Synchronization and Coordination in Heterogeneous Processors

by

Joel Hestness

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: November 14th, 2016

The dissertation is approved by the following members of the Final Oral Committee:

Mark D. Hill, Professor, Computer Science
Stephen W. Keckler (advisor), Adjunct Professor, Computer Science
Mikko H. Lipasti, Associate Professor, Electrical and Computer Engineering
Michael M. Swift, Associate Professor, Computer Science
David A. Wood (advisor), Professor, Computer Science

© Copyright Joel Hestness 2016

All Rights Reserved

ACKNOWLEDGMENTS

I journeyed a long and eventful path while working through the research contained herein. On this path, many people offered me their support, and I deeply appreciate it.

First and foremost, I thank my advisors Steve Keckler and David Wood for their advice, guidance, and mentorship during this journey. I started research with Steve more than 9 years ago at the University of Texas at Austin. Steve coached me to develop strong research and writing fundamentals early in my journey. I am very thankful for his continued support, especially after I transferred to the University of Wisconsin, because it gave me the confidence to finish the Ph.D. I am also very thankful to David, who ensured that it would be practical to move and continue my research at UW-Madison. Prior to joining UW-Madison, I had explored many research directions and identified a few potential novel ideas. It was David's keen eye for when to dig in and implement new ideas that really pushed me to develop the techniques here.

Along with my advisors, I thank my preliminary exam and dissertation committee members. Mark Hill has imparted many nuggets of wisdom throughout my research. Nam Sung Kim offered insightful feedback during my preliminary exam and ideas for how to best leverage our simulation infrastructure. I really appreciated Mikko Lipasti's Advanced Computer Architecture course. The course spent significant time on memory consistency models, and the deep treatment was very helpful as I developed simulation infrastructure for this thesis. I thank Mike Swift for his energy around research and a couple brainstorming sessions after some of my preliminary idea testing.

While at UW-Madison, I engaged with numerous faculty and students who each contributed to the unparalleled and rich computer architecture educational experience. I thank Prof. Guri Sohi and Prof. Karu Sankaralingam for welcoming me back to UW-Madison and for a few deep conversations about more abstract technology commercialization and business-related topics. I also appreciate the assistance from my many colleagues, from conversations about research, prior work, and reading group to practice talk feedback. I

thank Muhammad Shoaib bin Altaf, Newsha Ardalani, Raghuraman Balasubramanian, Arkaprava Basu, Chris Feilbach, Jayneel Gandhi, Vinay Gangadhar, Dibakar Gope, Gagan Gupta, Swapnil Haria, Derek Hower, Mona Jalal, Marc de Kruijf, Jason Lowe-Power, Jaikrishnan Menon, Tony Nowatzki, Lena Olson, Marc Orr, Sankaralingam Panneerselvam, Somayeh Sardashti, Rathijit Sen, Srinath Sridharan, Vijay Thiruvengadam, Nilay Vaish, Aditya Venkataraman, Hao Wang, and Hongil Yoon.

I would also like to acknowledge the valuable support and collaborations from the broader computer architecture research community. I thank Jason Power and Marc Orr; during our development of the gem5-gpu simulator, they provided very valuable tactical and philosophical discussions about how we should architect the simulator. As a result of our efforts, gem5-gpu is one of the most broadly applicable and widely-used heterogeneous processor simulators available. I also thank gem5 collaborators, Bradford Beckmann, Nathan Binkert, Steve Reinhardt, and Ali Saidi who mentored me in my early stages of developing open-source simulation tools and infrastructure.

I spent the first half of my Ph.D. at the University of Texas at Austin, and I thank those that helped me during that time. Prof. Doug Burger's course on Historical Readings in Computer Architecture cemented my interest in the field. I also thank the colleagues who contributed to my early graduate school learnings: Renée St. Amant, Katherine Coons, Hadi Esmaeilzadeh, Ehsan Fatehi, Mark Gebhart, Paul Gratz, Boris Grot, Ivan Jibaja, and Bertrand Maher.

Outside my Ph.D. and technical pursuits, I learned deeply about the startup world through 3 Day Startup (3DS), which provided immensely valuable broader context for my education. My 3DS family was integral to my Ph.D. journey. I have treasured my weekly meetings with Cameron Houser. His collaboration has grown my own marketing and operational skills. I thank Ruchit Shah for our deep philosophical conversations and his consistent and refreshing enthusiasm for life. I offer my deepest appreciation to Thomas Finsterbusch, who has been an exceptional co-founder and colleague. His collaboration

and influence has helped me refine my skills from defining high-level vision all the way down to implementing practical tactics.

Many other 3DS family members have also supported my growth. I specifically thank my other original 3DS co-founders, Vikram Devarajan and Jeremy Guillory, for their support and enthusiasm for entrