests to the same cache line in a vector load instruction, as often each lane in a vector load accesses a different word in the same cache line. This type of spatial locality, where nearby threads access nearby data, is also present between lanes in different warps, but ignored by current hardware. By exploiting this locality, requests can be merged across warps before those requests are sent to the cache. This helps mitigate throughput

loss due to memory divergence because effectively more than one request per cycle is being sent to the cache. The same hardware used for merging requests can also be used to reorder requests to put loads from the same warps closer together in time, increasing memory throughput by increasing the L1 hit rate. Chapter 3 introduces the *WarpPool* system that finds these new merging opportunities and queues memory requests to issue them in a more cache-friendly order, resulting in a 38% speedup via an 8% increase in merges and a 23% reduction in L1 cache misses.

1.2.2 Reducing Register File Energy and Storage

Instead of having enough storage for every register, this thesis replaces the register file with a staging unit which stores only the registers that will be imminently used by a small set of active warps. Most registers have a short lifetime and only need a temporary allocation in the staging unit. The few long-lived registers can be evicted from the staging unit to the L1 cache when they will not be accessed. Compiler annotations enable this system, as they allow register usage information gathered with static analysis to be visible by the hardware at run time to make allocations in the staging buffer and transfer values in just as registers are about to be accessed. In Chapter 4, the *RegLess* system implements this compiler-guided register buffer, reducing register storage to 25% of its original size with no average-case performance loss.

1.2.3 Controlling Interference in Shared GPUs

Public cloud operators like Amazon's AWS and Google's GCP have multiple tiers of service in order to increase utilization. For example, *spot* instances are sold to customers at a discount to fill unused capacity, with the condition that they may be preempted at any time when that capacity is needed. In Chapter 5, the *Scavenger* system controls the interference caused by sharing a GPU between multiple workloads to create similar tiers: one tier with a high performance target for customers requiring maximum throughput, and a second lower-performance tier for batch jobs. Because sharing the GPU will create a throughput surplus, the cloud operator can either operate less physical hardware or capture the excess throughput as profit. Scavenger increases the

batch workload throughput by 1.35x compared to temporal partitioning while maintaining a 90% performance target for the high-performance tier, increasing overall GPU throughput by 9.3%.

CHAPTER 2

Background

This thesis optimizes data movement and storage on graphics processing units (GPUs), throughput processors that operate on thousands of threads in parallel. Because each GPU thread is independent, instructions can be issued from any available thread, with the goal of keeping the functional units utilized. GPU kernels are complete when every thread finishes, so maintaining maximum utilization and throughput will finish the kernel in the minimum time possible. GPUs are optimized for running workloads consisting of many identical threads, setting them apart from other throughput-oriented processors like Niagara [68], because their original design was optimized for graphics vertex and fragment shaders, where the same code executes on every vertex or pixel. This chapter describes the baseline GPU architecture that the subsequent techniques in this thesis build upon.

2.1 SM Design

Figure 2.1 shows the design of a GPU core, or streaming multiprocessor (SM). Each hardware thread, called a *warp*, is allocated a *warp context*. As one of the optimizations for running many identical threads, warps issue 32-wide SIMD instructions. The warp contexts track the current PC for each warp and can activate and deactivate individual threads in a warp to implement conditional branches. The *warp scheduler* sends instructions to the functional units, selecting between ready warps; it is divided into several independent schedulers to allow more than one warp to be issued per cycle. The *register file* on a GPU is very large, in the hundreds of kilobytes per

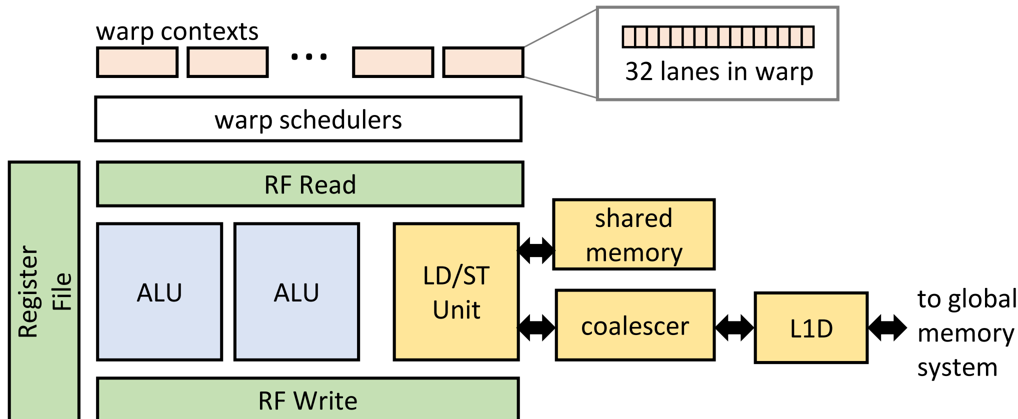


Figure 2.1: Design of a GPU core, called a streaming multiprocessor (SM). GPUs are optimized for throughput rather than individual thread latency and execute 32-wide vector instructions.

SM, because each thread in each warp is allocated its own register context. The functional units include floating-point ALUs and a load/store unit. The load/store unit handles requests to the different GPU memory spaces, including the *shared memory* scratchpad and *global memory*, which is backed by data caches and DRAM. Global memory requests are sent through a *coalescer*, which merges memory requests made by different threads in the same warp, and then are sent to an L1 data cache. The effectiveness of the L1 cache depends heavily on the workload, so some GPU designs disable it by default. In this thesis, Chapter 3 is built on a GPU where the L1 cache is enabled by default, and Chapters 4 and 5 model one where the L1 does not cache requests by default.

2.2 Memory System Design

The overall GPU system design, encompassing the SMs and the global memory system, is shown in Figure 2.2. The SMs lie on one side of the interconnect with shards of L2 cache and DRAM on the other. Neither the SMs or the L2 shards communicate with other units on their side of the interconnect, as the tasks on one SM are independent of those on another, and the L2 shards are partitioned by address. Each L2 shard includes a memory controller which communicates with one of the two channels exposed by each GDDR DRAM chip. There is significant latency communicating to L2, as graphics ROPs sit in front of the L2, so even L2 hits have high latency.

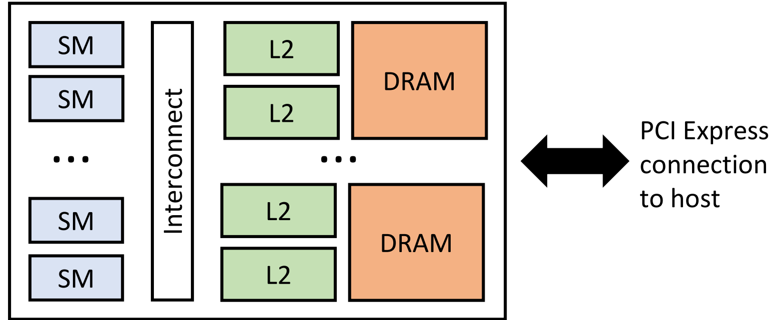


Figure 2.2: GPU system design, where SMs and memory partitions are on opposite sides of an interconnect. The GPU communicates with a host system through PCI Express data transfers to and from DRAM.

This is not an issue in the GPU design because independent instructions from other threads can hide the latency in any individual thread. Sustaining high bandwidth is more important – the L2 cache is more important for reducing duplicate requests than latency.

Data is transferred to and from the GPU though a PCI Express bus. There is a very large bandwidth differential between DRAM and PCI Express, with the DRAM able to supply 224 GB/s in the GPU models modelled in this thesis, whereas PCI Express 3.0 x16 has a limit of 16 GB/s. In this thesis, the experiments assume that the data is already present in DRAM. Complementary work [39, 19, 20] has examined how to overlap computation with memory transfers to reduce the overall task turnaround time.

2.3 Programming Model

General purpose GPU applications are typically implemented in languages like CUDA [100] and OpenCL [127], which are extensions to C++. In the programming model these use, programmers write code from the perspective of one thread. Although each thread must largely be independent, the programming model groups threads into *blocks*. The threads in a block are allocated the same region of scratchpad shared memory and can synchronize using barriers. In hardware, this requires that the warps in a block must run on the same SM and have their resources allocated as a group.

2.4 Design Convergence in Desktop, Data Center, and Mobile

There is increased convergence in GPU designs found in desktop PCs, servers, and mobile devices. For example, NVIDIA's Pascal architecture was used in GeForce gaming GPUs, Tesla compute accelerators, and integrated in a mobile SoC used in the Jetson TX1 board. The same design is scalable to these different environments by varying the number of SMs – the GTX 1080 has 20, the Tesla P100 has 60, and the Jetson TX1 has 2. This convergence means that techniques optimizing the GPU hardware in one setting can be applicable to the others, and that the same programming model and kernels can be used in each of these different settings. In this thesis, desktop GPU models like the GTX 480 and 980 are used for simulation, but because the techniques respect the loose coupling between SMs, the techniques can be extended to server GPUs with larger number of SMs and mobile GPUs with fewer without scalability issues.

CHAPTER 3

Inter-Warp Memory Request Merging and Prioritization

3.1 Introduction

GPUs are throughput processors designed to hide memory latency using multithreading. During the time some threads on the GPU are waiting for long-latency memory operations, others can be scheduled to do computation. However, in order for this strategy to keep the utilization of the arithmetic units high, data needs to come from the memory system fast enough to maintain a set of threads that are ready to do computation. Therefore, keeping the arithmetic units utilized depends on the throughput of the memory system matching the throughput of the compute pipelines.

However, previous studies have shown that for many benchmarks, the throughput of the global memory system is not adequate to keep the GPU from stalling. This can be due to saturated DRAM bandwidth [124], limited L1 cache resources such as MSHRs and cache sets [51], or small per-thread cache capacity [118]. To achieve high throughput despite bandwidth limitations in the global memory system, the GPU has a memory hierarchy that merges requests to the same cache lines to reduce traffic.

On a GPU, threads in a 32-wide warp execute in lockstep, but the threads in one load can generate accesses to many different cache lines. A *memory coalescer* is the first unit in the memory hierarchy, responsible for combining memory accesses to the same cache line made by the 32 threads in a warp. This unit is effective because spatial locality is expressed in a GPU through

nearby threads accessing the same cache lines [46]. Combining requests early in the memory pipeline is better for performance and energy efficiency than sending duplicate requests to the higher-level caches and DRAM.

However, the coalescer can become the bottleneck in memory system throughput, which happens under memory divergence, where the threads in a warp request more than one cache line in a load or store instruction. Because the L1 can only service one request per cycle, up to 32 requests must be serialized over 32 cycles instead of being serviced simultaneously. Another throughput problem is caused by limited cache resources, where the memory system stalls when the cache cannot allocate a resource like an MSHR to issue another outstanding miss. In both of these cases, the GPU becomes underutilized because the memory system cannot supply data fast enough to keep the arithmetic units busy.

The current memory coalescer is limited to merging requests between threads in the same warp. However, we show that spatial locality is not limited to threads in the same warp, so allowing the coalescer to merge requests from threads in multiple warps would allow for for a greater reduction in requests. If the coalescer were able to merge requests across warps using this inter-warp spatial locality before they reach the L1 cache, it would increase the effective bandwidth of the cache by relieving the one access per cycle bottleneck. During times when L1 resources are at a premium, it would enable resources to service requests from as many warps as possible. As well, having requests from multiple warps in scope allows the coalescer to act as a gatekeeper to the L1 cache and reduce thrashing.

We propose a novel memory coalescer, *WarpPool*, which is able to find inter-warp spatial locality. It increases the effective throughput to the L1, uses cache resources more efficiently, and reduces cache thrashing by prioritizing some warps' access to the cache. After a first level of coalescing to find intra-warp spatial locality, requests are inserted into inter-warp coalescing queues that merge requests from multiple warps. Doing both intra-warp and inter-warp coalescing reduces the number of requests that need to be made to the L1 cache. Because the requests exiting the coalescer now fetch data for more than one load, more than one load's requests can enter the coalescer

per cycle, which will keep throughput high under memory divergence. When cache resources are scarce, requests will build up in the inter-warp queues, which will increase the amount of inter-warp coalescing and enable requests using cache resources to service multiple loads. Furthermore, requests from the inter-warp queues can be selected to exit to the cache in an order that enhances temporal locality in the cache.

In this work, we make the following contributions:

1. We characterize a class of inter-warp spatial locality that current coalescers are unable to capture. We show that using this locality to merge requests would remove the bottleneck in a class of workloads limited by memory system throughput.

2. We propose *WarpPool*, an inter-warp memory coalescer that is able to merge requests between warps to convert this locality into increased bandwidth to the L1 cache and more efficient utilization of cache resources. It is also able to prioritize warps' access to the L1 cache, which reduces cache thrashing.

3. We implement *WarpPool* in GPGPU-sim [8] and achieve a 38% geometric mean speedup across a set of memory throughput-limited kernels. *WarpPool* increases the throughput to the L1 cache by 8% and reduces the number of L1 misses by 23%.

4. We evaluate a case study demonstrating that *WarpPool* improves GPU programmability by achieving a 2.0× speedup on straightforward code for which manual optimizations give a 2.6× speedup.

3.2 Background and Motivation

3.2.1 Background

GPUs are made up of multiple streaming multiprocessors (SMs), in Nvidia terminology. Inside each SM, warp schedulers select threads with ready operands and issue them to functional units.

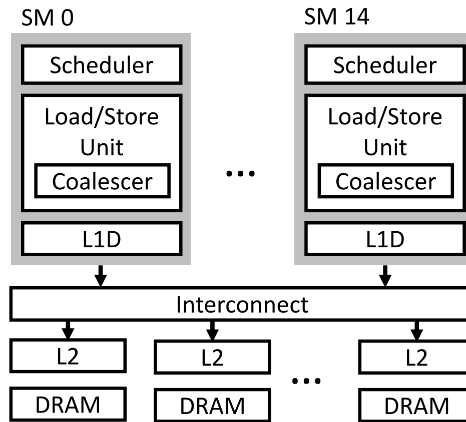


Figure 3.1: Diagram of GTX 480 memory system. Each SM has a load/store unit with a memory coalescer which sends requests to a private L1 cache. Requests to shared L2 caches and DRAM partitions are sent over an interconnect.

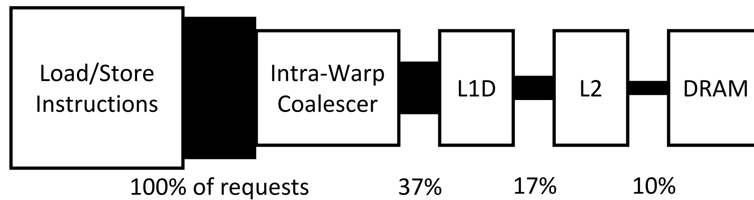


Figure 3.2: Memory request reduction for the `spmv` benchmark, showing the number of requests remaining at each level of the memory hierarchy as the coalescer and caches convert locality into fewer requests.

Threads are scheduled as a group of 32, called a warp. Threads in a warp execute the same instructions in lockstep, but can supply different values as inputs to those instructions.

The load/store unit (LSU) is the functional unit responsible for loads, stores, and memory barrier instructions. Like the other functional units, it is scheduled a warp of 32 threads at at time. There are multiple memory spaces in the GPU, and some, like the *shared* scratchpad memory, have enough throughput to service a different request from each of the 32 threads in a warp each cycle. However, all the data sent to the GPU for computation needs to be loaded from global memory, which can only process one request per cycle, and the final results of the GPUs computations also need to be stored in global memory for transfer back to the CPU. Global memory is backed by DRAM and implemented using a cache hierarchy similar to a CPU memory system, as shown in Figure 3.1.

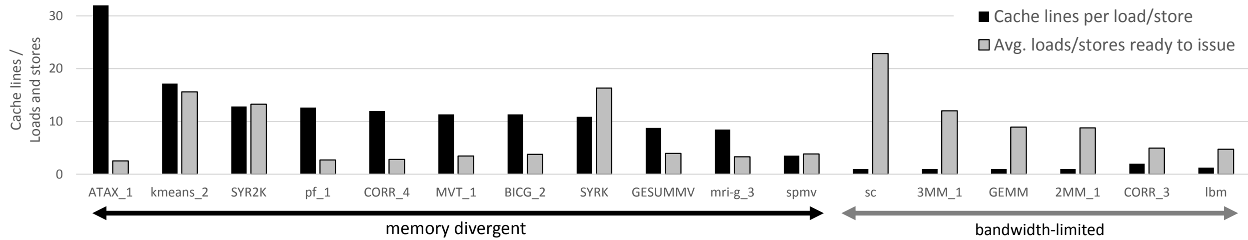


Figure 3.3: Memory throughput-limited workloads overwhelm the coalescing system either though generating many requests with memory divergence or by causing cache resource shortages by saturating memory bandwidth. The kernel number is its sequence in the kernel execution order in the benchmark.

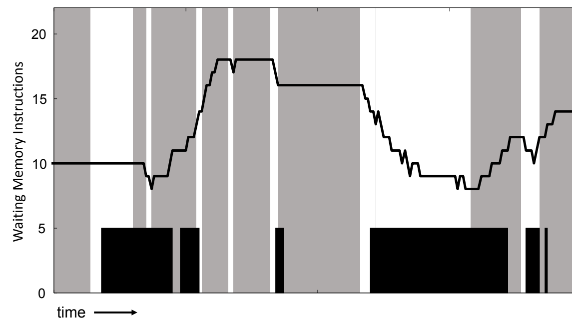


Figure 3.4: A portion of the execution of the `GEMM` benchmark. The solid line is the number of memory instructions waiting to be scheduled. The background is grey when the L1 cache resources are full. The bars at the bottom show when arithmetic is scheduled. For the cache resource-bound benchmarks, there is little overlap of computation with cache resource stalls.

The memory coalescer conserves bandwidth by merging requests to the same cache line made by threads in a warp, taking advantage of spatial locality between the threads in a warp to reduce the number of requests. When the warp scheduler sends a warp to the LSU, a load or store instruction contains 32 addresses, one for each lane in the warp. The memory coalescer determines which of the 32 addresses point to the same cache line and merges requests to the same line together. The coalescer is effective at reducing requests because spatial locality is often expressed in a GPU as nearby threads requesting nearby data.

Because there is high demand on the global memory system for limited bandwidth, each stage of the memory system is designed to conserve bandwidth by using locality to merge requests. Figure 3.2 shows the units in the GPU memory pipeline along with the percentage of requests that make it through each stage for the `spmv` benchmark, representative of a set of memory divergent

benchmarks detailed in Section 3.2.2. The memory coalescer reduces the number of requests by over 40%, only 17% of requests get past the L1 to the L2, and 10% of requests reach DRAM. Since the bandwidth decreases as requests go further down the pipeline and the energy cost to make an access increases at each stage, it is advantagous to merge requests as early as possible.

Since GPU cache lines are 128 bytes, designed so that each of 32 threads in a warp can request a 4-byte word, all 32 requests in many loads and stores map to a single cache line. However, each request may map to a number of different cache lines, called *memory divergence*. Under memory divergence, the coalescer is unable to reduce the number of requests, and must make up to 32 serialized requests to the L1 cache in the worst case. The L1 is only able to service one request per cycle, so a divergent load or store takes one cycle per distinct cache line to complete.

3.2.2 Oversubscription of L1 Bandwidth

Although the interface between the LSU and the L1 cache is the fastest link in the global memory system, it is the link that must accept the largest number of requests relative to its output throughput (32 to 1). The memory coalescer is responsible for matching the large throughput in to the small throughput out. However, under two common scenarios it is not able to do so.

Figure 3.3 shows the average number of cache lines per memory operation and the average number of waiting memory operations for kernels from the Parboil [128], Rodinia [18], and PolyBench [37] benchmark suites. These kernels were selected because they had more memory instructions ready to issue than the LSU could process for over 90% of their execution time. These instructions could be executed by the LSU if the memory system had higher throughput, so these are the workloads for which improving the throughput of the memory system has the potential to improve performance.

The first category of workloads are **memory divergent**, where the intra-warp coalescer requires an average of 13 cycles to send a warp's load or store instruction to the L1 because each thread requested a different cache line. This can be as high as 32 for `ATAX_1`. The L1 cache needs to service up to 32 times as many cache line requests for these workloads per load or store instruction,

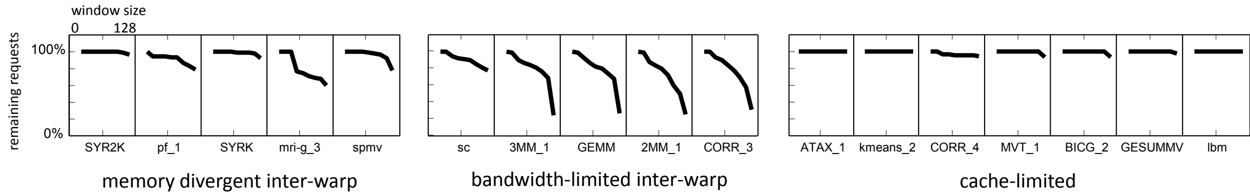


Figure 3.5: Many memory throughput-limited kernels show a large degree of inter-warp spatial locality. Each cell corresponds to a kernel, and inside each cell the window size gets larger from left (baseline window of only intra-warp coalescing) to right (window of 128 requests after intra-warp coalescing). As the window size increases the number of requests that must be sent to the cache decreases.

creating a bandwidth bottleneck at the L1 cache. The baseline memory coalescer is unable to merge many requests for these workloads, so it is unable to handle the bandwidth demand. A better coalescer would be able to reduce the effective level of divergence by finding locality between multiple divergent loads before sending them to the L1 cache.

The second category are **bandwidth-limited**. These workloads have low memory divergence, but due to high miss rate or high memory intensity saturate DRAM bandwidth and cause the L1 cache to run out of resources like MSHRs. Figure 3.4 shows how the execution of the `GEMM` benchmark, with 8.9 average waiting memory operations, follows a cyclical pattern. Most of the time, the cache has no resources to accept a new request, which causes ready memory instructions to back up and no computation to be done. When data comes back from the higher levels of the memory system, arithmetic begins to execute and new memory instructions are issued until the cache stalls again. A better coalescer would be able to make better use of limited cache resources by continuing to accept requests while the L1 cache's resources are full, using the time the cache is stalled to reduce the number of requests, and then making better use of the time the cache is accepting new requests by having each request service multiple load or store instructions. This way, more arithmetic can be done per request the L1 is able to service.

The benefits of a coalescer that is able to merge more requests propagate down to the rest of the memory system, where performing the merging is more expensive. The more requests that can be removed early in the pipeline, the fewer requests the later stages in the pipeline need to service.

3.2.3 Increasing Coalescing Window Size

The baseline coalescer can only merge requests between threads in one warp. To find more opportunities, the coalescer needs to look between requests made by different warps across multiple load instructions, increasing its window from the threads in one warp to requests from multiple warps.

Figure 3.5 shows the relative reduction in memory requests made by the memory throughput-limited kernels that could be achieved by increasing the coalescers window size to multiple warps, analyzed using a trace of global memory requests made to the L1. The window size increases from 0, equivalent to doing only intra-warp coalescing, to a window of 128 cache line requests made to the L1. The kernels without inter-warp locality have been split into a new category of **cache-limited** kernels that exhibit intra-warp temporal locality.

The **memory divergent inter-warp** workloads show inter-warp locality with larger window sizes. Because divergence creates many requests, locality begins to show up only at larger window sizes: if each load generates 32 requests, the window size of 128 is a window of 4 load instructions. Patterns like indirect accesses (`spmv`) and large memory strides (`SYRK`) create divergence and inter-warp locality. For these workloads, the spatial locality can only be found by having a larger window size than one warp, because the only spatial locality is between warps.

The **bandwidth-limited inter-warp** workloads exhibit a high degree of inter-warp locality. The locality is caused by the memory access patterns in these workloads, such as accessing a matrix column-wise and row-wise, but contiguously inside a warp (`GEMM`) or repetition of the same accesses in an inner loop across all threads (`streamcluster`). In these workloads, there is spatial locality both within warps and between warps. A coalescer with a larger window size will be able to find more coalescing opportunities in them.

A third category, **cache-sensitive** workloads, have low inter-warp spatial locality but exhibit temporal locality inside warps. Much previous work has focused on this category of workloads, by limiting access to the cache through scheduling [118] or bypassing [51]. A coalescer with a larger window size will not be able to find more opportunity to merge requests, but it will have more scope to prioritize requests. It will be able to choose a request from among multiple warps to

send to the cache, using that ability to prioritize certain warps' access to the cache.

Therefore, by increasing the window size in which the coalescer can merge requests together to include requests made by different warps, the coalescer is able to increase effective bandwidth to the L1 cache and stop this link in the memory system from becoming the bottleneck for the throughput of both the memory system and the entire GPU. A coalescer able to merge across warps will be able find spatial locality that is out of the scope of the current intra-warp coalescer, and has the opportunity to help even workloads without spatial locality.

To turn better coalescing into speedup, the improved coalescer will need to address the reasons why L1 bandwidth limits performance for each category of workload. For **memory divergent inter-warp** workloads, the coalescer will need to serialize divergent memory operations from more than one warp in parallel. For the **bandwidth-limited inter-warp** workloads, the coalescer will need to buffer requests received when cache resources are full. For the **cache-limited** workloads, it will need to leverage the coalescing window to schedule requests in a way that reduces cache thrashing.

3.3 WarpPool Design

3.3.1 Overview

The *WarpPool* system creates a window in which requests from multiple warps can be coalesced, in order to capture inter-warp spatial locality. Requests are inserted into this window after intra-warp coalescing, and requests removed from the window are sent to the L1 cache. In order for the inter-warp coalescing window to yield speedup, it needs to be supported by an intra-warp coalescing pipeline in the front end that can insert requests into the window at the same rate as they drain out. On the other end of the pipeline, selecting requests from the window to send to the cache needs to be done in a way that preserves intra-warp temporal locality, since reordering memory requests can easily cause cache thrashing.

A high-throughput intra-warp coalescing pipeline, a window to capture inter-warp spatial local-

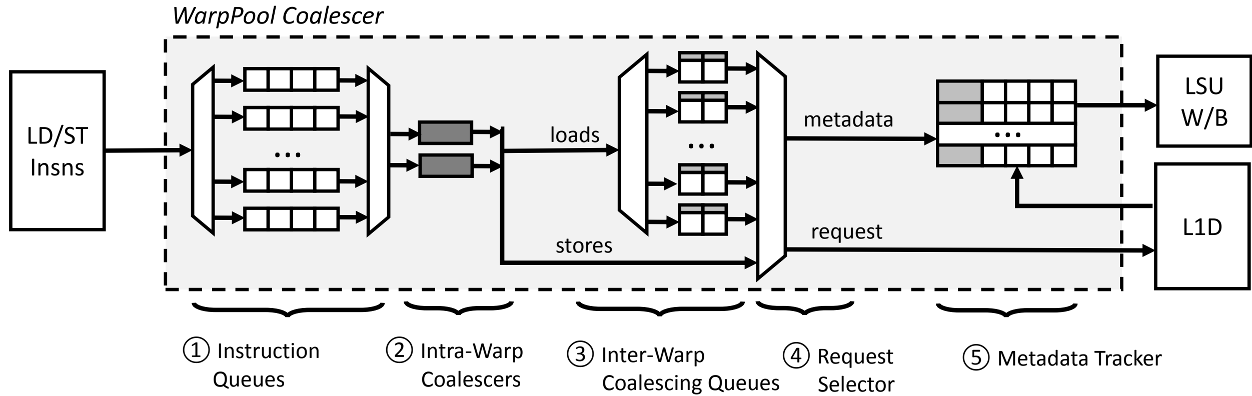


Figure 3.6: Diagram of the *WarpPool* system.

ity, and a selection policy that preserves intra-warp locality make up the substance of the *WarpPool* system, shown in Figure 3.6. Instruction queues ① hold load and store instructions issued by the scheduler, to prioritize access to the coalescer and cache. These issue into the intra-warp coalescers ②, which merge requests to the same cache line inside a warp. Inter-warp coalescing queues ③ combine requests between warps to find new inter-warp coalescing opportunities. Then, the request selector ④ determines which requests exit the inter-warp queues in a way that maximizes coalesces while maintaining intra-warp spatial locality.

In the first stage, *WarpPool* queues load and store instructions. Loads and stores are inserted into a queue ① based on which warp they were issued from, which allows *WarpPool* to prioritize some warps' access to the intra-warp coalescers. In the next stage, *WarpPool* uses multiple intra-warp coalescers ② to capture intra-warp spatial locality. These coalescers are identical to the baseline intra-warp coalescer, but there are more of them so that multiple divergent loads and stores can be serialized in parallel.

After intra-warp coalescing, *WarpPool* captures spatial locality between threads from different warps using inter-warp coalescing queues ③. Requests are mapped to a coalescing queue based on their address, similar to the way that requests are mapped to cache sets. Requests for the same cache line from different load instructions are matched against each other and merged into one request to be sent to the cache.

Requests stay in the queues until sent by a selector to the L1 cache ④. The order that requests

are sent to the cache is crucial for maintaining the intra-warp temporal locality exhibited by many benchmarks. *WarpPool* leverages having coalescing window containing many requests to schedule requests in a way that maximizes the number of coalesces and prioritizes access to the L1.

After loads return from the L1, the data from the cache line needs to be written to the registers of the threads that requested the line. *WarpPool* maintains metadata about the mapping of words in the cache lines to threads in a load (stage ⑤ in Figure 3.6), which allows a request made on behalf of multiple loads to be de-coalesced and written back to the correct registers. By taking advantage of common mapping patterns, this metadata can be kept to a manageable size, as explained in Section 3.3.6. *WarpPool* uses the crossbar already present in the GPU load-store unit to move the data from the cache line to the threads for writeback. Although *WarpPool* adds more stages to the GPU's memory pipeline[1], there are enough warps to hide this added latency with multithreading.

In the following sections, each part of the *WarpPool* system is described in greater detail. This is followed by a discussion of how metadata mapping data to threads is stored, how stores are handled by the system, and how memory consistency is maintained even as loads and stores are reordered in the coalescing queues.

3.3.2 Instruction Queues

Queues at the front of the pipeline allow *WarpPool* to prioritize access to the coalescing resources, which improves cache locality. These queues hold load and store instructions before address generation, so as to avoid storing 128 bytes of addresses. Loads and stores are mapped to one of these queues based on which warp they were scheduled from, with lower warp IDs mapped to queues with higher priority. In the configuration evaluated in Section 3.4.2, there are 16 of these queues with 3 warps mapped to each queue. The queues are needed over and above the scheduler for priority because in cases where the LSU has been stalled for several cycles then becomes available again, the GTO scheduler will schedule from its current warp rather than the oldest warp. Using these queues, *WarpPool* has more control of which warp can issue memory instructions, allowing

[1] The implementation in Section 3.3.9 has 6 pipeline stages.

it to prioritize warps to improve temporal locality. We use a fixed priority order for the queues, which was proved effective by Jia et. al [51].

3.3.3 Intra-Warp Coalescers

The intra-warp coalescers merge requests in the same warp to the same cache lines. Intra-warp coalescing is the bottleneck for the memory divergent benchmarks, because it takes multiple cycles for each request to exit the coalescer. To relieve this bottleneck, *WarpPool* has multiple intra-warp coalescers. Only one request for one cache line can exit an intra-warp coalescer per cycle, but each coalescer can issue requests in parallel. The design of the intra-warp coalescers is detailed in [78].

When an intra-warp coalescer is ready to accept a new instruction, a load or store is popped from the highest priority instruction queue. Before the instruction moves to the coalescer, its registers are read and addresses are generated. Once in the intra-warp coalescer, one cache line per cycle is issued. For loads, the intra-warp coalescer issues into the inter-warp coalescer, and for stores, it issues directly to the cache. Metadata about which threads in the warp request which parts of the cache line are passed along with the request.

3.3.4 Inter-Warp Coalescing Queues

The inter-warp coalescing queues make up the window in which requests from different warps are merged. Requests coming from the intra-warp pipelines are map-ped to one of many queues based on a subset of bits from their address. When a request is inserted into one of these queues, its cache line is matched against cache lines already in the queue, and requests to the same line are merged together.

Figure 3.7 shows the structure of the inter-warp coalescing queues. Inside each queue are two tags identifying a cache line with slots underneath that accumulate requests to that cache line. For each merged request, the queues need to track which warp they are from, which load instruction in that warp needs the data, and metadata about how to map data in the cache line to the threads in that warp. When a request is inserted into a queue, a lookup is done against the tags and the

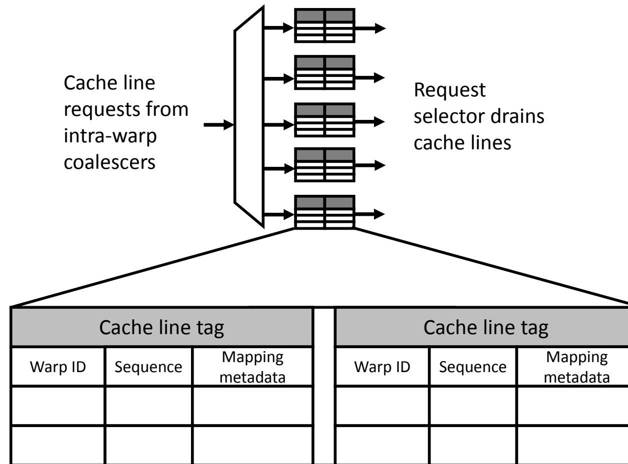


Figure 3.7: A diagram of the inter-warp coalescing queues. Requests exiting the intra-warp coalescers are merged with other requests to the same cache line in these queues.

request is inserted under a tag that matches. If no tag matches, a new tag will be allocated if free. Note that in the evaluated configuration, there are only two tags per queue, so only two tags will be searched per insertion. Previous work has found that the larger cache tag lookups (as part of GPU cache power) make up a very small fraction of total GPU power [78].

The total number of tags across the queues is the window size across which requests can be merged. Structuring these tags as two per queue with many queues minimizes the number of tag lookups that need to be done and reduces the number of times intra-warp coalescers attempt to insert into the same queue. Addresses are mapped to a queue based on bits in their address, using the method described in [96]. The same bits are also used to map addresses to cache sets, and matching the two hashes simplifies future designs where *WarpPool* issues to a cache banked by sets.

3.3.5 Request Selector

Requests remain in the inter-warp queues until they are selected to be sent to the L1 cache. There are three competing concerns that the request selection logic must balance: first, keeping requests in the coalescing queues for longer leads to more coalescing, because requests can only merge with requests in the queues. However, the second concern is latency, ensuring requests do not stay

in the queues so long that the coalescer adds latency to misses. Third, the order the requests are sent to the cache must preserve temporal locality inside warps, so that reordering requests in the inter-warp queues does not lead to cache thrashing.

For most benchmarks, the most effective strategy is to drain the *oldest* request in the queues. This is implemented with a circular queue that saves the order requests were inserted into the queues. Choosing the oldest request balances the three concerns: it keeps requests in the queues for as long as possible without adding latency, and it follows the order produced by the GTO scheduler and the queues in front of the intra-warp coalescers, both optimized to prioritize access to the cache.

For extremely cache-sensitive benchmarks, an alternate strategy that prioritizes one warp's requests, the *warp ID* policy, leads to a lower miss rate. Because the inter-warp queues store accesses to the cache at the granularity of individual requests rather than load instructions, *WarpPool* can prioritize requests at a finer granularity than the warp scheduler can, similar to the opportunity exploited by [51]. Being able to schedule requests rather than instructions allows newly issued requests from the warp with access to the cache to interrupt requests from instructions issued by other warps, leading to fewer requests by different warps between accesses by the prioritized warp and causing less cache thrashing.

WarpPool uses performance counters to determine when to switch selection policies. The benchmarks begin execution in *oldest* mode. During quanta of 100,000 cycles, each SM tracks the L1 miss rate. If the miss rate is above 99% during a quantum, *WarpPool* toggles the policy for the next quantum. This discovers whether one of the strategies causes thrashing for a benchmark and switches if it does. Quanta of 100,000 were chosen to be sufficiently long for changes in the miss rate to stabilize before a new policy decision is made.

3.3.6 Metadata Tracker

Because each thread in a SIMD load can request a different word in a cache line, the LSU needs to keep metadata about which words in the requests cache line map to which thread in each outstand-

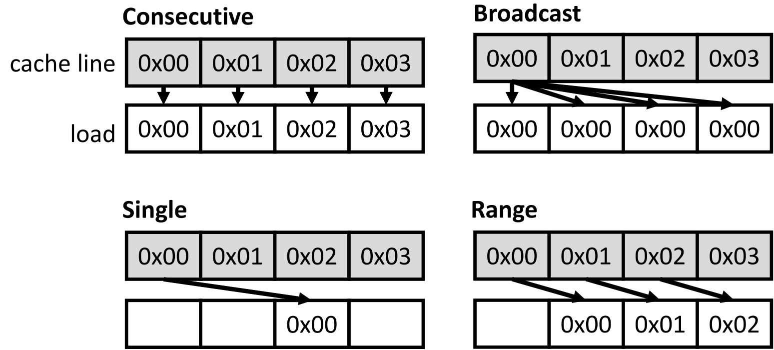


Figure 3.8: Common mapping patterns from cache line words to threads

ing load instruction. This way the LSU has enough information to write the correct word to the correct threads registers when the data returns. To move the data to the correct thread, the baseline load-store unit contains a crossbar that is able to move any word in the cache line to any thread. In *WarpPool*, this metadata must be stored for every request in the inter-warp coalescing queues as well for every request in the L1 cache's MSHRs. Because the metadata needs to be stored for so many requests, minimizing the size of the metadata is important to keep overhead reasonable. To do this, common mappings of threads to words in a cache line are recognized by the intra-warp coalescers and encoded in fewer bits.

There are four common mapping patterns that can be encoded by *WarpPool*, shown in Figure 3.8. In the *consecutive* mapping, the threads map 1-to-1 with the words in the cache line. In the *broadcast* mapping, every thread requests the same word. In the *single* mapping, one thread in a warp requests one word from a cache line. In the *range* mapping, a consecutive subset of the threads request a consecutive subset of the words in a cache line. Figure 3.9 shows what percentage of the memory requests across the benchmarks use each of these mappings. There are more *single* mappings than the other types because one load can generate up to 32 *single* mappings, each requesting one word for one thread, whereas the other types of requests are for multiple words for multiple threads. Each of these encodings requires a maximum of 10 bits, as opposed to the 320 bits otherwise needed. In the case where none of these mappings apply, an 8-entry *thread map table* entry is allocated to store the mapping. *WarpPool*'s intra-warp coalescers have an added pipeline stage to identify a mapping pattern and allocate a table entry, if necessary.

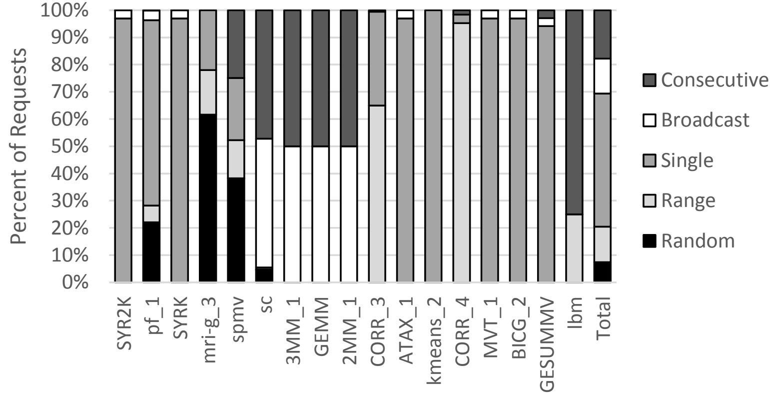


Figure 3.9: Relative occurrence of mapping patterns in benchmarks

Metadata needs to be stored for all requests sent to the memory system, including the requests in MSHRs. As requests exit the coalescer to the cache, their metadata is stored in the *MSHR metadata table* until the data comes back from the cache.

3.3.7 Writeback

When the data for a request comes back from the cache, the writeback unit uses the metadata in the table to map the data to the correct registers. If a map table entry was allocated for request, it is released at writeback. The mappings that do not require the map table can use simple selectors to move the data, whereas the mappings from the map table require the pre-existing crossbar. The registers for one warp can be written back to the registers at one time. Data returning from the cache along with its coalescing metadata is stored in a 2-entry queue as the data for each warp is sent to the register file. The cache stalls when this buffer is full.

3.3.8 Stores and Memory Consistency

WarpPool only performs inter-warp coalescing on loads. Stores progress through the instruction queues and intra-warp coalescers like loads, but instead of issuing into the coalescing queues, they issue directly to the L1 cache. Coalescing stores would require buffering the 128 bytes of data to be stored, and because GPU L1 caches are no-write-allocate, stores can be issued without concern

about destroying any intra-warp locality. Each cycle, a selector chooses whether to allow a load from the coalescing queues or a store directly from an intra-warp coalescer to drain to the L1 cache. In order to reduce the miss penalty, this selector prioritizes loads.

CUDA has a weak memory consistency model where there are no inter-warp consistency guarantees except as provided for atomics and barriers. Inside a warp, the baseline GTO warp scheduler always sends the loads and stores in program order, so any memory reordering in *WarpPool* needs to guarantee the observed behavior is the same as executing the loads and stores in one thread in program order. Previous work has maintained consistency either by flushing the reordering buffers before a store is sent to the cache [51], or by reordering only loads [123]. Neither of these is an option for *WarpPool*, as it would limit the coalescing window size to the interval between stores.

WarpPool guarantees memory consistency by using the warp scheduler to limit when stores can be issued to the LSU. A counter for each warp is incremented when a load is inserted into the instruction queues inter-warp coalescing queues and decremented when a request from that warp is sent to the L1 cache. When the counter is 0, there are no loads from the warp in the inter-warp coalescing queues. This counter needs to be 0 for a store to be issued, to ensure stores cannot be reordered with loads in the inter-warp coalescer, and to ensure stores cannot be reordered with other stores in the intra-warp coalescers. The scheduler will similarly wait for all stores to drain before issuing a load. A flag encodes when the previous global memory operation was a load, in which case it is safe to issue a load even when the counter is not 0.

3.3.9 Resource Configuration

We performed a design space sweep to determine the best number and size of each hardware resource for our workloads. Each of these resources is present in each SM. The instruction queues need to have at least 2 entries for each of 48 warps to allow for prioritization independent of the scheduler, suggesting a configuration of 48 instruction queues with 2 entries each. However, we found a configuration of 16 queues with 8 entries performed just as well but with much less selector overhead. Two intra-warp coalescers were adequate for most kernels, although some

Type	Entry Size	Entries	Total Storage	Dynamic Power	Static Power	Total Power
Instruction queues	37 bits	128	592 bytes	41.4 mW	0.4 mW	41.8 mW
Intra-warp coalescers	256 bytes	2	512 bytes	24.1 mW	0.2 mW	24.3 mW
Inter-warp coalescing queues	124 bits	64	992 bytes	43.8 mW	0.5 mW	44.3 mW
Thread map table	321 bits	8	321 bytes	9.8 mW	0.1 mW	9.9 mW
MSHR metadata table	23 bits	1024	2.9 KB	19.9 mW	2.1 mW	22.0 mW
Total per SM			5.23 KB	139.0 mW	3.3 mW	142.3 mW

Table 3.1: Per-SM storage and power overhead of *WarpPool* components

with high memory divergence like `SYRK` can achieve improved performance with more intra-warp coalescers. We used 32 coalescing queues with 2 tags each, as explained in Section 3.3.3, and allowed up to 4 inter-warp coalesces per request sent to the L1; increasing the number of coalesces increases the amount of metadata storage needed.

Table 3.1 describes the sizes of each of the hardware structures in our final configuration, per SM. The total overhead is 5.23 KB of storage, with over half of that used to build the MSHR metadata table.

3.3.10 Verilog Implementation

Since a substantial part of the hardware overhead of *WarpPool* is be the connections between components on top of any storage overhead, we implemented *WarpPool* in Verilog to perform synthesis and place-and-route to accurately determine power and area overhead. We synthesized *WarpPool* in a 45nm process at 1.2GHz to best match the GTX 480 baseline system. The MSHR metadata table can be implemented as a regular SRAM, so CACTI 5.3 [135] was used to estimate its power and area. RC values from the routed design and traces of memory accesses from the kernels were used to more accurately estimate dynamic power.

WarpPool as configured has an area of 0.36 mm^2 per SM, broken down by component in Figure 3.10. Routing accounts for 45% of the overall area, mostly in the intra-warp coalescers. This is highest in the intra-warp coalescers because they work with full load and store instructions after address generation. There is a wide bus necessary to move the addresses and store data into the intra-warp coalescers from the address generation logic and register file, and a 160-bit bus from the intra-warp coalescers to the thread map table to allow a load with a *random* mapping pattern to

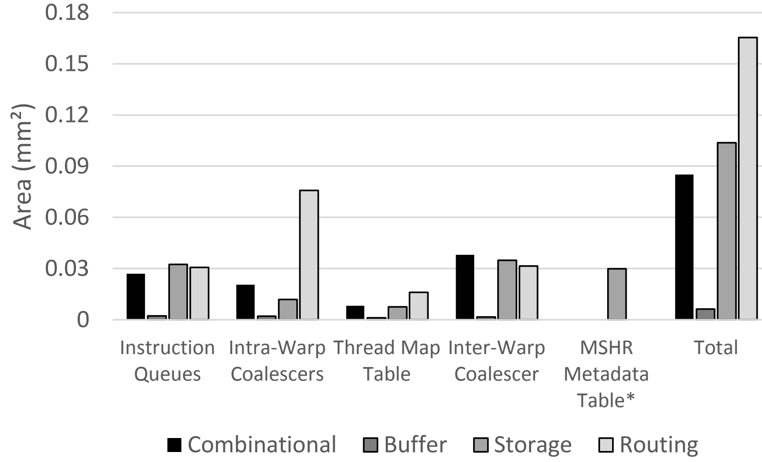


Figure 3.10: Per-SM area breakdown of *WarpPool* components, with a total area of 0.36 mm^2 per SM. (* = SRAM area calculated using CACTI)

Architecture	Fermi (GTX 480), 15SMs, 48 active warps per SM
L1 cache	32kB (64 sets, 128-byte lines, 4-way associative), 32 MSHRS
Warp scheduler	GTO
Instruction queues	16
Instruction queue entries	16
Intra-warp coalescers	2
Intra-warp coalescing queues	32 queues with 2 tags each, 4 merges per tag
Max. inter-warp coalesces/request	4

Table 3.2: Resource configuration evaluated

issue in one cycle.

Compared to the GTX 480, with a die size of 529 mm^2 [97], *WarpPool* adds 5.4 mm^2 or 1.0% to the total GPU area. The added static and dynamic power of the routed netlist, shown in Table 3.1, is 142 mW per SM, or 0.8% of the GTX 480 TDP [98].

3.4 Evaluation

3.4.1 Methodology

We use GPGPU-sim 3.2.2 [8] with the simulation parameters in Table 3.2 to model a Fermi-class GTX 480 GPU. Benchmarks were drawn from the Parboil [128], Rodinia [18], and PolyBench [37] benchmark suites. Rodina and Parboil are used as a representative cross-section of GPU workloads. The Polybench benchmarks are designed to test the effectiveness of memory access optimizations, modelling commonly used linear algebra operations; they have been used in other GPU memory optimization works such as [51]. Out of the kernels in these suites, we used a subset that is limited by GPU memory throughput, as measured by having waiting memory requests for more than 90% of execution time. Kernels were run until completion or for hundreds of millions to billions of instructions in the steady state.

We compare *WarpPool* against other techniques for increasing L1 throughput and reducing the L1 miss rate. Banking the L1 cache increases throughput by allowing multiple hits to be serviced in parallel. We implemented an L1 cache banked eight ways, with eight cache sets per bank. This cache could perform eight tag lookups and service up to eight hits per cycle, but can only service one miss per cycle because of the need to search MSHRs. Eight banks was chosen as the throughput did not increase with more banks. Each bank has a coalescing unit that selects a a line each cycle from the active load instruction that maps to that bank, which allows eight requests to be serviced in parallel by the cache.

We also compare against MRPB [51], which reorders memory requests to increase temporal locality by buffering memory requests going to the L1 cache. *WarpPool* is also able to reorder requests using the instruction queues and request selector to increase temporal locality, but adds the ability to combine requests across warps to exploit spatial locality between threads. We implemented MRPB with the configuration evaluated in [51], and calibrated the implementation against the results in that paper. The same paper also analyzes cache bypassing, which we do not implement as the bypassing technique is orthogonal to the memory reordering technique. Warp

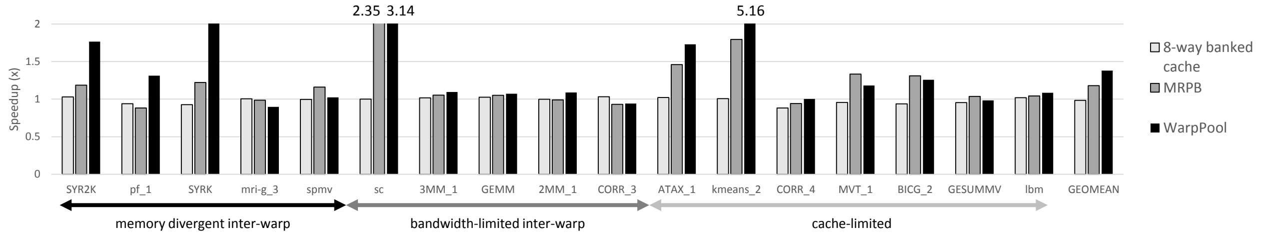


Figure 3.11: Speedup of GPU with banked cache, MRPB, and *WarpPool* over GTX 480 baseline.

scheduler-based techniques such as CCWS [118] and Mascar [123] also reduce cache thrashing, but by reducing the number of active warps. For some workloads, adding CCWS-SWL, a warp limiting technique, on top of *WarpPool* showed added benefit.

3.4.2 Results

3.4.2.1 Speedup

Figure 3.11 shows the speedup over the GTX 480 baseline of *WarpPool*, MRPB, and the 8-way banked cache. *WarpPool* yields a better improvement than the other techniques with a geometric mean 1.38× speedup. There were two mechanisms which produced this speedup.

The **memory divergent inter-warp** kernels benefitted from increased throughput to the L1 cache, created by inter-warp coalescing. The **bandwidth-limited inter-warp** kernels see speedup from more efficient utilization of the cache resources, creating more overlap between computation and L1 cache stalls. The **cache-limited** kernels see significantly fewer misses, caused by memory request prioritization.

Despite higher bandwidth to the cache, banking is not able to achieve a speedup because miss rates for GPU benchmarks are high and the banked cache could service only one miss per cycle. As well, banking creates more cache stalls because the miss is made by some banks more frequently than others, causing more cache line allocation stalls. MRPB gives a larger speedup on some of the cache-limited kernels due to its larger queue sizes, but is not as effective as *WarpPool* on the memory divergent or bandwidth-limited workloads. The following sections will examine the two mechanisms by which *WarpPool* yields speedup: increased L1 throughput and reduced L1 misses.

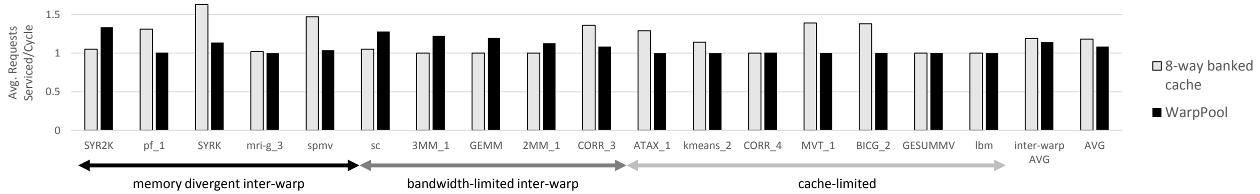


Figure 3.12: Average number of requests *WarpPool* coalesced into an L1 cache request, compared against the number of requests an 8-bank cache serviced each cycle.

3.4.2.2 L1 Throughput

Figure 3.12 shows the average number of load instructions coalesced into a request to the L1. The baseline coalescer will always yield one instruction per request, so values greater than one are due to inter-warp coalescing. The number of instructions per request can be interpreted as a multiplier on the core-side throughput of the L1 cache, with the reciprocal showing the reduction in L1 accesses. For the kernels with inter-warp spatial locality, *WarpPool* allowed the L1 cache to service an average of 1.14 requests per cycle, with 1.08 across all the kernels.

The 8-way banked cache serviced an average 1.18 requests per cycle, but its increased throughput was not effective at producing speedup. The difference from *WarpPool* was because *WarpPool* uses locality to increase throughput whereas the banked cache increases throughput by looking up requests from divergent loads in parallel. This allowed the banked cache to find opportunity in the intra-warp workloads that *WarpPool* could not, but made it ineffective at increasing bandwidth for workloads without much memory divergence like `sc`. The banked cache is not able to convert increased throughput to performance for two reasons: first, the miss rate of GPU workloads is high but only one miss could be serviced per cycle. Second, the banked cache is only able to service requests from one warp at a time, so the banks are often idle. Unlike the banked cache, *WarpPool* is able to merge misses and look across multiple warps to translate higher throughput into performance.

The number of inter-warp coalesces found by *WarpPool* is limited by two factors: window size and memory consistency. The window size limits how far apart merged requests can be. In the tested configuration, the window size was 64 distinct cache lines, because the inter-warp queues

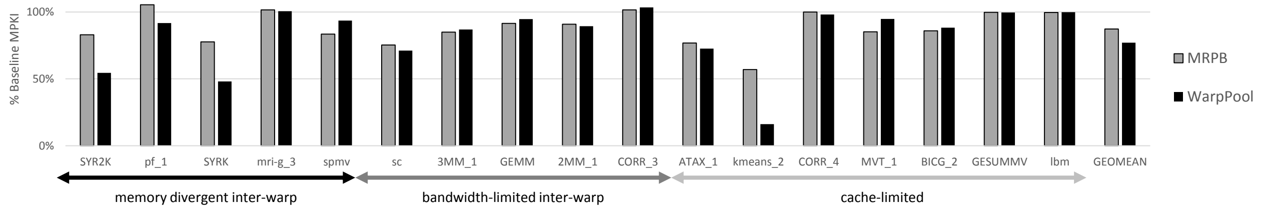


Figure 3.13: Number of misses per thousand instructions (MPKI) for MRPB and *WarpPool*, normalized to the baseline.

have 64 tags. The kernels' hit rate can also limit window size when requests drain too quickly for the queues to fill with requests. This is why `SYR2K` has a higher number of instructions per request than `SYRK`, which has a similar access pattern: the miss rate of `SYR2K` is higher, which causes more backup in the coalescing queues and leads to more coalescing. Maintaining memory consistency limited the window size for many kernels, especially for `GEMM`, `2MM_1`, and `3MM_1`, which have a store for every two loads.

3.4.2.3 L1 Misses

Figure 3.13 shows the number of misses per thousand instructions for each kernel, normalized to the misses in the baseline. The geometric mean is 77% of the baseline, showing *WarpPool* is able to not only reduce the number of accesses from the SM to the L1 cache, but reduce the number of requests the L1 made over the interconnect to the rest of the memory system.

This improvement is due to the prioritization schemes in *WarpPool*. The instruction queues allow *WarpPool* to prioritize warps more effectively than the scheduler by buffering requests rather than sending them immediately to the coalescer. The second prioritization scheme, using the *warp ID* selection policy, reduced the number of misses even further in a number of kernels, including `ATAX_1`, `MVT_1`, and `BICG_2`.

`CORR_3` saw the number of L1 misses increase, and several others do not see a reduction in L1 misses. This is caused by *WarpPool* causing early cache evictions due to two effects. First, when requests are coalesced together, the LRU is only updated once when multiple requests would have updated the LRU status multiple times. Second, inter-warp coalescing can increase the number of

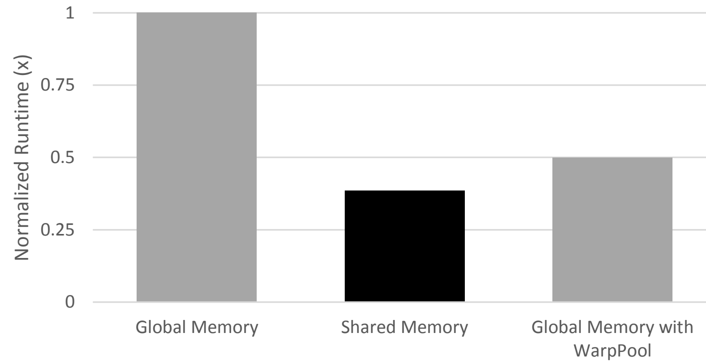


Figure 3.14: Relative execution time of matrix transpose versions, normalized to naïve global memory version.

unique requests in a given time interval, because duplicate requests are coalesced together, which causes more capacity pressure on the cache.

WarpPool was able to reduce the MPKI more than MRPB. The behavior of MRPB is similar to *WarpPool* always being in *warp ID* selection mode. This hurts `SYR2K` and `SYRK`, where temporal reuse is between warps in a block more than inside a thread. `kmeans_2` and `pf_1` likewise have reuse across warps which *WarpPool*'s *oldest* selection mode works better to find. MRPB reduced the miss rate more than *WarpPool* for `spmv`, `sc`, `MVT_1`, and `BICG_2`, because it has 6× the number of queue entries, and saw a corresponding speedup over *WarpPool* for those kernels.

3.4.3 Case Study

It can require a significant amount of programmer effort and expertise to make algorithms run efficiently on a GPU. Matrix transposes require careful implementation for GPUs because they load and store data along different dimensions of the matrix. A straightforward implementation of matrix transpose that does not use shared memory [43] has poor performance because the global memory loads are done in column-wise order, leading to high memory divergence for the loads. Copying a block of the matrix to shared memory allows both the loads and stores to be well-coalesced, although the shared memory implementation still requires padding to avoid bank conflicts.

For this benchmark, *WarpPool* runs the straightforward, unoptimized version using global

memory in 0.50× the time, which is near the runtime of the shared memory version, which runs in 0.38× of the baseline time. This shows that *WarpPool* is able to relieve the burden of programmers to optimize for particular memory access patterns.

3.5 Related Work

CPU request merging and cache throughput: Juan et al. [62] investigated methods of improving bandwidth for superscalar processors, including multi-porting and banking. Davidson et al. [24] studied memory coalescing to widen memory requests for CPUs. Our work differs by needing L1 throughput to satisfy parallel threads rather than wider single thread execution. Olukotun et al. [104] propose techniques that allow data returning from the cache to satisfy loads not yet issued to it. *WarpPool* differs by merging requests before sending them to the cache, which takes advantage of the GPU's relative latency insensitivity. Rivers et al. [115] reorder and combine requests in a CPU's load-store queue to optimize requests to a banked cache. Our work differs because GPUs do not have the same load-store queue structures. Quintana et. al [113] use a dual-banked cache to allow unaligned loads on a vector unit integrated in a CPU to complete in one cycle.

Analysis of GPU Coalescing: Hestness et al. [46] analyze the benefits of intra-warp coalescing in GPU memory systems, finding that increasing the window of threads inside a warp can lead to a large reduction in memory accesses but little speedup. We get past this limitation by merging requests across warps. Yang et al. [148] use compiler techniques to transform GPU kernels to use memory accesses with better coalescing behavior. Baskaran et al. [14] use polyhedral analysis to improve coalescing and locality in auto-parallelized code. Our work is able to optimize memory accesses dynamically in hardware.

Improving Inter-Warp Locality: Lee et al. [73] use a block scheduling policy that assigns nearby CTAs to the same SM, to capture inter-CTA spatial locality that is lost with a round-robin warp scheduler. Jog et al. [60] show there is benefit to scheduling spatially nearby warps temporally distant from each other so that warps will prefetch data for each other. Jog et al. [58] also

propose a warp scheduling technique that divides CTAs into warp groups that have different priority access to the cache. Lee et al. [69] use compiler analysis to map patterns to GPUs in ways that best preserve locality. Lee et al. [75] perform auto-parallelization for GPUs to improve inter-warp locality. Our work builds on these techniques by providing another place where inter-warp locality can improve performance.

Warp Throttling: Scheduling only a subset of ready warps can increase the amount of intra-warp temporal locality, as it prevents cache thrashing. Rogers et al. [118] detect cache thrashing and decrease the number of warps to prevent it. Later work by Rogers et al. [119] predicts how many warps' data will fit in the cache and limits the number of warps accordingly. Our work limits access to the cache after warp scheduling, which allows for finer granularity when choosing requests to send to the cache. Sethia et al. [124] use performance counters to detect cache sensitivity in order to reduce the number of threads. Our work also detects cache sensitivity with performance counters, but uses them to toggle the selection policy rather than the number of warps.

Cache Bypassing: Another technique to prevent cache thrashing is by routing requests from only a subset of warps to the L1 cache, forcing other warps' requests to bypass the cache. Chen et al. [22] watch for early evictions to prevent thrashing of cache lines with high contention, using bypassing to avoid the contention. Jia et al. [51] use a combination of request prioritization and bypassing to reduce cache contention. Zheng et al. [151] separate warps into groups, only one of which can access the cache. Cache bypassing is complementary to *WarpPool*'s prioritization methods and can be added to it to further reduce cache thrashing.

3.6 Conclusion

Many memory throughput-limited benchmarks are constrained by the interface between the SM and the L1 cache. We alleviate this bottleneck by extending the window size of the memory coalescer from the threads in one warp to requests made by multiple warps. For workloads with divergent requests, this reduces the cost of serializing multiple requests. For workloads limited

by memory bandwidth, this makes better use of cache resources. For cache-sensitive workloads, the coalescing window enables finer-grained request scheduling which reduces cache thrashing. This leads to a 38% speedup across a set of memory throughput-limited kernels. We also show our technique can help GPU programmability by achieving high performance without the need to optimize a workload's memory access patterns. We implemented *WarpPool* in Verilog to show that *WarpPool* achieves these benefits with minimal power and area overhead.

CHAPTER 4

Register File Storage and Energy Reduction

4.1 Introduction

As Graphics Processing Units (GPUs) proliferate from gaming desktops into datacenter and mobile environments, they are required to be more energy-efficient than ever before. GPUs' high computational throughput comes from their massively multithreaded architecture, where stalls in one thread are hidden by switching to another thread and many threads can issue each cycle. This requires the GPU to store the context for every active thread in a way that makes it available at any time.

Since registers make up most of each thread's state, GPUs have very large register files. To store the registers for the 32 SIMD lanes for each of the 64 hardware threads (called warps), each core (called an SM) in NVIDIA's Maxwell architecture has a 256KB register file. Because of its size, on GPU architectures similar to the GTX 980, the register file consumes up to 13% of total GPU power, nearly as much as the arithmetic units or DRAM [77]. As GPU designs provision more concurrency, the register file will only grow. Therefore, reducing the size of the register file and the energy used to access it is an important part of making GPUs more efficient.

Previous approaches have focused on optimizing register storage space, as shown in Figure 4.1. The baseline (a) reads all operands from the large register file (RF) and has a separate L1 data cache. By adding a smaller hardware [30] or software [32] managed cache in front of the main register file (b), most register accesses can be filtered by a smaller structure. By dynamically

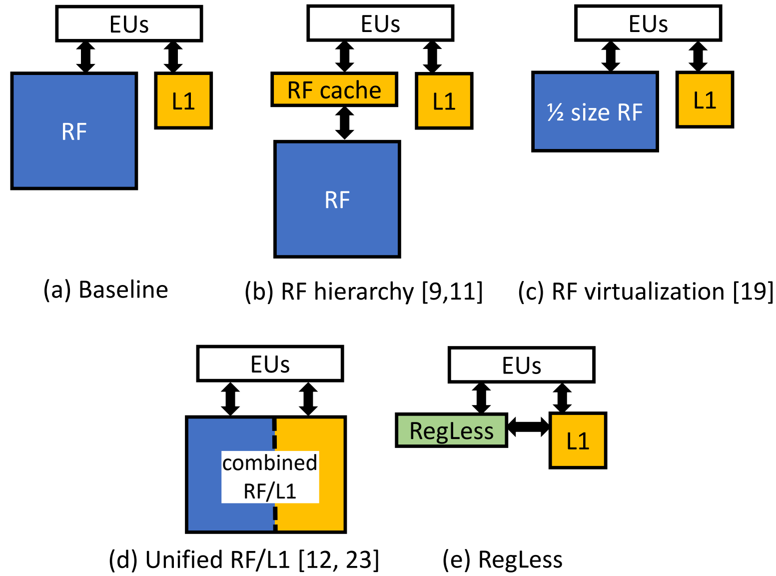


Figure 4.1: Comparison of GPU register energy reduction techniques that change how execution units (EUs) read operands from the register file (RF)

reusing register capacity otherwise used to store dead values, a smaller register file (c) can be provisioned or portions of the register file can be power gated [50]. For applications that do not use the entire register file, portions of the register file can be used as more L1 cache or scratchpad memory (d) to increase occupancy and performance [33, 56].

Our technique, RegLess (e), reduces the amount of storage space by anticipating when registers will be used in time. Instead of a full register file that contains every live value, RegLess maintains a small operand staging unit. Code running on the GPU is divided into regions and just in time for a region to begin execution RegLess allocates space for it in the staging unit. Most operands' lifetimes are contained in one region, so when that region is finished executing, the staging unit can reuse their storage. An operand value with a lifetime that spans regions can be evicted into the memory hierarchy when no active region is using it, so before a region can begin executing, RegLess fetches any needed long-lived registers from memory.

In order for the hardware to manage the operand staging unit effectively, it needs visibility into future register usage, which is provided by the compiler with annotations in the instruction stream. A hardware resource manager uses these annotations to anticipate which registers a warp is about to access. The resource manager also controls which warps are eligible to issue instructions,

ensuring the warps allowed to execute always have their registers ready in the staging unit. Other annotations inform the hardware when a register dies and can be erased from the staging unit or memory system.

Only a few registers can be transferred between the staging unit and memory without incurring performance loss. Because the L1 data cache in each SM can only service one request per cycle, the bandwidth available to fill the staging unit is much smaller than the bandwidth needed to service register reads and writes. To address this, the RegLess compiler divides regions at the points that maximize the number of registers interior to one region, as the values in these registers will never be transferred to or from memory. By creating regions that rarely need their operands fetched from memory and managing staging unit capacity for these regions in hardware, RegLess can maintain performance while vastly reducing register storage.

Our contributions in this work include:

- Replacing the GPU register file with a small operand staging unit that only holds values about to be accessed.
- Designing compiler techniques for dividing kernels into regions that maximize the number of registers interior to a region.
- Detailing hardware components for managing operand storage capacity, fetching operands from memory just before they will be used, and minimizing the performance impact of storing register values in memory.
- Analyzing the power and area required by RegLess with a placed-and-routed Verilog model.
- Demonstrating that the RegLess system can reduce register capacity by 75% with no average performance loss.

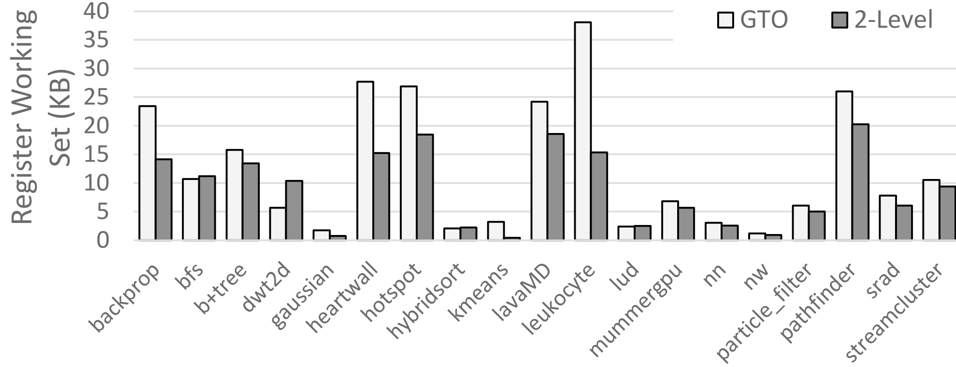


Figure 4.2: Average register working set in 100 cycle window for GTO and 2-level warp schedulers in baseline 2048 KB register file for benchmarks in Rodinia [17]

4.2 RF Replacement Challenges

Replacing the register file with an operand staging unit smaller than needed to hold all live registers presents several difficult challenges. Managing the staging unit must be done precisely, as exactly the right operands need to be present in the staging unit at exactly the right time or performance will suffer as warps stall for operands to become available. A first challenge is determining how much of the staging unit each warp will have access to. Another comes from the limited memory bandwidth available for moving values in and out of the staging unit. A final challenge is conserving memory system capacity to allow more cross-region registers to fit in the L1 cache.

4.2.1 Capacity Allocation

Because of the small capacity of the staging unit (25% of the baseline register file or less), only a subset of registers can be stored in it at any one time. The staging unit will hold fewer registers than might be live across all active warps, so not every warp can have all its live registers present in it. However, because not all registers are accessed by every warp all the time, there is an opportunity to store only the subset of registers that will be used in an interval of time. Figure 4.2 shows the register capacity accessed in a 100-cycle window in each Rodinia [17] benchmark. For most applications, this is 10% or less of the baseline register file's 2048 KB capacity.

One approach to allocating capacity would be with standard spills and fills inserted by the

compiler. Each warp would have an allocation in the staging unit that it would manage using load and store instructions. This strategy fails to take into account that warps are not equally likely to issue instructions – dynamically, some warps will be stalled for long-latency operations and their space in the staging unit would be better used by active warps. Another approach would be modelling the staging unit after a cache, allocating space based on which registers are most recently accessed. Although this works when the backing store for the cache is the main register file, this reactive strategy would cause stalls for register fetches if the cache was backed by main memory.

To allocate staging unit capacity only to active warps, RegLess coordinates the warps eligible to issue instructions with the warps that are allocated space in the staging unit. Figure 4.2 shows that the two-level warp scheduler from [30] reduces the amount of register space that is used in each interval relative to the baseline GTO by scheduling instructions from only a subset of warps at a time. Extending this insight, RegLess only allows warps that have an allocation in the staging unit to issue instructions. In this way, all allocated space is useful to a running warp.

In order to know how much capacity to allocate to each running warp, the RegLess hardware receives information from the compiler. Hardware by itself cannot know how much capacity each warp will use, but the compiler has a global perspective of exactly which registers will be accessed at which points in the program. The best allocation decision is a combination of the hardware's dynamic perspective of how much staging unit capacity is available and the compiler's global perspective of warps' future needs. The RegLess compiler divides a kernel into atomic regions, and the beginning of the region is annotated with how much capacity that region requires in the staging unit in order to run. A hardware resource scheduler activates warps when their next region is allocated capacity in the staging unit. In this way, RegLess anticipates warps' future resource needs in its allocation decision.

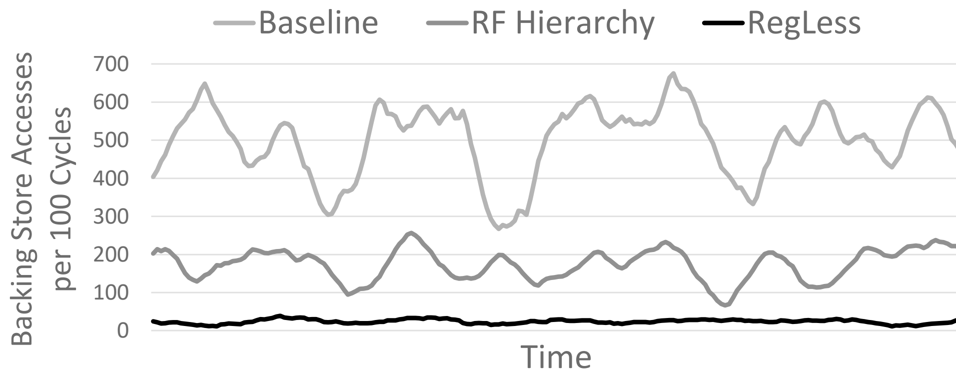


Figure 4.3: Accesses to the register backing store per 100 cycles during the steady state of hotspot for baseline, RF hierarchy [32] with 8-entry scratchpad, and RegLess with 8 entries per warp

4.2.2 Memory Side Bandwidth

The second problem with eliminating the register file is that the backing store, global memory through the L1, has limited bandwidth. In our model, only one access can be made to the global memory system through the L1 cache per cycle. This is also more constrained than previous work, which has recourse to a main register file with full bandwidth [30, 32]. In order for this not to limit performance, fewer than one request per cycle can be made to transfer a register in or out of the staging unit.

To reduce the number of accesses made to L1, the RegLess compiler creates regions with as many interior registers as possible. *Input* and *output* registers hold values used to communicate between regions, whereas the lifetime of an *interior* register lies entirely inside one region. By guaranteeing each region the space it needs in the staging unit while it executes, any interior registers whose lifetime is contained in the region will never need to be transferred in or out of it. The only registers that must be transferred in or out of the staging unit to the L1 are inputs and outputs – the values communicated between regions. Therefore, when the compiler decides where to put the boundaries between regions, it chooses points with the fewest number of live registers.

The other part of the solution is loading each regions' input registers into the staging unit sufficiently early that instructions do not stall waiting for their registers. Instead of loading registers from L1 when they are first accessed, all the input registers for a region are fetched before any

instructions from that region can be issued. We call this register fetching process *preloading*. The staging area needs enough capacity that several warps can be issuing from their regions while other warps preload registers for their next region. Output registers can stay in the staging unit until evicted, so RegLess prefers activating a region from a recently active warp in case an input to the new region was an output of a recent one.

Together, these strategies mean there are far fewer requests made to the backing store than in previous work. Figure 4.3 compares the number of accesses made to the main register file in the baseline to the accesses made to the large register file in [32] and the accesses made to the L1 cache in RegLess with the same capacity. Because so few accesses filter through RegLess to the L1, on average 0.9%, it becomes feasible to use the low-bandwidth L1 to store cold registers.

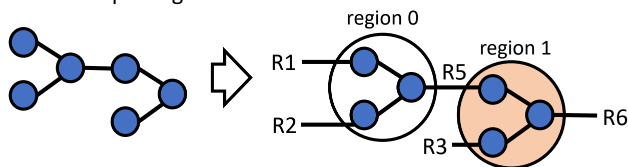
4.2.3 L1 Cache Capacity

A final problem is that the staging unit and the L1 combined are smaller than the register working set for many kernels. Because of this, registers and data in L1 would contend for space in L1 and registers may be evicted across the interconnect to L2 or DRAM. It would take hundreds of cycles to fetch these registers and they would contend for scarce L2 bandwidth. Previous work [74, 87] has recognized that many registers hold values that have similar values for each 4-byte contribution from each lane. This makes registers amenable to compression. RegLess compresses registers that are evicted from the staging unit to the memory system, matching them against fixed patterns that are intentionally simpler than full register file compression techniques, in order to fit more register values in the limited L1 capacity.

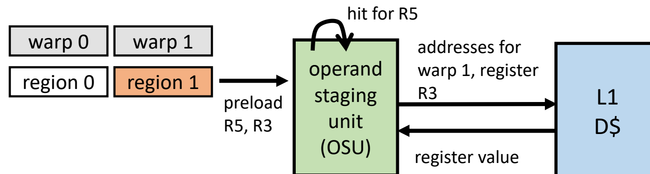
4.3 Design Overview

In these ways, RegLess' design overcomes the challenges of eliminating the register file using hardware capacity management guided by compiler annotations. To further demonstrate how RegLess operates, we will walk through each component of the system.

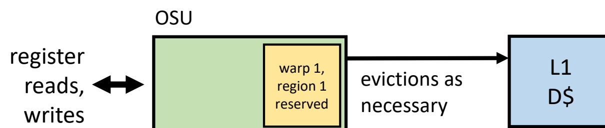
① Compiler divides code into regions and annotates region input and output registers.



② When a warp starts a new region, requests for that region's inputs are sent to the operand staging unit (OSU). Most will be hits in the OSU; a small fraction may be requested from the L1 cache.



③ As a region executes, all registers are serviced from the OSU. When output values are produced, they are saved in the OSU and may be eventually evicted to L1.



④ The capacity manager, guided by compiler annotations, manages which warps have access to space in the OSU.

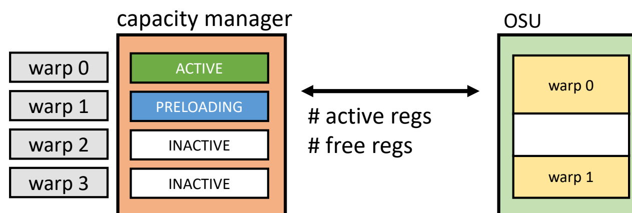


Figure 4.4: Walkthrough

First, at compile time (part ① in Figure 4.4), the kernel is divided into regions of instructions. The compiler annotates the input and output registers of each region. Since the vast majority of registers are intermediates with short lifetimes, these regions have a small number of input and output registers compared to the number of registers which are both produced and consumed inside the region.

At run time, registers are stored in a *operand staging unit* (OSU), with space allocated based on compiler annotations on the regions. When a warp starts running a new region, that region's input registers need to be assembled in the OSU ②. The OSU may already contain some of

these registers, and the others will be loaded from the L1 data cache. Not all of a warp's registers are loaded – only the ones that will be used in the next region. The instructions in a region are guaranteed to have the registers they need available in the OSU as they execute. As values are used for the last time in the region, they are erased from the cache or marked for eviction ③.

RegLess orchestrates this process by actively managing the OSU capacity. A *capacity manager* (CM) ④ makes warps eligible to issue instructions only when all the warp's input registers are present and there is space for all the warp's interior registers in the OSU. As warps complete regions, their registers are reclaimed and the CM uses the free capacity to preload registers for a new region. The register working set often fits in the OSU, and any overflow almost always fits in the L1 data cache and does not generate traffic at lower levels of the memory hierarchy.

Next, we will describe the compiler techniques and hardware implementation of RegLess.

4.4 Compiler Code Generation

In order for the hardware to make register allocation decisions, it needs to know which registers to move into the staging unit and when those registers will no longer be needed. The compiler provides this through metadata inserted in the instruction stream, as it has whole-program visibility into when each register will be used. In order to do this, the compiler divides the kernel into regions of instructions and annotates each region with data about which registers must be present to start the region, the number of temporaries used in the region, and when the regions' registers can be erased or evicted from the staging unit and memory system.

4.4.1 Region Creation

Where the compiler chooses to create region boundaries affects how much data movement is necessary when running a kernel. Registers that are produced outside the region but used inside it need to be fetched from memory before the region can start running, but registers with their lifetime entirely within one region are guaranteed to never be transferred to memory, as regions are scheduled

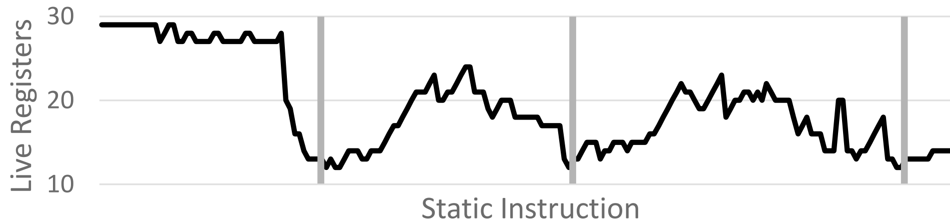


Figure 4.5: Count of live registers for a portion of `particle_filter`, with low live register points highlighted

atomically by the RegLess hardware. Therefore, the compiler should draw region boundaries to minimize the number of registers communicated between regions and maximizing registers interior to a region.

This matches register usage patterns in kernels. The number of live registers changes over time in a program – for example, while a complex expression is being computed, there will be many live registers to hold intermediate values, but these will be collapsed to a single value at the end. These points with fewer live registers form natural seams in the program for region boundaries. Figure 4.5 shows these seams in a portion of the `particle_filter` application.

It is also important that a long-latency global load and its first use are not inside the same region. If a warp were to stall on a long-latency load in the middle of a region, it would consume space in the OSU while not being able to issue any instructions. Instead, long-latency operations should happen on the edges between regions to overlap the time the register is waiting for the load with the time it is waiting for capacity in the staging unit. To achieve this, the compiler splits regions containing a load and its use.

Unlike the strands in [32], we do not allow regions to span basic block boundaries, which allows the register management to be oblivious of control flow. This does not increase data movement, since the OSU only evicts regions' output registers when more capacity is needed – if two regions from the same warp are scheduled close to each other in time, many of the input registers of the second region are often still in the OSU and are never transferred from memory. RegLess' register usage annotations are more specific than those in Zorua [136] as RegLess manages exactly which registers hold live values across region boundaries, not only how much register capacity is needed

overall.

4.4.2 Region Creation Algorithm

RegLess' region creation algorithm is shown in Algorithm 1. The `CreateRegions` procedure starts by creating a control flow graph with regions equal to basic blocks. It then iterates through each region, determines whether it meets all constraints, and if not splits it into two regions. The first new region from the split is guaranteed to be valid, but the second must be re-examined by the algorithm.

The `IsValid` function determines whether a region is valid by checking whether the region uses few enough registers to fit in the staging unit hardware. The maximum number of registers used in the region is used to limit the amount of the staging unit one region can fill, so that one region cannot take up too large a fraction of the OSU and limit concurrency (line 18). Because the staging unit is split into multiple banks, the registers used by a region must fit inside those banks (line 20). Finally, a global load and its first use may not be in the same region (line 22).

To determine where to split a region, the `FindSplitPoint` function identifies a window in which the split should happen. The last instruction in this window ($upperBound$) is the first PC where the first new region from the split would be invalid. The first instruction in this window ($lowerBound$) is the place that would put the region boundary between the most global loads and their first uses. The beginning of the window is adjusted to contain at least six instructions if possible, to avoid degenerately small regions. Then, the region is split at the point in this window where the split would create the least amount of input and output registers.

The annotations in Figure 4.6 that come from this compiler analysis are the bank usages of the input and interior registers, as well as the registers to preload.

4.4.3 Register Lifetime

Because both the staging unit and L1 cache have very limited capacity, it is vital that no space be consumed by dead registers. In order for the compiler to inform the hardware about when

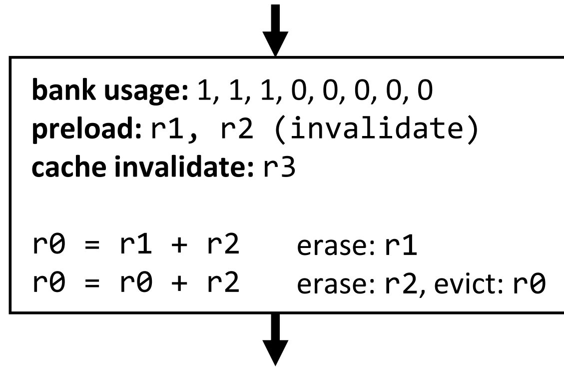


Figure 4.6: Compiler annotations added on regions and instructions

registers die, the compiler needs to take into account the two places where registers can be stored. Both interior registers and inputs and outputs can be stored in the staging unit, but only inputs and outputs can be evicted to L1. Therefore, the hardware structures in which a dead register needs to be erased depends on whether it is interior to a region or not.

Since registers with their entire lifetime within one region will only be stored in the staging unit, it is sufficient to mark the last use of the register in the region. In Figure 4.6, this is the *erase* annotation. Input and output registers also have a lifetime in the staging unit while a region is executing, in that there is some point in the region where they will be used for the last time in that region. These last uses are marked by the *evict* annotation in Figure 4.6 – note this does not mean the register must be evicted from the staging unit, only that it becomes eligible for eviction at that point.

The lifetimes of registers that live longer than one region need to be tracked so they can be erased from the L1 cache when no longer needed. These registers can either die when preloaded for the last time or when control flow eliminates the possibility of another preload. In the case that a preload is the last use of a register, the preload is set as an invalidating read, like `r2` in Figure 4.6. Registers known to be dead due to control flow at the beginning of a region are marked for cache invalidation, like `r3` in Figure 4.6.

4.4.4 Control Flow and Register Liveness

Finding the correct location to insert register invalidations is non-trivial problem on a GPU, because the threads in a warp can diverge for control flow. If not all lanes in a warp are active, a write to a register will only write to some parts of the register. Therefore, standard liveness analysis will produce incorrect results for GPU code, because it assumes that writing to a register kills the entire value. We call a definition that may not redefine an entire register's value a *soft definition*.

Tracking register liveness accurately is important for inserting cache invalidations in the correct place. A cache invalidation annotation deletes the entire register, not just the values for active lanes, so it is only safe to insert an invalidation when the entire register is known to be dead. Previous work [50] recognized this and described how invalidations must be inserted in a postdominator of both the definitions and uses in a live range. That is, the divergent control paths that use the register must reconverge before the invalidation. We expand on this contribution with more details about how to compute live ranges for GPU registers while accounting for control divergence.

To do so, liveness analysis must determine which definitions of a register are soft definitions. Algorithm 2 decides whether an instruction *insn* that defines a register *reg* is a soft definition, which is shown graphically in Figure 4.7. For a definition to be soft, there must be another definition that reaches a use with different control conditions than the candidate soft definition. Therefore, first the algorithm builds a list of the basic blocks that dominate the candidate soft definition, other than its own basic block (lines 2-3). (A basic block dominates another if control must pass through the dominator before the other basic block, and a basic block postdominates another if control must pass through the postdominator after that basic block.) Then, for each dominator, it tests whether there is a reconvergence point between the dominator and the candidate soft definition, done by testing whether there are any basic blocks that postdominate the dominator that dominate the definition (lines 6-8). This ensures that the dominating definition used is the nearest. Finally, it tests whether there is a successor with different control conditions than the candidate soft definition (lines 10-11) that uses the dominating definition (lines 12-13). If so, the candidate is a soft definition.

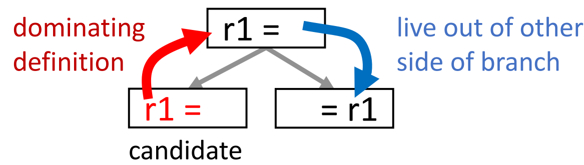


Figure 4.7: Determining whether a definition is soft. A soft definition of a register might not kill every thread's values.

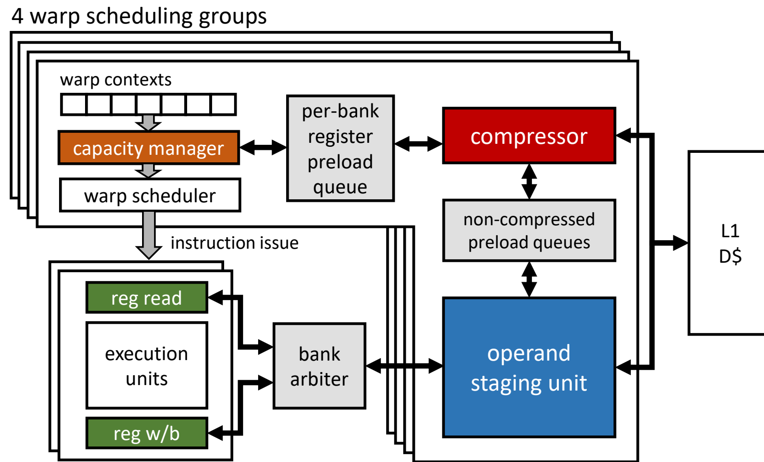


Figure 4.8: Block diagram of RegLess components in each SM

To compute when values die, standard dataflow analysis is used to compute live ranges, with the change that a live range does not end at a soft definition. Next, the death points of each live range are determined – either a last use or a control flow edge out of a loop. To cover the case where a register is defined but not used along a control flow path, the invalidation annotation is places in the postdominator of all the definitions and death points of the live range. Registers with a soft definition in a region are annotated for preloading, so that the values in lanes not taking the control flow path are preserved.

4.5 Hardware Design

At run time, the hardware follows these compiler annotations to manage staging unit capacity. **Capacity managers (CMs)** use the register usage annotations to make allocations for warps in the staging units. The **operand staging units (OSUs)** store registers for active warps and transfers

registers to and from L1 as needed to run new regions. **Compressor units** compress registers transferred to L1 to conserve capacity. Figure 4.8 shows how these RegLess components are integrated into an SM.

There is a separate shard of RegLess for each of the four warp schedulers in the GTX 980. That is, each of the schedulers has its own CM, OSU, and compressor unit. Multiple warp schedulers allow the GPU to easily issue multiple instructions per cycle, so making independent register scheduling decisions for each scheduler is important to keep this concurrency. No communication between shards is necessary because warps cannot read each others' registers. However, only one shard can access the L1 at a time, as the L1 cache can only accept one request per cycle.

The CMs sit in front of the warp schedulers, allocating space in its OSU for warps as they begin regions, and only allow the warp scheduler to issue instructions from warps that have their registers ready. The CMs read from a metadata store, not shown, which is filled by the decode stage. Active warps read their registers from an OSU. Before a warp can become active, it must assemble its active registers in the OSU, either from registers already in the OSU or by loading them from L1. Any unallocated OSU capacity is used to cache output registers for inactive warps in case they are inputs to another region.

Each execution unit has a corresponding register read unit that assembles the source operands from the OSU and reserves space for the destination registers for each instruction. After an instruction is finished executing, its value is written back to the OSU. Since instructions at the execution units may be from any warp in any scheduling group, an arbiter directs register reads and writes to the correct OSU.

In order to reduce the memory system throughput requirements of loading and storing registers from memory, compressors identify common patterns in register values, storing a compressed representation of a register if possible. The compressors contain a small amount of storage to cache compressed values.

4.5.1 Capacity Managers (CMs)

The capacity manager is responsible for allocating OSU resources to active and preloading warps. Figure 4.9 shows the components of the capacity managers. Each CM contains state machines for its supervised warps that tracks whether they are in an inactive, active, or preloading state, as well as counters tracking the number of preloads and evictions to determine when the states should transition. They also maintain a list of inactive warps in the *warp stack*. The top warp in the stack is the last one to have executed, so its input registers are the most likely to already be in the OSU.

Each cycle, the CM determines whether there is enough free capacity in its OSU to activate the top warp on the stack, by comparing the registers needed by the warp's next region against a counter of free registers. If there is space in the OSU, the CM places the registers the compiler annotated to be preloaded or evicted into queues to send to the OSU banks and updates the warp stack and counters. There is a queue entry for each line in the OSU banks, so there is guaranteed to be enough queue space to insert the preloads. The OSU notifies the CM as preloads and evictions are processed, and once all of them are completed that region's warp is activated and the warp scheduler can issue instructions from that warp. The warp scheduler does not require any changes from the baseline GTO policy.

When a region has issued its last instruction, there still may be registers that have yet to be written back to the OSU. For example, if the last instruction in the region is a global load, the value may take hundreds of cycles to be written back. While it waits, any other registers that were allocated to that region can be freed for other warps, but the pending register must stay allocated. The capacity manager tracks the number of outstanding writes for a region, and keeps its state machine in a draining state until all of its registers are written. At that point, the final registers are reclaimed and the warp is deactivated and pushed onto the warp stack.

4.5.2 Operand Staging Units (OSUs)

The operand staging units store the register values for active and preloading warps. Each OSU is made of 8 independent banks, which are independently tracked by the CM. Each cycle, the register

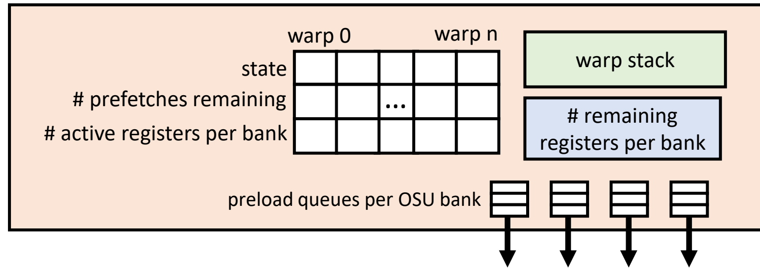


Figure 4.9: Capacity manager (CM) design. CMs track which warps have registers allocated in the OSU and are ready to execute instructions.

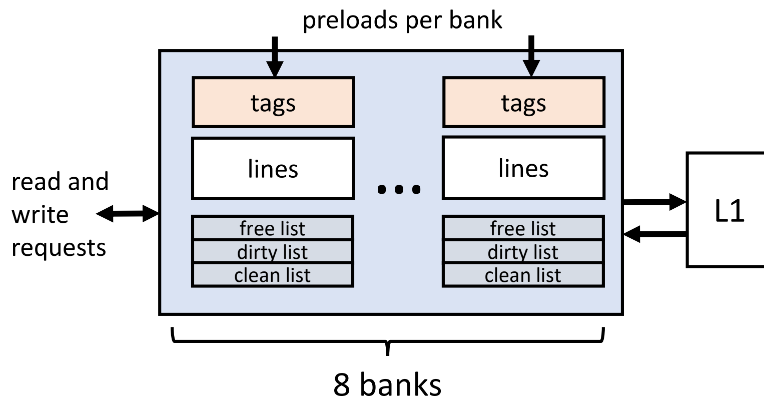


Figure 4.10: Operand staging unit (OSU) design. OSUs store register values and service register read and writes.

read and writeback units in the execution units arbitrate for access to the OSU banks of the warps and registers that they need; each bank can process either a read or a write per cycle. To read a register, the read units request a value from a bank. To write a register, the read units request an OSU entry for the future writeback, which the writeback units provide once the instruction has completed.

Figure 4.10 shows the structure of the OSUs. There are 8 banks in each OSU, with registers assigned to a bank by taking the lower 3 bits of the sum of the warp ID and register number (the compiler selects register numbers in a manner that reduces bank conflicts). Most instructions require 2 register reads and 1 write, so it is possible for each OSU to service two instructions per cycle, necessary to match the dual-issue capability of the GTX 980 schedulers. The tags in each bank store the warp ID and register ID, matching those to a 128-byte line in the data store. Each bank can complete one tag lookup per cycle, which is used when performing a register read or

preload. The OSU maintains three lists of lines that are not being used by an active region: the free list tracks empty entries, the clean list tracks registers that have not changed value since being read from L1, and the dirty list tracks lines written since their last read from L1. When a register is allocated, an entry is used from the free list if possible, then from the clean list if necessary, then from the dirty list if needed, which reduces the number of writebacks needed to the L1.

4.5.2.1 Preloads and Allocations

Registers are allocated in the OSU either through preloads or writes to interior registers. Preloads are passed from the capacity manager for each bank in parallel. If the tag access for a bank was not used by a register read in a cycle, the bank can process the preload by checking to see if the register is present in the bank. If it is present, the register is removed from the clean and dirty lists; if it is not, the bank passes the request on to the compressor. The compressor either replies with the full register value or with a signal that the register was never compressed, in which case the OSU fetches the value from L1. Cache invalidation requests are sent through this pipeline as well, but are routed immediately to the L1 cache. For interior registers, space is allocated when a warp writes to the register for the first time.

4.5.2.2 Evictions

The register lifetime annotations inserted by the compiler determine when register values are no longer needed. Registers marked for invalidation are added to the free list to be recycled. Output registers marked for eviction are placed in the clean or dirty list, depending on the value of a dirty bit that is set if the register is written. When a register write is the last use of a register in a region, the OSU passes a flag to the register read unit that reserved an entry for the write. This flag is later passed with the write's value, telling the OSU to mark the register as evictable and dirty as soon as it is written.

4.5.2.3 Register to Memory Mapping

The memory space for registers is allocated by `cudaMalloc()`, similar to other global memory buffers. Our CUDA API detects when the first kernel is launched in an application, and makes this allocation automatically. The register base pointer is passed to hardware like a kernel parameter, and the registers are laid out in memory in order of register number, such that all the values of R0 for every warp are sequential in memory, then all the values of R1, and so on. Because different warps tend to access the same register numbers close to the same time, this minimizes cache set conflicts.

The L1 cache is by default write-through and write-evict, which would prevent dirty register values from being stored in the L1. We modify the L1 to be write-back for register values with the added optimization that the old value does not need to be fetched from memory on a write, as we guarantee the write will overwrite the entire cache line by preloading any register that may be only partially written.

4.5.3 Compressor

Register compression is able to reduce the amount of memory traffic required to supply the OSU. The goal of compression is to reduce both the number of accesses sent to the L1 and the space each cold register consumes, as both L1 bandwidth and capacity are scarce. Instead of needing to fetch or evict one cache line per register, many compressed registers can be stored in one line. As registers move in and out of the OSU when preloaded or evicted, a compressor matches the register value against a set of patterns and if possible moves only a compressed representation to and from the L1 cache. The compressor also contains a small cache for compressed registers.

For preloads, the compressor is on the datapath between the CM and the OSU. The register index is first matched against a bit vector which tracks whether a register is compressed. This way, the compressor does not need to bring in a line of compressed registers from the L1 only to determine whether a register is compressed. Evictions from the OSU first pass through the compressor, where the value is matched against common patterns by a compression unit. Any

misses or incompressible evictions return to the OSU to be sent to the memory system.

Compression is effective due to the way kernels use registers. Previous work [74, 87] also took advantage of this with a general-purpose compression scheme, but RegLess uses a simpler scheme that matches a set of common use patterns. These patterns are constants, where all lanes of the register have the same value, stride one values, stride four values, and half-warp versions of the stride one and four patterns. For each compressed register, 8 bytes need to be stored for values for the half warp cases and 4 for the others. There are 5 compression schemes and the uncompressed state, so 3 bits per register are needed to store the state. This means that 15 compressed registers can be stored in a 128-byte cache line. Compressed lines are mapped to a separate main memory space adjacent to the uncompressed registers.

The compressor adds one extra cycle of latency for non-compressed preloads, to match against the bit vector. Compressed registers require two more cycles to match against the compressor's tags then uncompress and return the value. This added delay is small compared to the benefit of using less of the limited throughput to the L1, and preloading registers ahead of time allows this latency to be hidden. The compressor also adds similar delay when compressing registers evicted to L1, but this latency does not affect the rate warps become active.

4.5.4 Metadata Encoding

Metadata is inserted into the instruction stream by the compiler. With 10 bits of each 64-bit instruction used for the opcode [50], 54 bits of metadata can be passed per instruction. A region starts with a flag instruction which includes the bank usage and up to 3 preloads and cache evictions; more metadata instructions for preloads and cache evictions are emitted as necessary. For every 9 instructions in a region, a metadata instruction is emitted to mark when the last uses of registers: 1 bit to determine whether an operand is a last use, and a second for whether it is an erase or invalidate flag. Some regions, especially in control-flow intensive code, have few instructions but correspondingly few preloads and invalidations, so a single-instruction encoding is used for these that can encode up to 2 preloads or invalidations and flags for up to 4 instructions.

SMs	16, 64 warps each, 4 schedulers
Warp scheduler	GTO
L1 cache	48KB, 32 MSHRs, data accesses bypassed [100]
L1 bandwidth	one request per cycle
Memory system	2 MB L2, 4 memory partitions, 224 GB/s B/W
Compressor	one read or write per cycle, 16 lines internal storage (48 per SM)

Table 4.1: GPGPU-sim simulation parameters

4.6 Evaluation

4.6.1 Methodology

RegLess was implemented in GPGPU-sim 3.2.2 [9], with the parameters in Table 4.1 based on the GTX 980. Register assignment was done by `ptxas` and loaded into GPGPU-sim as PtxPlus, and the compiler infrastructure used a custom framework built upon GPGPU-sim's IR. Every benchmark in the Rodinia [17] benchmark suite was used, to evaluate against many different types of GPU workloads. The simulation accounts for the performance and energy impact of the metadata inserted into the instruction stream.

We implemented RegLess and the baseline register file design (including register banks, arbitration logic for register read and write back units, and operand collectors) in Verilog and synthesized it to a 28 nm technology netlist using Synopsys' Design Compiler. Clock gating was implemented in RegLess and the baseline to reduce power consumption during periods of inactivity. Interconnect overhead was estimated by using Cadence's Encounter tool to place-and-route the designs and extract the resistance and capacitance values of the circuits. Traces from the GPGPU-sim simulations were used to stimulate the netlist running at 1GHz in order to produce power metrics. Power information for added L1, L2, and DRAM accesses came from GPUWattch [77].

We compared the register file and overall GPU energy savings against two other register file energy saving schemes. The first is Jeon et. al [50] (RFV), which reduces the size of the register file by renaming short-lived registers. Our implementation assumes a half-size register file and a negligible cost for the rename table and metadata instructions. The other technique, in Gebhart et. al [32] (RFH) uses a compiler technique to place registers in one of two smaller structures instead

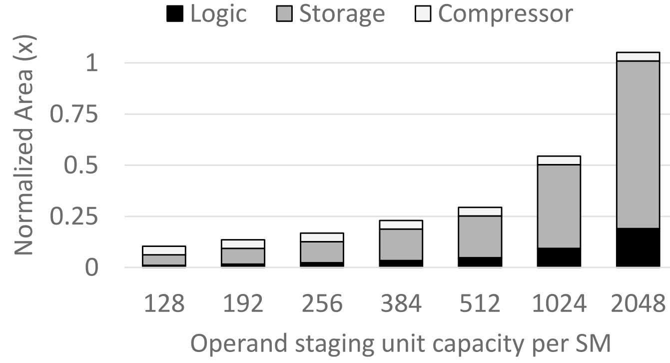


Figure 4.11: Area for RegLess configurations, normalized to 2048-entry baseline RF

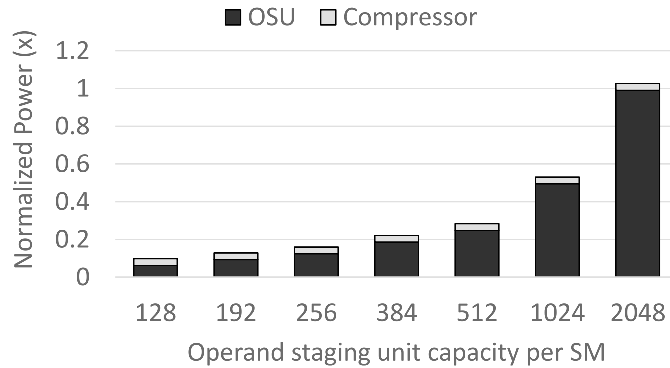


Figure 4.12: Combined static and average dynamic power for RegLess configurations, normalized to baseline RF

of the main register file when possible; we implemented the compiler technique and modelled the register file and added component energy in the same process technology as RegLess. We do not compare against works that repurpose unused register file space for other memory spaces, such as [33, 56], because their benefit comes through increasing occupancy or L1 capacity.

4.6.2 Area and Power

We evaluated multiple capacities of RegLess to find the most energy-efficient design. The area and average power of each capacity is shown in Figures 4.11 and 4.12. Both logic and storage area scales with the capacity, as more logic is needed for tags and decoding. The average power also scaled with the capacity, since more energy was required to access the larger hardware structures. Because of the added tag and compressor logic, the RegLess designs require slightly more energy

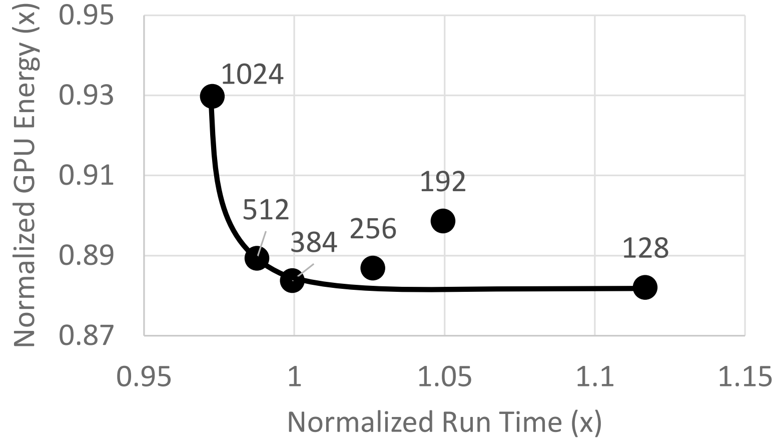


Figure 4.13: Run time vs. GPU energy for RegLess configurations, normalized to baseline. The line marks the Pareto frontier.

and power than the baseline register file scaled to their capacity.

Although smaller capacities use less area and power, they can also affect performance if too many registers must be transferred to L1. Figure 4.13 shows the geometric mean total GPU energy and running time for different RegLess capacities across all Rodinia benchmarks. Small capacities like 128 registers are Pareto-optimal in terms of energy, but our goal in RegLess was no average performance loss, so we use the 512-register version in the remainder of our results as one optimal tradeoff point between performance and energy; this capacity has better worst-case performance than the 384-register version. Larger RegLess capacities see a slight speedup, which we discuss in Section 4.6.4.

4.6.3 Energy Savings

RegLess significantly reduces the energy consumed both by the register structures and by the entire GPU, as shown in Figures 4.14 and 4.15. Focusing first on register structure energy, RegLess provided a 75.3% reduction, as compared to 45.2% for RFV and 62.0% for RFH; this added benefit came from reducing the amount of register storage below what was possible with the previous techniques. Because the register structures make up a significant amount of overall GPU energy, this led to a 11% overall GPU energy savings for RegLess, compared to 3.7% for RFV and 2.9%

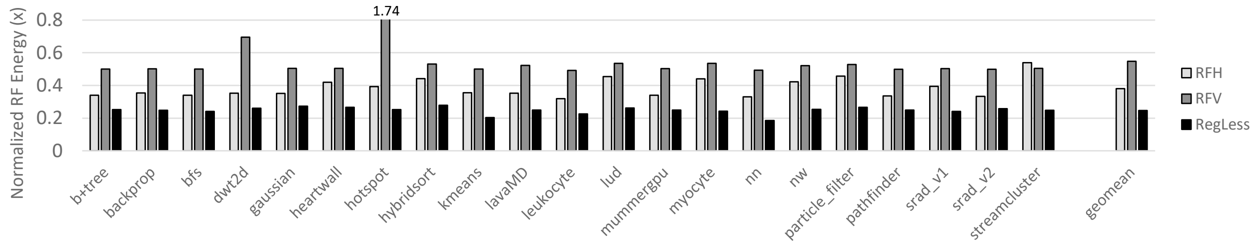


Figure 4.14: Register file energy for RFV [50], RFV [32], and RegLess, normalized to baseline

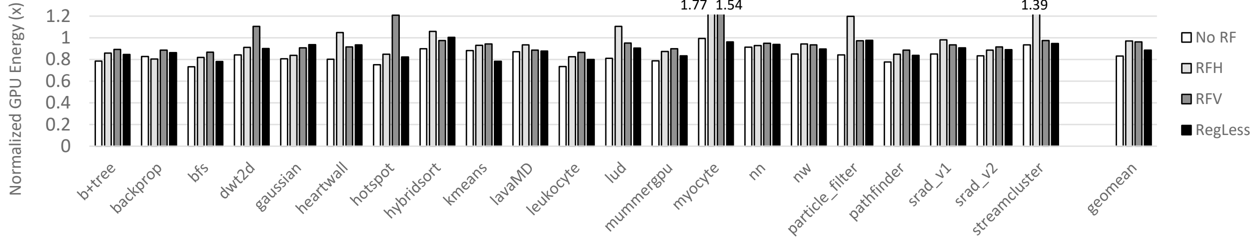


Figure 4.15: Normalized total GPU energy, including added instruction and memory accesses. The “No RF” entry is the upper bound for energy savings, which uses the baseline performance and a register file that consumes no energy.

for RFH. When computing the overall GPU energy for RegLess, the cost of added L1, L2, and DRAM traffic was included. Figure 4.15 also shows how RegLess approaches the upper bound for GPU energy savings from reducing register file energy, 16.7%, which comes from maintaining the performance of the baseline while incurring no register file energy cost.

Compared to RFV, RegLess can maintain a register structure of half the size of even the reduced register file because of the synergy between the compiler and hardware manager. As well, some register-intensive benchmarks like `dwt2d` and `hotspot` saw performance degradation with RFV due to register pressure, as noted in their paper [50]. Compared to RFH, RegLess is able to eliminate the register file backing the compiler-managed buffer. Although RFH can save energy by accessing the large main register file significantly fewer times than the baseline, each access to that register file is more expensive than to RegLess’ staging units. A two-level warp scheduler is integral to the RFH technique, which can cause performance loss relative to the baseline GTO scheduler, causing RFH to consume more energy.

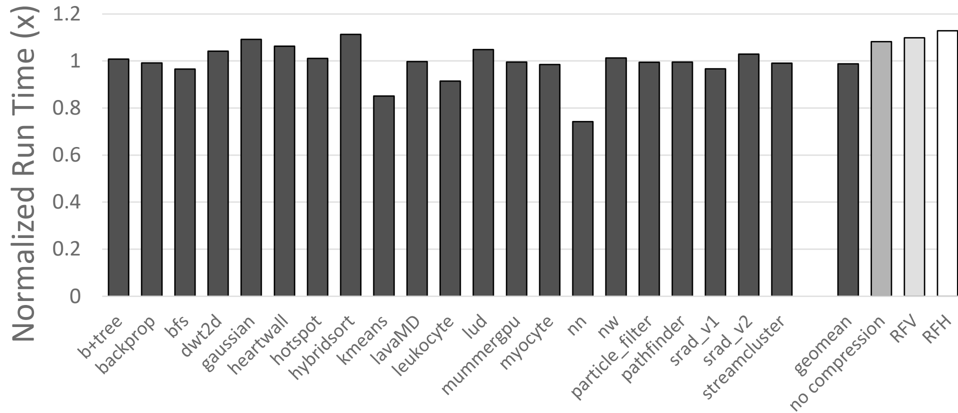


Figure 4.16: Run time (lower is better) for 512-register RegLess design normalized to baseline with full RF. The geomean is compared with RegLess with no compressor, RFV, and RFH.

4.6.4 Performance

Despite the much smaller register structure, RegLess is able to maintain application performance. Figure 4.16 shows the performance impact of RegLess on the Rodinia benchmarks relative to the baseline with a full register file, demonstrating that RegLess can match the baseline run time. Most benchmarks, such as `b+tree`, `myocyte`, and `streamcluster` saw no performance change; many of these have a small register working set that RegLess is able to easily manage. Three benchmarks (`gaussian`, `heartwall`, and `hybridsort`) saw over 5% slowdown with RegLess. `hybridsort` and `heartwall` have kernels with complex control flow structures; since registers can often not be invalidated until their last use along all paths, there are a large amount of potentially live registers that RegLess must manage. `gaussian` has many registers live across global loads, which means that there are fewer opportunities for scheduling consecutive regions from the same warp. Other benchmarks, like `kmeans`, `leukocyte`, and `nn` saw speedup, because RegLess activates fewer warps at a time than the baseline, increasing temporal locality between memory accesses. Other register file work has seen the same effect.

Figure 4.16 also compares the RegLess geometric mean performance with other configurations. The first is the same size RegLess but without the compressor, which degrades performance by 10.2%. We also compare against the geometric mean performance of RFV and RFH, which are slower than RegLess due to their use of a 2-level warp scheduler. RegLess is independent of the

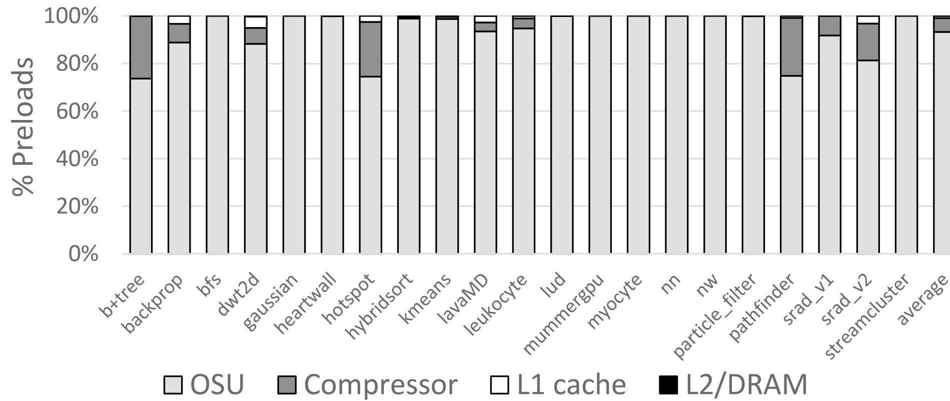


Figure 4.17: Location from which registers were preloaded. 0.9% of registers were preloaded from L1 and 0.013% were preloaded from L2 or DRAM.

choice of warp scheduler, allowing it to use the baseline GTO which is known to perform better than 2-level schedulers due to better memory locality [117].

4.6.5 Register Preload Location, L1 Bandwidth

Although the memory system is the backing store for the OSUs, Figure 4.17 shows that preloads very rarely need to access it. Some benchmarks, like `bfs` and `nw` never miss in the OSUs, because their register working set is small. Others, like `b+tree`, `hotspot`, and `pathfinder` use the extra capacity the compressors provide. Only an average of 0.9% of requests miss to the L1 cache and 0.013% miss to lower levels of the memory system. The only benchmarks that had a non-negligible number of added L2 accesses were `kmeans` (0.5% added requests), `hybridsort` (1.0%), and `dwt2d` (2.6%). For `dwt2d` and others, this is due to a large number of simultaneously live registers, few of which are compressible.

Figure 4.18 shows the average amount of L1 bandwidth consumed by transfers to the compressor and OSU per SM during the execution of each benchmark, out of the total L1 cache bandwidth of 1 request per cycle. On average, fewer than 0.02 requests per cycle were used for RegLess transfers. The benchmarks that do not miss in the OSU in Figure 4.17 do not consume any L1 bandwidth. Both `hybridsort` and `srad_v2` issue more stores to L1 than loads; this occurs when there are redefinitions of a register on a control path before the register is read. For `hybridsort`,

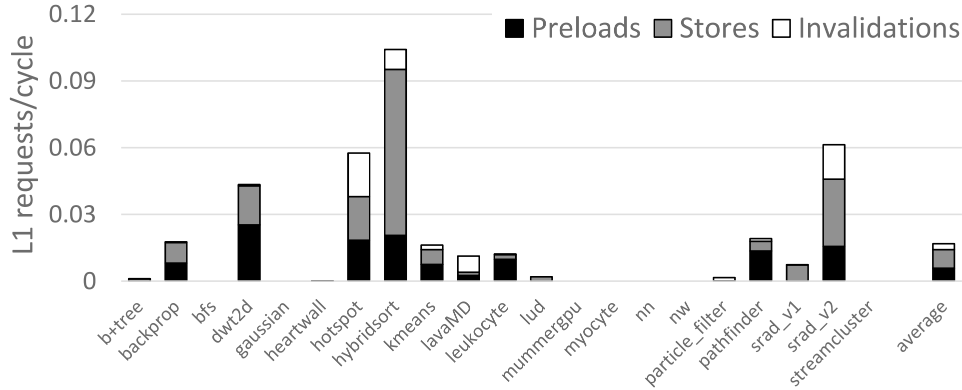


Figure 4.18: Average RegLess L1 requests per cycle

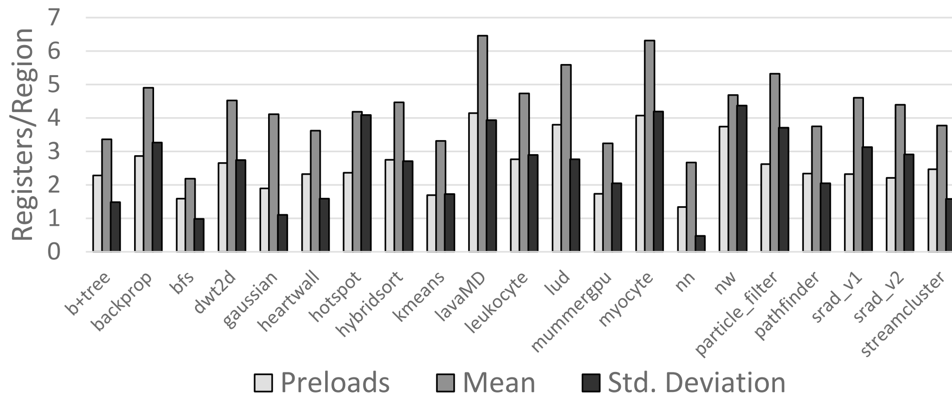


Figure 4.19: Average number of preloads, average number of concurrent live registers, and standard deviation of number of concurrent live registers per region

conservative liveness analysis again meant that more register values had to be stored than were later read.

4.6.6 Region Sizes

Figure 4.19 shows the average number of input registers, average concurrent live registers, and standard deviation of concurrent live registers for each benchmark. The number of registers reserved for a region in an OSU is equal to the number of concurrent live registers in that region. Non-overlapping short-lived registers can share the same allocation, and once an input is read for the last time, its allocation can be reused by a short-lived register. Therefore, across the benchmarks, not only is the number of concurrent live registers is consistently larger than the number of input registers, but each entry made in the OSU can be reused for several registers, showing that

	Insns.	Cycles		Insns.	Cycles		Insns.	Cycles
b+tree	3.7	150	hybridsort	6.5	379	nn	6.3	940
backprop	6.7	323	kmeans	3.9	993	nw	10.8	78
bfs	3.3	60	lavaMD	7.5	1601	particle_filter	10.0	20
dwt2d	9.5	457	leukocyte	7.7	297	pathfinder	4.9	72
gaussian	8.1	1207	lud	16.0	816	srad_v1	9.1	350
heartwall	4.6	32	mummergpu	6.4	240	srad_v2	6.9	323
hotspot	6.4	75	myocyte	9.3	120	streamcluster	4.3	16

Table 4.2: Average number of static instructions per region and average dynamic cycles per region

most register lifetimes are inside a region.

The standard deviations show that the region size varies substantially within each benchmark. Since the registers in each region cannot be negative, the standard deviations show a larger variation than if that were possible. The heterogeneity in region sizes allows warps to have different-sized register allocations at different points in execution. The region creation algorithm tends to creates smaller regions in memory-intensive or control-intensive phases and larger regions when the workload is compute-intensive, leading naturally to a mixture of different region sizes. Several of the benchmarks like `dwt2d`, `hotspot`, and `myocyte` had regions with 20 or more concurrent live registers.

Table 4.2 shows the average number of instructions per region and the average number of cycles each region was active for each benchmark. Larger regions allow there to be more interior registers, and longer-running regions reduce the rate at which L1 transfers are made. The main factors that limit region size are control flow and the restriction that global loads and uses cannot be in the same region. Therefore, compute-intensive benchmarks like `dwt2d`, `lud`, and `nw` have the largest region sizes, whereas memory-intensive benchmarks like `bfs` have smaller region sizes. There is large variation in how long regions execute, influenced by the number of instructions in a region and how many registers active regions use. When each warp has a large OSU allocation, fewer warps will be active, so active warps will make more progress than if more warps with smaller allocations were active. Therefore, memory-intensive `bfs`, with small regions with few registers each, has a smaller execution time per region, whereas `lavaMD` with larger regions with many registers switches regions less frequently.

4.7 Related Work

CPU virtual register files and instruction clustering: Oehmke et al. [103] created a *virtual context architecture* for CPUs that serviced registers from cache of a register space in memory. Because the amount of data in a GPU's register working set is much larger because many threads are active at the same time, our technique requires more active management of the register cache. Roth [121] describes techniques for releasing virtual registers when they are no longer needed. Architectures such as TRIPS [122, 34] and others using block-structured ISAs, described by Melvin et al. [90], have executed blocks of code similar to our regions. Work such as by Ponomarev et al. [112] have diverted short-lived values from handling like other registers. Yan et al. [147] allow short-lived values to be communicated through a CPU's forwarding network. We use regions as an overlay of a traditional ISA.

GPU register caching and RF size reduction: Vijaykumar et al. [136] oversubscribe resources, including registers, by annotating kernel phases. Our work focuses on reducing the size of hardware structures, and uses a more precise set of registers that need to be present. RegLess would be able to oversubscribe the register file without any design changes. Gebhart et al. [30] proposed a register cache in front of the main register file and a 2-level scheduling scheme to control access to the cache, to save the dynamic power of accessing the main register file. Other work by Gebhart et al. [32, 31] sorted registers at compile time into a 3-level register storage hierarchy, also to save dynamic power. The novel contribution of our work is eliminating the main register file as a level in the register hierarchy. Gebhart et al. [33] also propose sharing the same SRAM structures between registers, shared memory, and L1 cache. Jeon et al. [50] allow new values to replace other warps' dead values in the register file, allowing the size of the register file to be reduced. By removing the main register file and caching the active set, our technique reduces the register size to the minimum needed to maintain performance.

Compiler-assisted GPU scheduling: Park et al. [107] use compiler annotations so the warp scheduler can prioritize warps with that will soon issue a load. Wu et al. [143] expose hardware scheduling decisions on GPUs to programmers. Xie et al. [144] use a compiler to make opti-

mal register allocation and thread throttling decisions. We add a layer of scheduling that makes dynamic decisions based on static analysis. Hsieh et al. [47] use compiler analysis to determine offload candidates for near-data processing. Li et al. [82] use compiler analysis to place data in different on-chip memory resources.

Resource-aware GPU scheduling: Jog et al. [61] classify warps into short and long latency to determine memory scheduling policy. Jog et al. coordinate warp scheduling with DRAM bank-level parallelism [59] and prefetching [60]. Li et al. [83] allocate cache space to a set of prioritized warps. Narasiman et al. [93] describe two-level scheduling to allow for larger warp sizes. Pichai et al. [111] show the need to coordinate warp scheduling and MMU designs. Pai et al. [105] use elastic kernels in order to better utilize registers. Gregg et al. [38] merge kernels to increase register utilization. Lee et al. [76] coordinate warp priority and access to cache resources, Liu et al. [86] prioritize warps to reduce time waiting for barriers. Kayiran et al. [64] adjust TLP for highest performance. Rogers et al. [116] use variable warp sizing and warp ganging to decrease the impact of memory divergence. Ausavarungnirun et al. [7] change cache and memory controller policies based on warp divergence.

Divergence-aware compiler techniques: ElTantawy et al. [28] track register dependencies for control divergent threads separately in hardware, and use compiler analysis [27] to analyze control divergence to eliminate deadlocks. Rhu et al. [114] analyze divergence patterns to allow for better SIMD lane permutation. Anantpur et al. [5] transforms control divergence using linearization. Jablin et al. [49] use traces for instruction scheduling on GPUs.

Value compression and scalarization: Lee et al. [74] compress register values using base-delta-immediate encoding introduced by Pekhimenko et al. [110], which reduces the number of register file banks needed to load and store registers. Gilani et al. [35] propose a GPU architecture with scalar units and 16-bit register reads. Abdel-Majeed et al. [2] use the redundant computations done between lanes for error detection. Kim et al. [67] exploit value structure using an affine functional unit. Stephenson et al. [126] show that a large fraction of register writes are constant across warps and threads. Pekhimenko et al. [109] compress data over the GPU interconnect while

minimizing the number of toggles. Vijaykumar et al. [138] propose using excess GPU computation resources for memory compression. Keckler et al. [66] propose temporal SIMT, where scalar computations do not need to be computed by all threads.

Register file implementation: Abdel-Majeed et al. [1] reduce register file dynamic and leakage power by adding a drowsy state to the storage circuits and only reading register values for active lanes in a warp. Jing et al. [53] propose register file bank scheduling techniques that reduce bank conflicts. Namaki-Shoushtari et al. [92] power gate unused register file banks. Other work by Jing et al. [55] implemented the register file using eDRAM instead of SRAM and proposed refreshing the DRAM during bank bubbles [54]. Mao et al. [89] and Wang et al. [140] implement a register file using racetrack memory, and Goswami et al. [36] implement it using resistive memory. Tan et al. [132] implement the GPU register file using STT-RAM for energy savings, and Yu et al. [149] implement it with an SRAM-DRAM hybrid memory. Tan et al. [131] develop a method for classifying registers as fast or slow due to process variation, and Liang et al. [85] introduce a variable-latency register file to mitigate process variation. Li et al. [84] implement register files using a hybrid CMOS-TFET process. Our design because of its small size can be implemented using conventional techniques.

Register file voltage: Kayiran et al. [65] tune down performance of GPU register file and operand collector components to save energy. Tan et al. [133] reduce GPU register file energy with aggressive voltage reduction. Leng et al. [80, 79] throttle the register file when it causes voltage droop to reduce the GPU voltage guardband.

4.8 Conclusion

The register file is one of the structures on a GPU that consumes the most power. Our technique, RegLess, can replace the register file with a smaller staging unit by actively managing the contents at run time with the help of compiler annotations. The compiler divides the kernel into regions and annotates input register and the points where register values are used for the last time. At

run time, the hardware allocates capacity in the staging unit just in time for a region to begin execution. Short-lived registers spend their entire lifetime inside one region's allocation. Longer-lived registers can be evicted to memory, so the capacity manager must anticipate they will be used in order to load them before a region becomes eligible to execute. When transferred to the L1, a compressor can reduce the amount of storage needed for a register. Using RegLess instead of a full register file reduced register access energy by 75% and total GPU energy by 11%.

Algorithm 1 Region Creation

```
 1: function CREATEREGIONS(cfg)
 2:     regions ← ∅
 3:     worklist ← basic blocks in cfg
 4:     while worklist is not empty do
 5:         region ← worklist.pop()
 6:         if not IsValid(region) then
 7:             splitPc ← FindSplitPoint(region)
 8:             Split region at splitPc into firstRegion and secondRegion
 9:             region ← firstRegion
10:             worklist.append(secondRegion)
11:         end if
12:         regions.append(region)
13:     end while
14:     return regions
15: end function
16:
17: function ISVALID(region)
18:     if region.maxLiveRegs > maximum registers per region then
19:         return false
20:     else if region.maxRegsPerBank > registers in each OSU bank then
21:         return false
22:     else if region contains a global load and its first use then
23:         return false
24:     end if
25:     return true
26: end function
27:
28: function FINDSPLITPOINT(region)
29:     upperBound ← first PC where the first region becomes invalid
30:     lowerBound ← PC <= upperBound where the number of global loads and uses in both new
        regions is minimized
31:     lowerBound ← min(max(region.startPC + 48, lowerBound), upperBound)
32:     return PC such that lowerBound <= PC <= upperBound and splitting at PC results in the
        fewest number of input and output registers in both new regions combined
33: end function
```

Algorithm 2 Identifying Soft Definitions

```
 1: procedure IsSoftDef(insn, reg)
 2:     insnBB ← the BB containing insn
 3:     strictDoms ← dominators(insnBB)−insnBB
 4:     for all domBB in strictDoms do
 5:         strictPDoms ← postdominators(domBB) −domBB
 6:         if dominators(insnBB) ∩ strictPDoms ≠ ∅ then
 7:             continue
 8:         end if
 9:         for all successorBB of domBB do
10:             if successorBB dominates insnBB then
11:                 continue
12:             else if reg is live on the edge from domBB to successorBB then
13:                 return true
14:             end if
15:         end for
16:     end for
17:     return false
18: end procedure
```

CHAPTER 5

Multi-Kernel Resource Management

5.1 Introduction

Public cloud services such as Amazon's AWS, Google's GCP, and Microsoft's Azure allow users to lease time on virtual servers, which frees them from operating their own data centers and operations teams. Although these cloud services began with traditional virtual machines with allocations of CPUs, memory, storage, and networking, providers now offer virtual machine instance types that include accelerators and GPUs to meet customer performance demands for applications like genomic analytics and neural network training.

Maintaining high utilization is a challenge for large data centers like public clouds. Even well-managed data centers often operate between 10 to 50% utilization, because of overprovisioning to cover failures or spikes in demand [13]. Public clouds are especially vulnerable to low utilization because they must be ready for surges in customer demand [12, p. 98], increasing costs. As one strategy for increasing utilization, cloud providers sell more virtual machines than there are physical CPUs backing them, as many servers spend most of their time waiting for network requests to arrive or performing I/O. Cloud providers also have created several tiers of service, separating out batch jobs into a *spot* instance tier which is packed into excess capacity and sold at a discount but can be preempted at any time when that capacity is needed.

Although these strategies work well for CPU instances, increasing utilization on GPUs will require a different set of techniques. Customers paying a premium for the use of an accelerator

are doing so to constantly take full advantage of its performance, so there are few idle periods to fit another user's tasks onto the GPU. As another challenge, GPU workloads do not have periods where they wait for long-latency external events, like network requests or I/O, so interleaving waiting applications across time like an operating system's task scheduler is not effective.

Despite these challenges, previous work has developed ways to effectively share GPUs between multiple applications. NVIDIA's GRID GPUs include a virtualization layer that can divide a server GPU between multiple virtual desktops [45], and other NVIDIA GPUs also include the capability to launch kernels from multiple work queues across processes [102]. *Spatial partitioning* techniques allow workloads to run on different cores in the same GPU [3], and *simultaneous multi-kernel* (SMK) can further divide the resources inside cores [141, 146]. Sharing at these lower levels unlocks more throughput than dividing the GPU across time, as workloads with complementary resource demands can together better utilize GPU resources than either could alone. However, when workloads do not have perfectly complementary resource needs, they interfere, leading to one or both applications running slower than they would alone even in cases where the overall GPU throughput increases. Previous work controlling interference has optimized for metrics like system throughput and turnaround time that do not capture the concerns of cloud customers seeking high performance and low cost [106].

Instead, in this work, we leverage multi-kernel GPU execution to provide two tiers of service that correspond to the needs of cloud customers, while still increasing overall throughput and utilization to address the needs of the cloud provider. Traditional cloud instances are commonly sold in at least two tiers: one with guaranteed provisioning and a high level of access to the CPU suitable for latency-sensitive tasks, and another more suited to batch jobs that is cheaper but can be preempted at any time. This work, Scavenger, translates these tiers to a GPU context. It provides one tier of service with a high performance target, such as 90% of the performance of running alone, and a second tier for batch tasks that takes advantage of any resources unused by the performance tier. Because sharing the GPU between these tasks will often lead to extra throughput, both tiers can be sold at a lower price than if they were run on dedicated GPUs and

still leave profit for the cloud operator.

Previous work [142] has divided resources within a core using SMK while still reaching quality-of-service targets. This work assumed tasks were run by the data center operator who could supply task deadlines and target performance ahead of time. In contrast, Scavenger focuses on the different set of concerns that arise in the public cloud setting, where performance analysis and resource allocation must occur online, as workloads and their target performance are not known before they are launched by customers.

In order to create the two tiers of service when sharing GPU resources between two applications, Scavenger has two tasks to accomplish. First, it must determine the performance target for the high-performance application. The cloud operator will be able to specify this target as a percentage of the performance that an application achieves when running alone, but the system must translate this into a measurable metric like instructions-per-cycle (IPC). Second, the GPU compute and memory resources inside each core must be allocated between the two applications. This allocation must ensure the high-performance application meets its performance target while also providing as many resources as possible to the lower-tier batch application to maximize overall throughput.

Scavenger accomplishes both of these tasks online. To determine the performance target, the high-performance workload is periodically run alone for short periods of time. Performance counters collected during these profile intervals are used to create a performance target during the longer periods when the GPU is split between workloads, and to detect when new execution phases begin. To allocate resources between the workload tiers, Scavenger uses two types of controllers. For resources that have low overhead to adjust, PID controllers are used to control the allocation to match performance to the target. For higher-overhead changes that require context switches, a more conservative controller adjusts allocation to match actual resource usage over the long term. Together, these techniques allow Scavenger to meet the performance target for the high-performance workload while substantially increasing the throughput of the batch task relative to temporal partitioning.

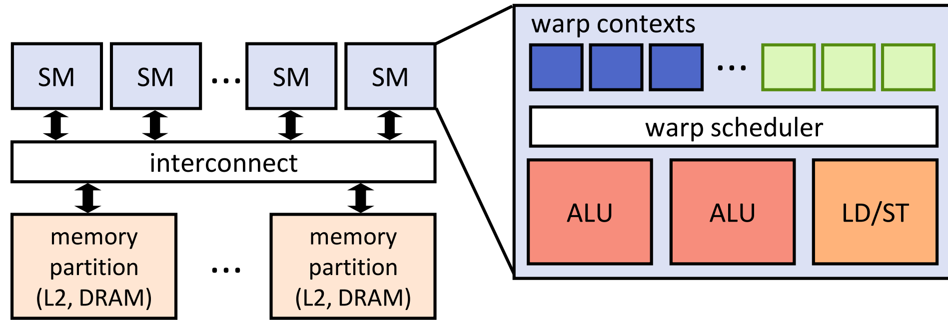


Figure 5.1: Diagram of GPU and SM design. In SMK [141, 146], the warp contexts are split between applications.

The contributions of this work include:

- Demonstrating that the interference created when sharing a GPU between two workloads using SMK requires a control mechanism suitable for a public cloud setting;
- Creating a system for performance prediction that achieves less than 5% average error when profiling for less than 10% of run time;
- Building techniques for allocating active warps, memory requests, and thread blocks between workloads that meet a performance target while maximizing total throughput;
- Showing Scavenger increases the batch throughput 1.35x relative to temporal partitioning while meeting a 90% performance target for the primary workload. This can be leveraged by a cloud provider to increase revenue by up to $0.18 per dollar.

5.2 Background and Motivation

5.2.1 GPU Architecture and Multitasking

GPU design relies on overlapping the execution of many independent hardware threads. In the architecture model used in this work shown in Figure 5.1, similar to the NVIDIA GTX 980, each GPU core, called an *SM*, contains 64 hardware threads, called *warps*. Warps are assigned to an SM in *blocks* which can synchronize with each other and share scratchpad memory. The system

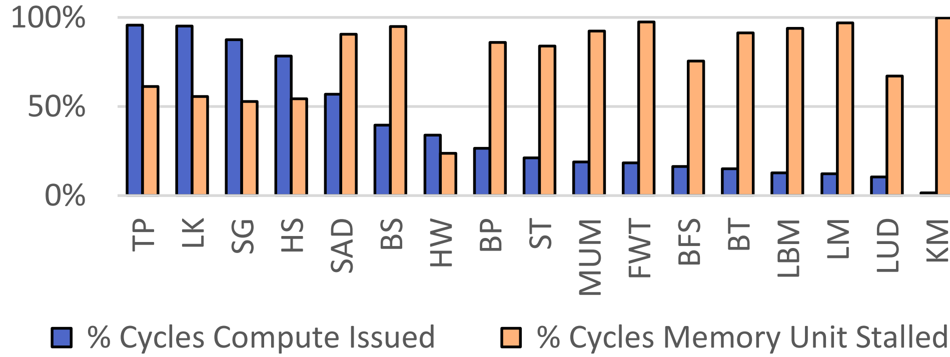


Figure 5.2: Resource demands for GPU workloads (methodology in Section 5.6.1); workloads on the left saturate compute resources, and workloads on the right saturate memory resources. Sharing an SM between complementary workloads increases overall throughput.

is divided in two halves, with the SMs on one side of an interconnect and memory subpartitions, made up of an L2 shard and DRAM controller, on the other. Inside each SM is a warp scheduler that issues instructions to arithmetic and memory functional units.

Although the hardware is able to dynamically schedule instructions in a way that keeps the functional units utilized, each thread in a kernel is identical and therefore has the same resource demands. Because of this, the resource with the highest demand in the kernel becomes saturated while other resources are underutilized. Figure 5.2 shows the resource utilization for a selection of workloads on a GPU platform similar to the NVIDIA GTX 980 (for benchmarks and simulator methodology, see Section 5.6.1). It shows the percentage of cycles compute instructions are issued during execution as well as the percentage of cycles the load/store functional unit is stalled because of resource limits in the memory system. While some workloads like SAD make heavy use of both computation and memory resources, most either saturate one of compute or memory.

As a way to increase utilization, previous work has developed methods for running multiple kernels on the same GPU. Simultaneous Multi-Kernel (SMK) [141, 146] is a technique allowing multiple kernels to execute on the same SM, similar to the way simultaneous multi-threading on a CPU allows multiple threads to execute on the same core. When workloads with complementary resource demands are scheduled on the same core, the throughput of the GPU can be higher than running the workloads sequentially. SMK does this by splitting the GPU's warp contexts between

the workloads.

5.2.2 Disadvantages of Temporal and Spatial Partitioning

Besides SMK, temporal and spatial partitioning also allow a GPU to be divided between workloads. *Temporal partitioning* devotes the entire GPU to one application at a time. This method has been used to share a GPU between compute tasks and high-priority graphics rendering tasks [63] and in other situations where a GPU is split between short-lived tasks, such as rendering graphics frames for multiple users on a server GPU [45]. For non-graphics workloads, sharing a GPU with temporal partitioning allows for a precise split of GPU resources and time, since an allocation of 90% of the time on the GPU would correspond to 90% throughput of running alone. The drawback of temporal partitioning is that there is no throughput advantage from dividing the GPU. For a public cloud provider, this offers no utilization advantage over selling GPUs as a unit.

In contrast, other methods of sharing the GPU are able to increase overall throughput, such as *spatial partitioning* [3], which commits SMs to workloads as a unit. For workloads that do not need all the SMs to achieve much of their performance, such as workloads that are limited by memory bandwidth or have a limited number of threads, spatial partitioning shows benefits over temporal partitioning. However, spatial partitioning can strand unused resources inside SMs and can only allocate resources at the granularity of entire SMs. Unlike under temporal partitioning, workloads can interfere with each other in the shared global memory resources.

5.2.3 Interference under SMK

Using SMK rather than temporal or spatial partitioning can lead to higher throughput when sharing because it can partition resources in a very fine-grained way, but it also exposes the workloads to more potential interference with their co-runners. Figure 5.3 shows the relative performance of `sgemm` (SG) when co-running with other workloads with resources divided equally between them using SMK. When running with LK, both SG and LK meet or exceed 50% of their performance. However, when SG runs alongside SAD and MUM, although there is an overall throughput boost

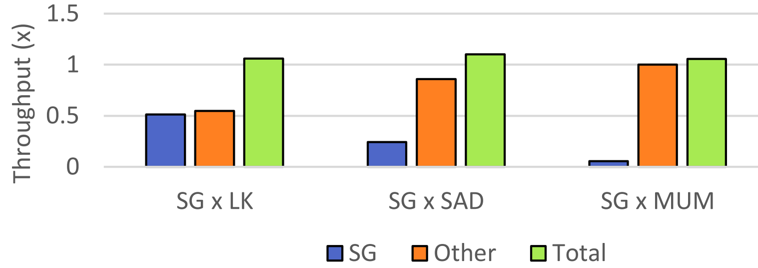


Figure 5.3: Running multiple kernels using SMK results in interference. The SG benchmark is run alongside three other benchmarks, sharing resources evenly. Interference causes throughput loss for one or both workloads.

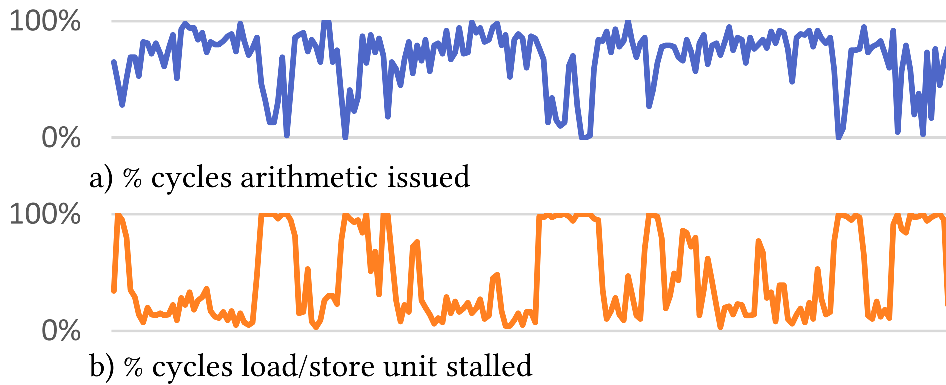


Figure 5.4: Timeline of % cycles arithmetic issued and load/store unit stalled, averaged over 100-cycle windows, for a 20,000-cycle interval of BP. A co-running workload is able to issue compute and memory instructions at times of low utilization.

from sharing resources, these other workloads are more aggressive and reduce its throughput. This is consistent with studies of sensitivity and contentiousness in CPU SMT systems like [150].

Besides the ability of aggressive kernels to overwhelm others, interference is problematic because it is difficult to predict ahead of time. First, kernels have phases of execution, which makes their resource utilization change over time. Figure 5.4 shows the resource demands of a fragment of the BP benchmark. Although it issues arithmetic instructions most cycles and is limited by memory throughput some cycles, there are also many other times were another application could insert arithmetic or memory instructions of its own without interfering. Because of this variation, the degree to which a co-running workload would be complementary or interfering is difficult to predict.

The design of the global memory system also makes interference hard to determine. Memory

system resources are on the other side of the interconnect and the L2 is partitioned into several shards in a decentralized way. There are many different queues in the memory system as well as caches, MSHRs, and other shared resources. This means that the interference happens outside of the SMs, and it may take on the order of hundreds of cycles for memory requests issued from an SM to interfere in a queue or MSHR in the memory system.

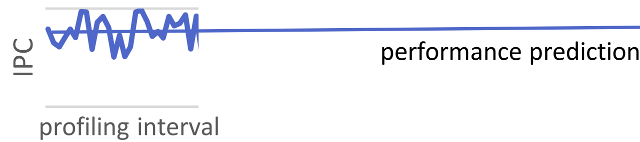
5.2.4 Opportunities to Control Interference

In light of these challenges, interference must be controlled in particular ways to successfully share a GPU between customers in a cloud environment in a way that benefits customers. Rather than maximizing throughput or minimizing turnaround time, a useful scheme from the clients' point of view will offer a level of performance for shared workloads, so that when they purchase time on a GPU they can rely on that performance. This level of service must be relatively high, such as 90% of the throughput of running alone on the GPU, since customers are using GPUs because of their high performance. In hardware, controlling interference means protecting this high-performance workload from disruption from other workloads.

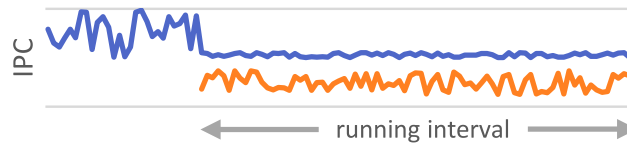
By running this *primary* workload at 90% rather than 100% of its baseline throughput, more opportunities will open up for a *secondary* workload to find a throughput advantage using SMK. With temporal partitioning, the secondary workload would achieve 10% of its baseline performance, but because of the throughput benefit of SMK, more performance will often be possible. Although the secondary workload will have much lower performance than if it ran alone, the cloud operator can sell these cycles cheaply due to the surplus throughput, creating a tier of service like spot instances which are useful for batch workloads. The extra 10% margin will also provide slack for the runtime system to ensure the target is met.

Scavenger implements a system for allocating access to SM resources between the primary and secondary workloads. Since interference is caused by both workloads attempting to use the same resources at the same time, controlling interference involves partitioning resources dynamically between the workloads. In practice, this will mean allocating as few SM resources to the

① Scavenger allocates all resources to primary workload to determine its performance when run alone.



② Resource allocators distribute resources between workloads to achieve the primary's performance target while maximizing the secondary's throughput.



③ When performance counters signal a change in the primary kernel's characteristics, a new profile is collected.

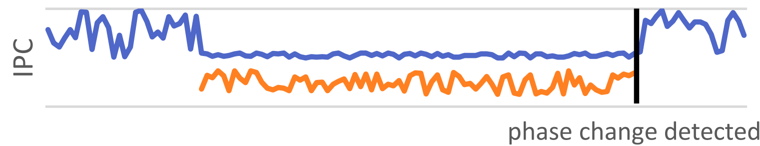


Figure 5.5: Overview of the Scavenger system

primary workload as possible before dropping below the performance target, so that the secondary workload can use more resources to increase its throughput. Every part of this system must be performed online, as clients can provide any workloads to run on the cloud and cannot be counted on to provide representative inputs for an offline profiling phase.

5.3 Overview

An overview of how Scavenger performs online resource allocation inside each SM is shown in Figure 5.5. Execution is divided into two states, profiling and running. The first stage of execution is a profiling interval, where the primary workload is run alone to measure its target IPC. Then, during the following running interval, the target IPC is used as input to resource allocators which partition compute and memory resources between the two workloads to both achieve the performance target for the primary workload while maximizing the throughput of the secondary. Finally,

Scavenger must detect when the performance information from the profile interval no longer can predict current performance and start a new profile phase. To implement this runtime system, Scavenger has two components: a performance predictor and a set of resource allocators.

5.3.1 Online Performance Prediction

The first component, a performance predictor, determines the primary workload's throughput to achieve the performance target. The performance target is specified by the cloud operator as a percentage of the workload's throughput when running with all a GPU's resources (e.g., 90% of a dedicated GPU).

To translate the target percentage to a measurable target IPC, Scavenger runs the primary workload alone for a short period of time and extrapolates the average IPC forward. With this technique, there is no throughput loss relative to temporal partitioning during profiling periods, although there is also no throughput boost due to SMK. GPU kernels tend to have average IPCs that are even over the long term, even if they show short-term oscillations or noise, allowing this technique to have reasonable accuracy. However, it is important to detect when a new profile is needed should a new kernel start or the current kernel change phases, for example when blocks have finished loading data into shared memory and a computation phase begins. By tracking performance counters for kernel characteristics that co-runners cannot interfere with, Scavenger is able to detect when a phase change has occurred and collects a new profile in response.

5.3.2 Dynamic Resource Allocation

The second component of Scavenger allocates SM resources between the two workloads. The two resources that need to be allocated are access to the computation units and access to memory. Controlling access to computation resources requires adjusting how many thread contexts are allocated to each kernel. Any changes to this allocation require costly context switches to memory. Therefore, layered on top of thread block preemptions is a mechanism for deactivating some thread contexts from either kernel. This allows Scavenger to adjust the ratio of thread contexts devoted to

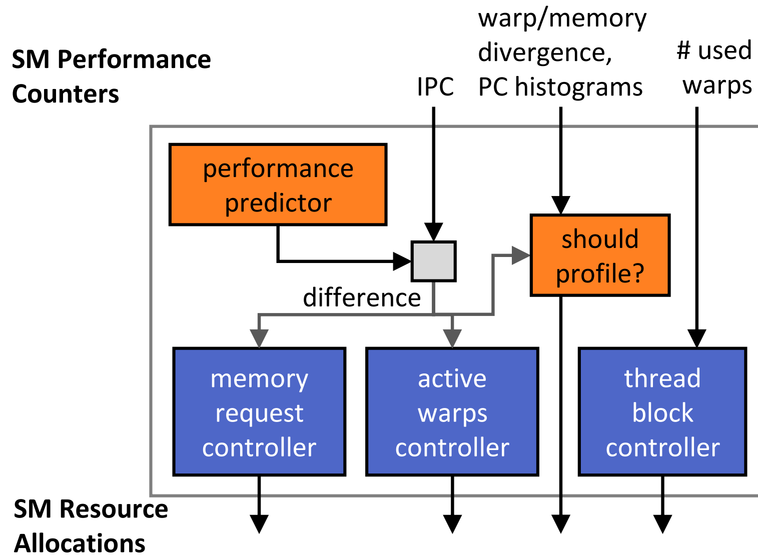


Figure 5.6: The Scavenger components in each SM, which use performance counters to determine resource allocations. The upper components (in orange) predict the primary workload's performance and detect when it has entered a new phase. The lower components (in blue) adjust the resource allocation to achieve the primary workload target and maximize secondary workload throughput.

each kernel that may issue instructions without the need for preemption.

The other resource, access to memory, is allocated as a number of requests each workload is allowed to have outstanding in the memory system. This allows the SMs to control memory contention without the need to communicate across the interconnect to the global memory resources directly, following work by Dai et al. [23]. The system evaluated in this paper has the L1 cache disabled as it is in the GTX 980 system that is modelled, so cache resources do not need to be allocated; a system that distributed cache would need more complex compute and memory resource allocation schemes because fewer resources would sometimes result in higher performance [52]. In GPU architectures, the L2 cache is used for filtering duplicate requests rather than reducing latency, so it does not need explicit management.

5.3.3 Hardware Components

Figure 5.6 shows an overview of the Scavenger system, integrating both components for performance prediction and resource allocation. The difference between the predicted and actual per-

formance for the primary workload informs the controllers whether to increase or decrease the primary workload's resource allocation. Interference-invariant statistics like warp and memory divergence allow the predictor to detect new phases or restart the profiling process. Controllers determine the allocation of resources in the SM – controllers for the number of memory requests and number of active warps are used to react quickly to changing performance, whereas the thread block controller tracks the longer-term number of warps issuing instructions in order to make preemption decisions.

Scavenger is a decentralized system that controls each SM separately. Global coordination between the SMs in a GPU is difficult because no mechanisms are provided for communication between them in the baseline system, so this distributed design avoids adding new coordination between SMs. Scavenger is also designed to share resources between exactly two workloads, as the high performance target for the primary workload leaves few resources to divide among multiple batch workloads. Other techniques, like selecting the most complementary workloads to run on the same GPUs, can therefore improve utilization more effectively than increasing the number of workloads that share the GPU.

The next sections will detail the two main systems in Scavenger: performance prediction and resource allocation.

5.4 Online Performance Prediction

In order to meet the performance target for the primary kernel, Scavenger must determine what the performance of a kernel running alone would be. Since the primary and secondary kernels interfere in complex ways when running together, it would be difficult to infer what the primary kernel's performance would be if run alone from data collected while sharing the GPU between workloads. The flexible scheduling between threads makes this especially difficult for a GPU, since interference not only introduces latency and contention for functional units but also changes the possible instruction interleavings between threads.

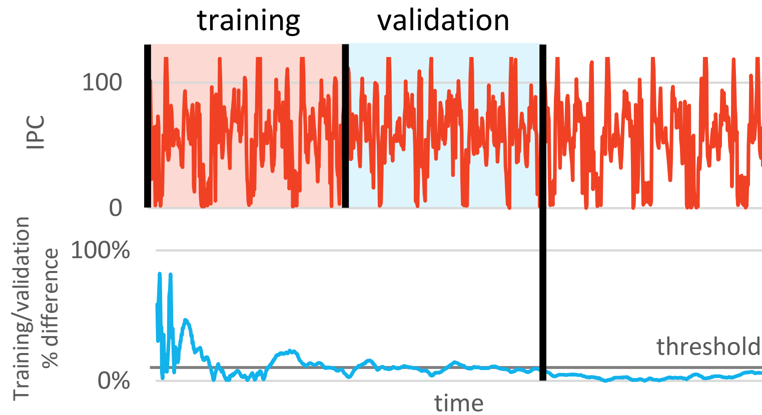


Figure 5.7: Performance of excerpt of HS over time along with the difference between the mean training and validation interval IPCs. To detect an appropriate profile length, Scavenger continues profiling until the difference stays below a threshold.

Because of this, Scavenger analyzes the primary kernel's performance during intervals of time in which the primary kernel is given the full resources of an SM. The primary kernel is run alone for enough time to account for performance volatility, then the mean profiled performance is used as a prediction for future performance. Because kernels often have phases of execution, the profile cannot be relied upon past the end of these phases. To detect when a phase ends, Scavenger uses runtime counters to track statistics about kernel execution that cannot be affected by interference, such as warp and memory divergence. Significant changes in these counters signal the beginning of a new phase. However, a first challenge is determining a profile length that minimizes overhead by not lasting any longer than is necessary to generate an accurate performance prediction.

5.4.1 Determining Profile Length

The number of cycles needed to profile a representative same of execution differs for each workload, as performance that is stable over the long term may nevertheless exhibit short-term noise. As an example, in the 2000-cycle window of HS in Figure 5.7, the IPC varies significantly but has a stable average. To discover the length of time needed to sample the long-term performance, Scavenger divides the profiling state into two intervals, a training interval and a validation interval. The mean IPC in the training interval is continuously compared to the mean IPC of the validation

interval. If they are far apart, more profiling is necessary. The bottom of Figure 5.7 shows how Scavenger detected a reasonable profile length when the percent difference stayed below a threshold for a minimum number of cycles. To avoid having earlier minima stop the profile too early and create larger error later, profiles must exceed a minimum time (see Table 5.1 for parameters). The training and validation IPC difference threshold is tunable to achieve a tradeoff between profiling time and accuracy, as evaluated in Section 5.6.4.

Some workloads have unpredictable performance patterns. Scavenger detects this when the difference between the training and validation interval IPC means does not settle below the threshold. The means can also fail to stabilize when there is a phase change while profiling is taking place. In these cases, after a maximum profiling time, the accumulated training data is discarded and a new profile stage is started. The effect of this is that Scavenger does not attempt to share GPU resources when the primary workload's performance is unpredictable, in effect falling back to temporal partitioning.

Because profiling does not require any instrumentation beyond hardware performance counters, no throughput is lost relative to temporal partitioning. However, while profiling there is no throughput gain possible from sharing the SM, and there are preemption costs associated with switching to the profiling state as all blocks of the secondary kernel must be context switched out.

5.4.2 Detecting Phase Boundaries

Predicting future performance based on a past profile period will only be accurate when current execution is similar to the behavior during the profiling interval. Kernels often have phases of execution, such as a phase where data is loaded into shared memory, a phase of computation, then a phase storing the results. The performance can change dramatically between these phases, so a profile collected during one phase should not be used to predict performance in another. Since the IPC of the primary kernel running alone cannot be determined while both workloads share an SM's resources, Scavenger tracks statistics that are not affected by a co-runner to detect whether the current behavior of a kernel is the same as when it was profiled. These statistics include

the number of active lanes in issued instructions, which is a measure of control divergence, the degree of memory divergence in load and store instructions, and the PCs of the instructions being executed. If any of these are markedly different than their values during profiling, a phase transition has likely occurred. Run intervals are also ended after a maximum length of time.

During profiling, just as the average IPC is sampled, values for the average number of active threads and loads/stores per memory instruction are collected along with a sample of the distribution of PCs executed. After profiling, during the run state, these same statistics are collected. Their average values are compared periodically to the values found during profiling. If they differ by more than some threshold, Scavenger detects a phase boundary and collects a new profile.

Some workloads have more variation in these values over time than others. The train and validation interval mechanism used for performance also can be used to find useful thresholds for when the change in these values indicates a new phase. At the end of profiling, the train and validation intervals have a similar average IPC, so they can be assumed to be part of the same phase. Therefore, the thresholds used to detect a phase change must be high enough that the difference between the validation and train intervals does not trigger a boundary. The phase detection thresholds are set to the difference between the average values of the train and validation intervals, plus a 10% margin. To avoid spurious phase change detections, these thresholds have a minimum value, the run state must last for at least as long as the profile state, and the values are averaged inside a sliding window.

While control and memory divergence can be sampled with performance counters, determining whether different distributions of PCs are executed in different intervals is a more difficult task. Scavenger creates *PC histograms* to track these distributions. PC histograms are implemented as a vector of 128 counters, with an instruction incrementing the counter for its PC divided by 8 (the instruction size), modulus the number of counters. Each PC histogram also has a counter tracking the total of all values in the counters, to allow for histograms to be normalized to each other.

Minimum profiling time	20000 cycles
Maximum profiling time before profile restart	200000 cycles
% of profile used as validation interval	50%
Minimum active thread difference threshold	8
Minimum memory divergence difference threshold	2
Minimum PC histogram difference threshold	12K of 64K range
Phase detector window size	128
Run length maximum	262144 cycles
Window size for phase boundary statistics	8192 cycles
Time above threshold to trigger phase	500 cycles

Table 5.1: Performance predictor parameters

5.5 Performance Controllers

Once the performance goal for the primary kernel has been determined, the allocation of SM resources must be adjusted between the primary and secondary kernels to achieve the performance target while maximizing the secondary kernel's throughput. Because a kernel's performance characteristics and resource usage vary significantly over time, any resource allocation solution should be dynamic, leveraging low resource utilization times in the primary kernel to boost the performance of the secondary kernel.

Managing access to resources is done in two levels. The first level makes rapid adjustments to lightweight resource allocations. To adjust access to compute resources, this first level activates and deactivates individual warps to adjust the ratio of active warp contexts between workloads. A similar controller adjusts memory resource by varying the number of memory requests each workload can have outstanding in the memory system. On the second level, another controller uses preemption to adjust the number of thread blocks to match the number of contexts needed in the long term.

When finding a resource allocation, Scavenger must be careful not to take away resources from the primary workload without a corresponding gain in performance in the secondary workload. As an example, reducing the primary workload's memory resources might bring it closer to a 90% performance target, but the secondary workload would need to translate those memory resources into 10% of its own baseline performance in order for there to not be overall throughput loss. To ensure this does not happen, Scavenger measures *slack*, the amount of the resource allocated to the

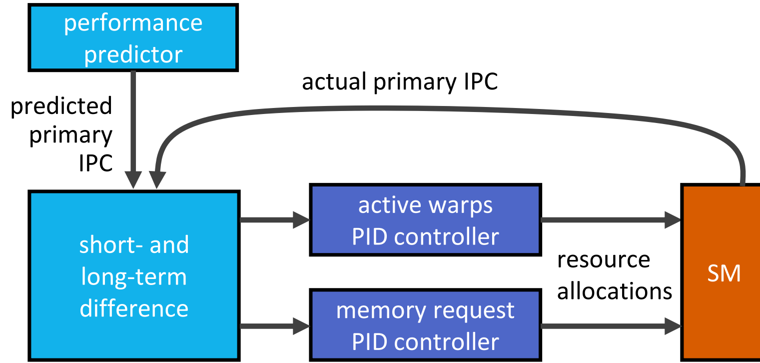


Figure 5.8: Feedback control system for active warps and outstanding memory requests. The short-term and long-term difference between the predicted IPC and actual IPC is used to adjust resource allocations using PID controllers.

primary kernel that is not used by it. The slack for each resource is tracked and used to determine which resource allocations to adjust. Further, should Scavenger detect that sharing SM resources is leading to throughput loss, it will fall back to temporal partitioning.

5.5.1 Controlling Warps and Memory Requests

The first level of resource allocations are made by feedback controllers that make frequent adjustments. These allocations, for the number of warps from each workload to activate and the number of memory requests each workload can have outstanding in the memory system, can be made using well-studied PID controllers for online feedback control. PID controllers take as input the difference between an actual and desired output, and adjust an input value to minimize that difference. In this case, the difference between actual performance and the predicted performance is used as input, with the resource allocation to the primary workload as output. Because the resource allocation affects performance, this forms a closed-loop system.

Figure 5.8 shows the general form of each of the two feedback controllers. The performance predictor provides a target IPC, which the PID controller seeks to match. Scavenger computes the difference between the predicted IPC of the primary kernel and the measured IPC while running. It computes two window sizes: a short-term error used for the proportional component of the PID controller, and a long-term error used for the integral term instead of the controller having an

internal counter for integral state. No derivative component is used, because of the large amount of noise in the input. The interval at which the PID controllers are cycled as well as empirically found gains are listed in Table 5.2.

5.5.1.1 Active Warps

The thread block controller allocates warp contexts to workloads, but has to do so at the granularity of entire thread blocks. A lighter-weight way to adjust the ratio of warps between workloads is possible by enabling and disabling individual warp contexts. One PID controller outputs the number of the primary kernel's warps to activate, with any remainder being allocated for the secondary. Therefore, when performance is too low, more of the primary's and fewer of the secondary's warps are activated, and the reverse when performance is too high.

5.5.1.2 Outstanding Memory Requests

To limit memory interference, another controller divides outstanding memory requests between the two workloads. A workload is prevented from issuing global load or store instructions when it is at or above its allocation of outstanding requests. The challenge making this allocation comes from variation in the total number of outstanding requests due to workload characteristics. For example, MSHR merges in the L2 cache allow for more outstanding requests. To overcome this, Scavenger divides a fixed number of outstanding requests, 256 (found empirically as a maximum from the baseline system), between the two workloads and each workload can send up to its allocation of requests to the memory system. Unused requests are allocated to the primary kernel.

5.5.1.3 Weighting PID Control

Because both PID controllers have the same input signal, they would move in lockstep without a mechanism to attribute slowdown or speedup to one resource rather than the other. As a mechanism to detect whether the performance deviation from the estimate is due more to the allocation of active warps or memory requests, slack metrics are used. Intuitively, when resources need to be

taken away from the primary kernel, the resource chosen should be the one with the most slack, and when resources need to be added, the resource chosen should be the one with the least slack. Therefore, when the primary kernel is running too fast and resources can be taken away, the PID controller gains are scaled by the amount of slack, where more slack means a bigger change can be made. When the primary kernel is running too slow, the weight is inverted and smaller slack creates a larger change.

For the active warp controller, the slack metric is the average percentage of the primary kernel's warps that have an instruction ready to issue since the PID controller was last cycled. For outstanding memory requests, since more outstanding requests can be allocated to the primary kernel than are used, Scavenger records the total number of outstanding requests on cycles when the load/store unit is stalled, as a way to determine the actual maximum number of outstanding requests for the workload pair. The slack is this recorded number of outstanding requests minus the average number of outstanding requests for both primary and secondary workloads in the memory system, as a percentage of the recorded number of outstanding requests.

5.5.2 Controlling Thread Blocks and Preemption

To distribute access to computation resources, Scavenger splits the warp contexts in an SM between kernels, allowing the warp scheduler to select which warps have access to ALUs in individual cycles. Switching which kernel occupies a warp context involves preempting the kernel at the granularity of a thread block, since warps in a block can synchronize with each other and SM resources like shared memory and registers are allocated to blocks, not individual warps. Preempting a block involves saving tens of kilobytes of registers and shared memory state to global memory, so it is important to minimize the amount of preemption done.

Scavenger uses a slack metric to determine how many blocks of each kernel should be running at a given time. At an intuitive level, neither kernel should occupy warp contexts that are not contributing to performance by issuing instructions. Therefore, the slack metric for warps tracks how many warps allocated to a kernel did not issue any instructions in a previous window. If this

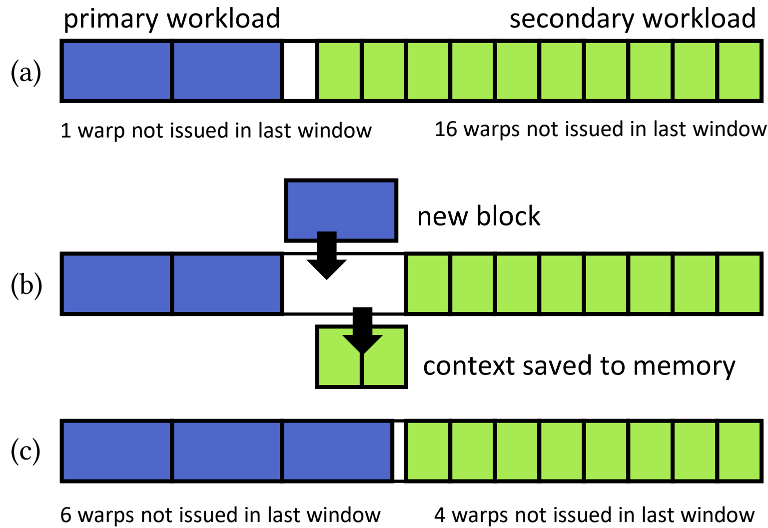


Figure 5.9: The thread block controller preempts blocks to balance warp slack between the workloads. (a) Scavenger detects too little slack for primary workload, too much for secondary. (b) Blocks of secondary workload are preempted to make way for primary. (c) Slack is more evenly distributed between workloads.

number is too high, then warps can be taken away from the kernel and given to the other. Figure 5.9 illustrates this process. In (a), the primary workload has too little slack, as all but one of its warps was used in the last window. Therefore, Scavenger redistributes the blocks using preemption (b), which results in a more even distribution of slack warps (c). The controller only selects between allocations that do not have enough empty space for an additional block of either kernel and that include at least one block of the secondary kernel; section 5.5.3 details how the system detects when sharing with this allocation is not profitable.

The detailed algorithm is as follows: If the primary kernel's slack is below a threshold (see Table 5.2), then an additional block is allocated to it, to avoid starving the primary kernel and leading to throughput loss. Otherwise, if the secondary kernel's slack is below its threshold and the secondary's is above its threshold, the excess slack from the primary is allocated to the secondary kernel. Finally, if the secondary has above its threshold of slack, the excess slack is allocated back to the primary, to ensure that the secondary only has the least number of warps it will actually use. The maximum change each time the thread block controller runs is one quarter the difference in slack, to make sure the controller does not overshoot its target. After a transition back to the

Block controller interval	8192 cycles
Primary minimum warp slack	4 warps
Secondary maximum warp slack	8 warps
PID controller interval	100 cycles
PID controller proportional gain	1.5
PID controller integral gain	1.0

Table 5.2: Controller parameters

SMs	16 SMs, 4 LRR warp schedulers each, 64K registers, 96kB shared memory, 2kB L1I
Warps	64 warps per SM, 32 thread blocks
Memory system	L1 cache bypassed, 2 MB L2, 4 memory partitions, 224 GB/s B/W
Performance predictor tolerance	7.5% IPC difference between training and validation interval

Table 5.3: Simulator configuration

running state from profiling, the secondary kernel is allocated half the warps it did before profiling, and allowed to discover a more aggressive allocation over time.

5.5.3 Avoiding Throughput Loss

Because at least one thread block of the secondary kernel is always provided by the thread block controller, there can be cases where no resource allocation by the controllers can meet the performance target. In these cases, continuing to attempt to share resources will lead to throughput loss, so the best option is to fall back to temporal partitioning by devoting all SM resources to the primary workload. Scavenger detects this case with a performance counter that tracks the cumulative difference between the predicted and actual IPC. If the actual number is too far below the estimate for an extended period of time, this indicates the performance controllers were not able to achieve the target by changing the resource allocation, and Scavenger preempts all warps of the secondary kernel and run the primary kernel alone for a length of time before collecting a new profile and trying to share resources again.

	Name	Type	Source
BFS	bfs	MEM	[17]
BP	backprop	MEM	[17]
BS	blackscholes	MEM	[99]
BT	b+tree	MEM	[17]
FWT	fastWalshTransform	MEM	[99]
HS	hotspot	COM	[17]
HW	heartwall	MEM	[17]
KM	kmeans	MEM	[17]
LBM	LBM	MEM	[129]

	Name	Type	Source
LK	leukocyte	COM	[17]
LM	lavamd	MEM	[17]
LUD	LUD	MEM	[17]
MUM	mummerGPU	MEM	[17]
SAD	SAD	COM	[129]
SG	sgemm	COM	[129]
ST	stencil	MEM	[129]
TP	tpacf	COM	[129]

Table 5.4: Benchmarks

5.6 Evaluation

5.6.1 Methodology

Scavenger was evaluated using an implementation in GPGPU-sim 3.2 [9] extended for multi-kernel execution. The parameters used are shown in Table 5.3. Memory requests were set to bypass the L1 cache, as is the default for the GTX 980. To add L1 cache allocation to a system like Scavenger, there would need to be compensation for speedups produced by reducing cache thrashing by restricting the number of active threads; other work such as [139] has studied the considerations needed when dividing cache resources. LRR warp scheduling is used instead of the usual GTO because greedy scheduling algorithms do not divide access to compute resources in a way that respects the proportion of warp contexts allocated to each workload. Other work [108] has proposed methods to extend greedy schedulers to issue in a more proportional way. The time needed for thread block preemption was modelled conservatively by assuming contexts are saved to DRAM.

The benchmarks evaluated are shown in Table 5.4. These workloads were selected to be similar to previous work in GPU multitasking [108] and present a mix of workload characteristics. Workloads are classified as compute-intensive (COM) if when run alone they issue compute more than 50% of the time, as shown in Figure 5.2. Each benchmark was used as a primary workload, with six secondary benchmarks selected randomly as co-runners – three with COM type and three with MEM type. Each pair was run until completion or 500 million instructions of the primary workload had retired; the secondary workload was repeated if it completed before the primary workload. The measured performance includes both profiling and running intervals. As men-

tioned in Section 5.3.3, Scavenger focuses on the two-kernel case because reaching a high target for the primary workload does not leave enough unallocated resources to divide among multiple secondary workloads.

5.6.2 Hardware Implementation

Implementing Scavenger in hardware requires creating sliding windows, counters, and PC histograms for both the profile and run states. Sliding window averages are implemented using shift registers with a total, with each value in the shift register accumulated over a number of cycles. The PC histograms are implemented with 128 buckets alongside a total counter. Because the profiling state is a variable length, its counters and PC histograms are implemented using one large counter or histogram for the first half of the profiling interval, and 4 smaller counters and histograms for the second. The four smaller counters are arranged as a circular queue, with the tail counter combined with the large first half counter when the counters are rotated. The amount of time between rotations varies to keep the number of cycles of data represented in the large counter and the four smaller counters the same.

Comparing PC histograms requires normalizing them, as the comparison is a test for whether they form similar distributions rather than if the absolute values are similar. The comparison value used is the sum of the differences of each bucket, with the histogram with the smaller total shifted to match the same magnitude as the larger total. Because the need for comparisons are infrequent (every 8192 cycles in the evaluated design), the difference can be computed over multiple cycles.

The total overhead is 1.6KB of storage per SM. 1.2KB is used for profiling (which includes 1KB for the 5 PC histograms). The performance validation logic requires two sliding windows and a PC histogram, for a total of 338 bytes, and all other counters for the PID controllers, block controller, and other counters require under 15 bytes. For perspective, each SM contains a 256KB register file and hundreds of FPUs.

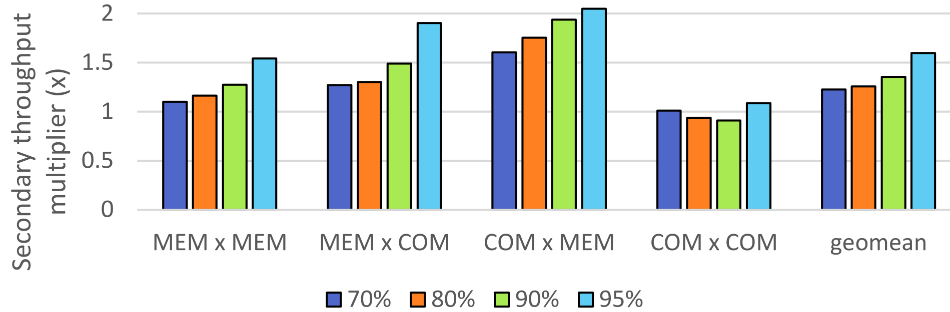


Figure 5.10: Secondary workload throughput with Scavenger compared to temporal partitioning, by primary x secondary workload category. Pairs violating the primary kernel performance target are excluded.

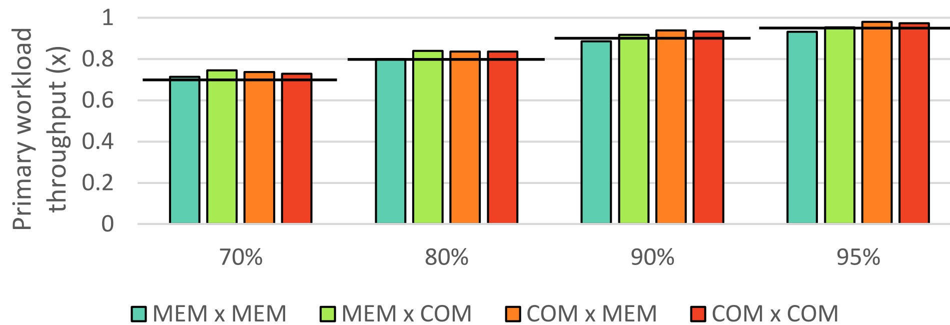


Figure 5.11: Geomean primary workload throughput by pair category and target

5.6.3 Performance Targets and Throughputs

Temporal partitioning can be used to achieve the primary workload's performance target but limits the secondary kernel's throughput to 100% minus that target. Figure 5.10 shows the multiplier on the secondary workload's throughput supplied by Scavenger over temporal partitioning, as the geometric mean of the pairs of workloads matching each category. As an example, for MEM x MEM, Scavenger achieved 12.7% of the secondary workload's performance when running alone with a 90% performance target for the primary, leading to a 1.27x throughput multiplier. This can be viewed as a return on the investment made with the primary kernel's resources. Across the categories at a 90% primary performance target, Scavenger unlocked 1.35x more throughput for the secondary workload.

The tighter primary performance targets saw a larger throughput multiplier. This is because part of the throughput increase is due to synergies that do not cause interference, such as a com-

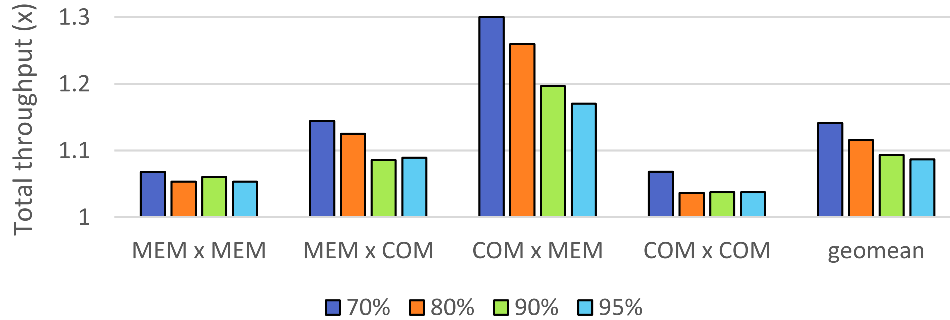


Figure 5.12: Total primary and secondary throughput with Scavenger compared to temporal partitioning. Pairs violating the primary kernel performance target are excluded.

pute phase and memory phase running at the same time. These can be present regardless of the resource allocation, but other gains come from times when resource allocation must control interference, which is why looser targets like 70% see higher overall throughput in Figure 5.12. The COM x COM pairs sometimes show lower throughput than temporal partitioning. In these cases, resource allocation is a nearly zero-sum partition between the workloads, and Scavenger allocates slightly more resources to the primary workload than necessary because the design of the system for avoiding throughput loss errs on the side of achieving the performance target for the primary workload.

While Scavenger increases the secondary workload's throughput, it must keep the primary workload near its target performance. Figure 5.11 shows the geometric mean primary workload throughput across each type of pair. Across every type of pair except MEM x MEM at 90% and 95%, where the throughput is within 2% of the target, this throughput is above the target. The primary workload throughput may differ slightly from the target for a number of reasons. Since the performance predictor only profiles for a segment of time, there is a small amount of error in its predictions. The resource allocators attempt to match the prediction, so this error is propagated to the final throughput. Second, the relationship between performance and resource allocation differs widely between pairs, so it can take more time for the allocators, especially the block allocator, to find an optimal resource allocation. In times where there is synergy independent of the resource allocation, the resource allocators will not slow down the primary kernel when taking resources

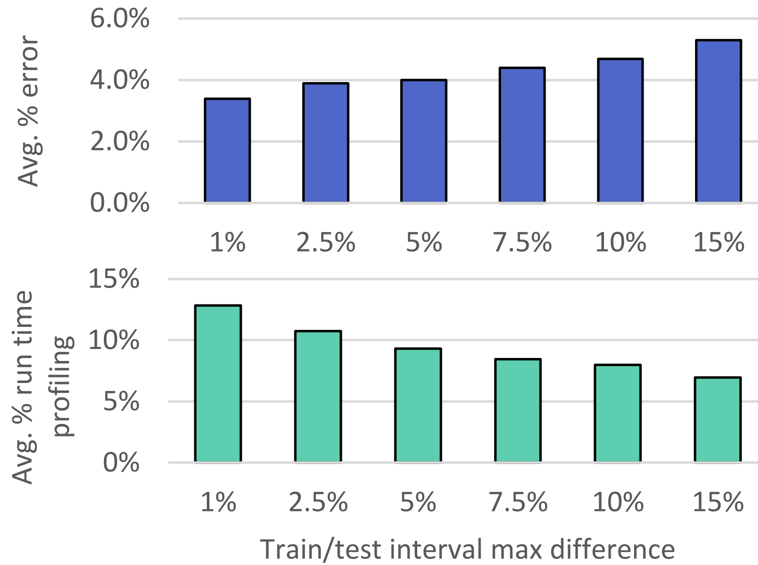


Figure 5.13: Cumulative % error and % time profiling, averaged across benchmarks vs. maximum difference between training and validation interval IPC while profiling

away, which leads to higher primary workload throughput than the allocators expect. From a cloud client's perspective, it is better to exceed the primary workload throughput than to be below it.

Figure 5.12 shows the overall throughput multiplier from using Scavenger to share the GPU rather than temporal partitioning. Here, the lower performance targets are able to unlock more total throughput as less of the potential throughput gain must be given over to maintaining the performance target. The heterogeneous pairs (MEM x COM and COM x MEM) see the largest gain as their complementary characteristics allow them to more fully utilize GPU resources. Both the COM x COM and MEM x MEM pairings see a modest throughput increase, as although the heterogeneity is limited, there is still more than when all the threads come from a single workload. At the 90% primary performance target, Scavenger increased the GPU's throughput by a geometric mean 1.09x across the categories.

5.6.4 Performance Predictor Accuracy

The tradeoff between the amount of error and the percentage of time spent profiling can be controlled by increasing or decreasing the tolerance for IPC difference between the training and valida-

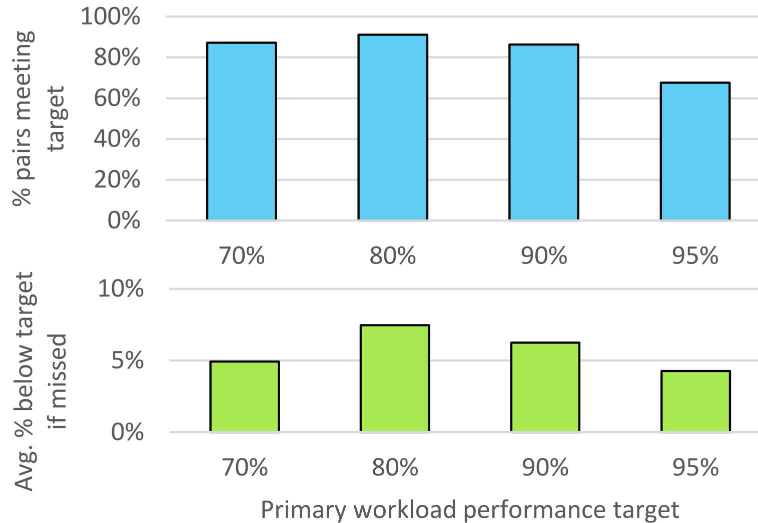


Figure 5.14: Percentage of pairs where the primary kernel's performance was below the target with 1% error margin, and the average percentage by which pairs that did not achieve the target were below the target performance

tion intervals while profiling. Figure 5.13 shows how the profiling error and profiling time changes as that tolerance changes, averaged across each primary workload. Increasing the tolerance decreases the percentage of execution spent profiling, but increases error. The tolerance used for the throughput experiments in this section was 7.5%, as it had below 5% error while still profiling less than 10% of the time. The amount of time the training-validation IPC difference must stay below the threshold before ending the profile also has a major influence on both the amount of error and the time spent profiling – this result used 128 cycles below the threshold, but 256 cycles can decrease the error below 2% at the expense of nearly doubling the profile time.

5.6.5 Performance Targets Achieved

Figure 5.14 shows the percentage of pairs that met the primary performance target for each target percentage, as well as the amount by which the workloads that missed the target were below it. A 1% margin of error is used for determining whether the target was hit. For 70%, 80%, and 90%, Scavenger hits the target in between 86% and 91% of runs, whereas at the more aggressive 95% target, Scavenger achieves it 67% of the time. There are two mechanisms that can lead to

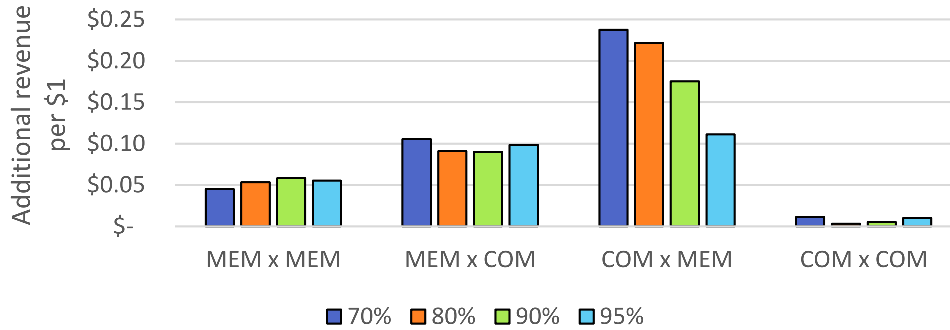


Figure 5.15: Additional revenue per dollar realizable with Scavenger over leasing GPUs as a unit or using temporal partitioning.

Scavenger missing the target. First, if the performance predictor provides too low a target to the controllers, the controllers do not know that they should allocate more resources. Second, because sharing must be attempted for a time before falling back to temporal partitioning, aggressive primary performance targets like 95% are missed because of the overhead of detecting the throughput loss and of the preemptions needed to switch to temporal partitioning. As an online system, some level of error is unavoidable. The 86% of the time Scavenger reaches the 90% QoS target is comparable to prior work [142] that meets this target approximately the same percentage of the time, despite being supplied an IPC goal ahead of time.

5.6.6 Cloud Operator Revenue

The throughput gains achieved by Scavenger are able to supply a cloud operator with additional profit over using temporal partitioning or selling time on GPUs as a unit. Figure 5.15 shows how using Scavenger supplies additional revenue. This model assumes that the client with the primary workload pays for their fraction of performance (e.g. $0.70 for 70% performance), even if Scavenger exceeds that level of performance. For this simple model, the batch tier pays $0.01 per percentage of its performance running alone that the system achieves. Some of the additional value created could be used to discount batch tier service to compensate for decreased performance of batch jobs relative to running them alone, with the rest captured as profit. The results show that matching complementary workloads is key to realizing maximum revenue, with a COM workload

able to gain up to an extra $0.23 per dollar when matched with a MEM secondary as opposed to another COM.

5.7 Related Work

Multi-Application GPUs: Several previous works have detailed how to run multiple applications simultaneously on a GPU while optimizing for overall throughput or fairness. One line of research uses compiler techniques to interleave instructions from multiple kernels. Guevara et al. [40] merged the source code of multiple kernels into one. Another set of techniques scheduled data transfers and kernel executions for multiple kernels or applications on a non-preemptible GPU. Rossback et al. [120] created a dataflow programming model that can build schedules with fairness guarantee. A third stream developed ways to execute multiple applications simultaneously with hardware support. Adriaens et al. [3] showed throughput benefits with spatial partitioning. Wang et al. [141] and Xu et al. [146] developed methods to share GPU resources inside SMs. Park et al. [108] developed techniques for finding allocations that optimized throughput or fairness, Jog et al. [57] create a fair DRAM scheduler for co-running applications, and Dai et al. partition memory requests [23], complementing work on CMP memory requests [26]. Preemption techniques detailed by Tenasic et al. [134] and Park et al. [106] allow these resource partitions to be adjusted.

GPU QoS and performance targets: Previous work has implemented quality of service support either in software in the runtime system or device driver, and in hardware for spatial partitioning and SMK systems. On the software side, Kato et al. [63] uses the device driver to provide a soft real-time guarantee for graphics workloads when running with a compute task. Lee et al. [70] build a real-time scheduler for launching non-preemptible kernels, and Chen et al. [21, 20] create systems for building kernel launch schedules that achieve QoS goals while maximizing utilization. In hardware, Aguilera et al. [4] develop a QoS system for spatial multitasking, and Wang et al. [142] create a QoS system for SMK GPUs by adjusting the thread block allocation while tracking performance quota. Scavenger manages not only thread blocks but also active warps and memory

resources, and can determine the IPC goal at run time without it being provided by an OS-level scheduler.

CPU resource partitioning: Multi-core CPUs share memory system resources, including cache capacity and bandwidth. Guo et al. [41] provide a quality-of-service framework that steals resources from jobs with excess resource allocations in CMPs. Nesbit et al. [94] create virtual private caches that prevent threads from interfering with other threads' cache bandwidth, and extend their concepts to virtual private machines in [95]. Xu et al. [145] create a performance model that finds cache allocation sizes for multiple processes online. Lee et al. [71] measure the resource allocations needed to reach a given QoS goal in a CMP. GPUs have different memory access characteristics and caching is useful for different purposes on GPUs, requiring different approaches.

Feedback control has been used for resource allocation in data centers and CPUs. In data center scheduling, Lo et al. [88] use feedback control and offline profiling data to maintain a latency quality of service target while batch tasks also execute, managing cache, bandwidth, network, and power resources. Sharifi et al. [125] use feedback control in the operating system to manage resources inside of a CMP, including cores, cache capacity, and memory bandwidth. Li et al. [81] also integrate feedback control into a CMP. Scavenger's feedback controllers partition resources inside of cores in a finer-grained way, are found in hardware rather than the OS, and are designed for the decentralized GPU architecture.

Online performance estimation: Subramanian et al. [130] estimate performance of applications when run alone on CMPs using cache metrics and memory controller priority. Eyerman et al. [29] track waiting cycles for applications co-running using SMT to estimate alone execution time. Besides alone execution time, online techniques have estimated power consumption [16] using performance counters.

5.8 Conclusion

Data center and public cloud operators must maximize the utilization of their hardware, which increasingly includes accelerators and GPUs. Sharing a GPU between multiple workloads is able to increase their throughput and utilization, but the interference between the workloads must be controlled. Scavenger is a system that controls interference to create two tiers of service on a shared GPU while still increasing throughput: one with a performance target and one for batch jobs. The key techniques enabling Scavenger are an online performance predictor for the primary workload and a set of dynamic resource allocation controllers. Scavenger can increase the throughput of the batch tier by 1.35x relative to temporal partitioning while maintaining a primary workload at 90% of its performance relative to running alone, for an overall GPU throughput increase of 9.3%.

CHAPTER 6

Conclusion and Future Work

6.1 Summary

Since workloads like neural network training and computer vision require both energy efficiency and high performance, accelerators like GPUs have an important role in modern data centers. GPUs are throughput processors optimized for tasks decomposed into thousands to millions of identical threads. Their performance comes from an execution model that allows them to select instructions from any ready thread to keep their functional units highly utilized, since each thread is independent. Their significant efficiency advantage over CPUs comes from minimizing overhead through grouping instructions into vectors and avoiding the need for complex out-of-order execution logic. Although this design can be effective for many workloads, the overhead of moving and storing on-chip data as well as managing access to off-chip memory resources between running applications is often what limits GPU performance and efficiency.

This thesis addressed these data management overheads and bottlenecks by targeting the places where these inefficiencies are most critical in GPU architectures. One of these bottlenecks came from duplicate memory requests issued across different threads, where they could not be merged together before being sent to the L1 cache. Chapter 3 created new opportunities to merge these requests by expanding the window in which nearby requests could be found. The *WarpPool* system this chapter described merged requests made by different load instructions, made possible by the freedom in the GPU programming model to reorder requests made by different independent

threads. Merging these additional requests was able to address the inefficiencies caused by memory divergence and to more efficiently utilize cache resources. As well, queuing requests while they waited for merges allowed prioritizing some warps' access to the cache, reducing thrashing. Across a set of memory throughput limited workloads, *WarpPool* produced a 38% speedup.

Another overhead for GPUs comes from storing the large number of registers needed for the thousands of threads active on each SM. Since the schedulers can interleave instructions from different threads, each thread must have all its state available at any time. Most of a thread's state consists of registers, which means the GPU's register file is very large, up to 256KB per core, at the same time it must be able to sustain many reads and writes per cycle. Chapter 4 of this thesis described a technique, *RegLess*, which reduces the size of register storage. Building on the insight that not all registers are equally likely to be accessed at every point in time, RegLess coordinates which registers are stored in a smaller high-bandwidth structure with the warps eligible to be issued. Registers that are not immediately about to be accessed can be moved into the global memory system via the L1 cache. To coordinate register movement from memory, hardware needs to know which registers will be accessed in the future, which is supplied by compiler directives inserted into the instruction stream. RegLess was able to replace the register file with a structure 25% of the size without affecting average case performance, reducing total GPU energy by 11%.

Finally, in a data center or public cloud setting, increasing GPU utilization is important as low utilization not only leads to extra hardware being purchased but also the ongoing additional electricity expenses to run and cool it. Sharing the resources in a GPU between multiple workloads improves utilization and overall throughput, because complementary workloads can saturate different resources. However, there is difficulty leveraging this technique because co-running workloads can interfere with each other and cause uncontrolled slowdown. Chapter 5 described the *Scavenger* system, which controls interference to create two tiers of service: one with high performance, and one with low cost for batch jobs. To achieve this, Scavenger predicted the performance of the high-performance workload online and used dynamic resource allocators to achieve that target while maximizing the resources provided to the batch tier. Scavenger increased GPU throughput

by 9.3% and the batch workload throughput by 1.35x while achieving a 90% primary workload performance target.

Together the techniques in this thesis comprise a system that improves both the energy efficiency and performance of data center GPUs, increasing their potential to accelerate workloads like neural network training which require ever-increasing amounts of computation.

6.2 Future Work

The directions explored in this thesis open avenues for future work in GPU memory access analysis, cooperation between hardware and software using compiler directives in the instruction stream, multi-kernel execution, and broader GPU architecture.

The memory request merging and prioritization in *WarpPool* depended on characteristics of GPU workloads that are qualitatively different than those in CPU workloads. As an example from this work, because CPU threads often are not running the same code as each other, unlike on a GPU, there would be few to no opportunities to merge requests between threads. For single-threaded workloads, static analysis such as polyhedral compiler optimizations can be used to find the kinds of reuse patterns across time that WarpPool found dynamically across space at run time. Although there has been work on polyhedral analysis when translating sequential loops to parallel processors like GPUs [14], there is opportunity for better compiler analysis of GPU memory requests across threads and transformations to restructure accesses for locality and contiguity across the spatial dimension on GPUs that does not exist on CPUs.

The hardware-software synergy used in *RegLess*, where the global perspective from software allowed the hardware to make precise dynamic allocations, has applicability beyond register storage. Other work has used this style of technique to annotate coarser-grained performance phases with multiple resource requirements [137]. Other opportunities could include annotating computation and memory-intensive sections of code, to give the warp scheduler better ability overlap regions with different characteristics or select warps to activate from a larger set to meet dynamic

resource needs. This could also be applicable in a multi-kernel setting, where small parts of many different workloads could form a work pool, with annotations about each segment's characteristics, and an SM could pick complementary sets of segments from this pool at run time.

For multi-workload execution in a public cloud, finding practical ways to co-locate workloads from different host machines is a promising direction for future work. Google uses large PCIe switches in its cloud infrastructure to connect GPUs with different host machines [42], but currently supports only mappings of 1 host to some number of GPUs. To find complementary workloads to co-locate using SMK, workloads may need to come from multiple host systems. This poses a challenge for moving data to and from the GPU as well as analyzing and matching workloads from a large pool. Motivating this problem, Chapter 5 describes how effective matching of workloads leads to the greatest cloud operator profit.

For GPU designs in general, there is much future work to be done about how to continue to move GPUs from their roots as gaming graphics cards repurposed for other tasks into accelerators for different niches. Deep learning has been one of these applications that has influenced GPU design, with recent NVIDIA architectures including tensor cores specialized for matrix multiplication [44]. Other architectural additions will be profitable for other problem domains like computer vision and database acceleration. Another change to GPUs' design that could increase their applicability to more tasks could be tighter coupling between a GPU and a CPU, decreasing the amount of time needed to move data to the GPU, allowing memory management hardware to be shared between the CPU and GPU and decreasing the size of jobs needed to yield a speedup on the GPU. Although this kind of design has been explored for mobile and embedded systems, there is potential for larger systems to be integrated this way as well.

This thesis showed how managing data movement and storage improved both performance and energy efficiency for GPUs, and lays the groundwork for capturing even more of this large space of opportunity.

BIBLIOGRAPHY

[1] M. Abdel-Majeed and M. Annavaram. Warped register file: A power efficient register file for gpgpus. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 412–423. IEEE, 2013.

[2] M. Abdel-Majeed, W. Dweik, H. Jeon, and M. Annavaram. Warped-re: Low-cost error detection and correction in gpus. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342. IEEE, 2015.

[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[4] P. Aguilera, K. Morrow, and N. S. Kim. Qos-aware dynamic resource allocation for spatial-multitasking gpus. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 726–731. IEEE, 2014.

[5] J. Anantpur and R. Govindarajan. Taming control divergence in gpus through control flow linearization. In *International Conference on Compiler Construction*, pages 133–153. Springer, 2014.

[6] Apple. An on-device deep neural network for face detection. *Apple Machine Learning Journal*, 1(7), 2017.

[7] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu. Exploiting inter-warp heterogeneity to improve gpgpu performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 25–38. IEEE, 2015.

[8] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[9] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[10] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[11] J. Barr. New p2 instance type for amazon ec2 - up to 16 gpus. https://aws.amazon.com/blogs/aws/new-p2-instance-type-for-amazon-ec2-up-to-16-gpus. Accessed: May 2017.

[12] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[13] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[14] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2010.

[15] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with gpus. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 97–108. ACM, 2012.

[16] W. L. Bircher and L. K. John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2012.

[17] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[18] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE.

[19] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, page 13. ACM, 2011.

[20] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32. ACM, 2017.

[21] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGARCH Computer Architecture News*, 44(2):681–696, 2016.

[22] 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 *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 343–355. IEEE, 2014.

[23] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou. Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[24] J. W. Davidson and S. Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 186–195, New York, NY, USA, 1994. ACM.

[25] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[26] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ACM Sigplan Notices*, volume 45, pages 335–346. ACM, 2010.

[27] A. ElTantawy and T. M. Aamodt. Mimd synchronization on simt architectures. In *Proceedings of the 49th annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[28] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt. A scalable multi-path microarchitecture for efficient gpu control flow. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 248–259. IEEE, 2014.

[29] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in smt processors. *ACM Sigplan Notices*, 44(3):133–144, 2009.

[30] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 235–246. ACM, 2011.

[31] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. A hierarchical thread scheduler and register file for energy-efficient throughput processors. *ACM Transactions on Computer Systems (TOCS)*, 30(2):8, 2012.

[32] M. Gebhart, S. W. Keckler, and W. J. Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 465–476. ACM, 2011.

[33] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 96–106. IEEE Computer Society, 2012.

[34] M. Gebhart, B. A. Maher, K. E. Coons, J. Diamond, P. Gratz, M. Marino, N. Ranganathan, B. Robatmili, A. Smith, J. Burrill, S. W. Keckler, D. Burger, and K. S. McKinley. An evaluation of the trips computer system. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 1–12, 2009.

[35] S. Z. Gilani, N. S. Kim, and M. J. Schulte. Power-efficient computing for compute-intensive gpgpu applications. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 330–341. IEEE, 2013.

[36] N. Goswami, B. Cao, and T. Li. Power-performance co-optimization of throughput core architecture using resistive memory. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 342–353. IEEE, 2013.

[37] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[38] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, 2012.

[39] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[40] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, volume 9, 2009.

[41] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 343–355. IEEE, 2007.

[42] J. Hansbrough. Fuzzing pci express: security in plaintext. https://cloudplatform.googleblog.com/2017/02/fuzzing-PCI-Express-security-in-plaintext.html. Accessed: May 2017.

[43] M. Harris. An efficient matrix transpose in cuda c/c++. http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc. Accessed: April 2015.

[44] M. Harris. Inside pascal: Nvidias newest computing platform. https://devblogs.nvidia.com/parallelforall/inside-pascal. Accessed: May 2017.

[45] A. Herrera. Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation. *Tech. Rep.*, 2014.

[46] J. Hestness, S. Keckler, and D. Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 150–160, Oct 2014.

[47] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. OConnor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offlloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, pages 204–216, 2016.

[48] Intel. Intel xeon processor e7-8800 series. https://www.intel.com/content/dam/support/us/en/documents/processors/xeon/sb/xeon_E7-8800.pdf. Accessed: March 2018.

[49] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy. Warp-aware trace scheduling for gpus. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 163–174. ACM, 2014.

[50] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram. Gpu register file virtualization. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 420–432. ACM, 2015.

[51] W. Jia, K. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 272–283, Feb 2014.

[52] W. Jia, K. A. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 272–283. IEEE, 2014.

[53] N. Jing, S. Chen, S. Jiang, L. Jiang, C. Li, and X. Liang. Bank stealing for conflict mitigation in gpgpu register file. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pages 55–60. IEEE, 2015.

[54] N. Jing, H. Liu, Y. Lu, and X. Liang. Compiler assisted dynamic register file in gpgpu. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, pages 3–8. IEEE Press, 2013.

[55] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang. An energy-efficient and scalable edram-based register file architecture for gpgpu. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 344–355. ACM, 2013.

[56] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, and X. Liang. Cache-emulated register file: An integrated on-chip memory architecture for high performance gpgpus. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[57] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *Proceedings of workshop on general purpose processing using GPUs*, page 1. ACM, 2014.

[58] A. Jog, O. Kayiran, N. Chidambaram 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 *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406, New York, NY, USA, 2013. ACM.

[59] A. Jog, O. Kayiran, N. Chidambaram 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 *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[60] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 332–343. ACM, 2013.

[61] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Exploiting core-criticality for enhanced gpu performance. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 351–363. ACM, 2016.

[62] T. Juan, J. J. Navarro, and O. Temam. Data caches for superscalar processors. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 60–67, New York, NY, USA, 1997. ACM.

[63] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[64] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 157–166. IEEE Press, 2013.

[65] O. Kayıran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das. μc-states: Fine-grained gpu datapath power management. In *2016 International Conference on Parallel Architecture and Compilation (PACT)*, 2016.

[66] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, (5):7–17, 2011.

[67] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten. Microarchitectural mechanisms to exploit value structure in simt architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 130–141. ACM, 2013.

[68] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE micro*, 25(2):21–29, 2005.

[69] H. Lee, K. Brown, A. Sujeeth, T. Rompf, and K. Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 63–74, Dec 2014.

[70] H. Lee, A. Faruque, and M. Abdullah. Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 220. European Design and Automation Association, 2014.

[71] J. W. Lee and K. Asanovic. Meterg: Measurement-based end-to-end performance estimation technique in qos-capable multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 135–147. IEEE, 2006.

[72] K. Lee. Introducing big basin: Our next-generation ai hardware. https://code.facebook.com/posts/1835166200089399/introducing-big-basin-our-next-generation-ai-hardware. Accessed: May 2017.

[73] 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 *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271, Feb 2014.

[74] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram. Warped-compression: enabling power efficient gpus through register compression. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 502–514. ACM, 2015.

[75] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[76] S.-Y. Lee, A. Arunkumar, and C.-J. Wu. Cawa: coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 515–527. ACM, 2015.

[77] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, 41(3):487–498, 2013.

[78] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

[79] J. Leng, Y. Zu, and V. J. Reddi. Gpu voltage noise: Characterization and hierarchical smoothing of spatial and temporal voltage noise interference in gpu architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 161–173. IEEE, 2015.

[80] J. Leng, Y. Zu, M. Rhu, M. Gupta, and V. J. Reddi. Gpuvolt: Modeling and characterizing voltage noise in gpu architectures. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 141–146. ACM, 2014.

[81] B. Li, L.-S. Peh, L. Zhao, and R. Iyer. Dynamic qos management for chip multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):17, 2012.

[82] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into gpu on-chip memory resources. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 23–33. IEEE, 2015.

[83] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based cache allocation in throughput processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 89–100. IEEE, 2015.

[84] Z. Li, J. Tan, and X. Fu. Hybrid cmos-tfet based register files for energy-efficient gpgpus. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 112–119. IEEE, 2013.

[85] X. Liang and D. Brooks. Mitigating the impact of process variations on processor register files and execution units. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 504–514. IEEE Computer Society, 2006.

[86] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu. Barrier-aware warp scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*, page 42. ACM, 2016.

[87] Z. Liu, S. Gilani, M. Annavaram, and N. S. Kim. G-scalar: Cost-effective generalized scalar execution architecture for power-efficient gpus. In *IEEE Int. Symp. on High-Performance Computer Architecture (HPCA)*, 2017.

[88] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[89] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li. Exploration of gpgpu register file architecture using domain-wall-shift-write based racetrack memory. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[90] S. Melvin and Y. Patt. Enhancing instruction scheduling with a block-structured isa. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[91] T. Mostak. An overview of mapd (massively parallel database). *White paper. Massachusetts Institute of Technology*, 2013.

[92] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta. Argo: aging-aware gpgpu register file allocation. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 30. IEEE Press, 2013.

[93] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.

[94] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 57–68. ACM, 2007.

[95] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE micro*, 28(3), 2008.

[96] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48, Feb 2014.

[97] Nvidia. GeForce GTX 480. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications.

[98] Nvidia. GeForce GTX 680. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[99] NVIDIA. Gpu computing sdk.

[100] Nvidia. Nvidia cuda programming guide. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf. Accessed: April 2015.

[101] Nvidia. Nvidia volta. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture. Accessed: May 2017.

[102] NVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110.

[103] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt. How to fake 1000 registers. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–18. IEEE Computer Society, 2005.

[104] K. Olukotun, M. Rosenblum, and K. Wilson. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 147–147, May 1996.

[105] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418. ACM, 2013.

[106] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[107] 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 the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 18. ACM, 2015.

[108] J. J. K. Park, Y. Park, and S. Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 527–540. ACM, 2017.

[109] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. A case for toggle-aware compression for gpu systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–200. IEEE, 2016.

[110] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate compression: practical data compression for on-chip caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 377–388. ACM, 2012.

[111] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on gpus: designing memory management units for cpu/gpus with unified address spaces. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 743–758. ACM, 2014.

[112] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose. Reducing datapath energy through the isolation of short-lived operands. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 258–268. IEEE, 2003.

[113] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit to a superscalar processor. In *Proceedings of the 13th international conference on Supercomputing*, pages 1–10. ACM, 1999.

[114] M. Rhu and M. Erez. Maximizing simd resource utilization in gpgpus with simd lane permutation. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 356–367. ACM, 2013.

[115] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–56. IEEE, 1997.

[116] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler. A variable warp size architecture. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 489–501. ACM, 2015.

[117] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[118] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE.

[119] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 99–110, New York, NY, USA, 2013. ACM.

[120] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[121] A. Roth. Physical register reference counting. *IEEE Computer Architecture Letters*, 7(1):9–12, 2008.

[122] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 422–433. IEEE, 2003.

[123] A. Sethia, D. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 174–185, Feb 2015.

[124] A. Sethia and S. Mahlke. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 647–658, Dec 2014.

[125] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 13–24. ACM, 2011.

[126] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler. Flexible software profiling of gpu architectures. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 185–197. ACM, 2015.

[127] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[128] J. A. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign.

[129] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[130] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 62–75. ACM, 2015.

[131] J. Tan and X. Fu. Mitigating the susceptibility of gpgpus register file to process variations. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 969–978. IEEE, 2015.

[132] J. Tan, Z. Li, and X. Fu. Soft-error reliability and power co-optimization for gpgpus register file using resistive memory. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 369–374. IEEE, 2015.

[133] J. Tan, S. L. Song, K. Yan, X. Fu, A. Marquez, and D. Kerbyson. Combating the reliability challenge of gpu register file at low supply voltage. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 3–15. ACM, 2016.

[134] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 193–204. IEEE, 2014.

[135] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi. Cacti 5.0. *HP Laboratories, Technical Report*, 2007.

[136] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *Proceedings of the 49th annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[137] N. Vijaykumar, K. Hsieh, G. PekhimenW, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–14. IEEE, 2016.

[138] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu. A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 41–53. ACM, 2015.

[139] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. Efficient and fair multi-programming in gpus via effective bandwidth management. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[140] S. Wang, Y. Liang, C. Zhang, X. Xie, G. Sun, Y. Liu, Y. Wang, and X. Li. Performance-centric register file design for gpus using racetrack memory. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 25–30. IEEE, 2016.

[141] 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 *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 358–369. IEEE, 2016.

[142] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Quality of service support for fine-grained sharing on gpus. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 269–281. ACM, 2017.

[143] 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 *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.

[144] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan. Enabling coordinated register allocation and thread-level parallelism optimization for gpus. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 395–406. ACM, 2015.

[145] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 76–86. IEEE, 2010.

[146] 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. *ACM SIGARCH Computer Architecture News*, 44(3):230–242, 2016.

[147] J. Yan and W. Zhang. Exploiting virtual registers to reduce pressure on real registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):3, 2008.

[148] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[149] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh. Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 247–258. ACM, 2011.

[150] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 406–418. IEEE Computer Society, 2014.

[151] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive cache and concurrency allocation on gpgpus. *Computer Architecture Letters*, PP(99):1–1, 2014.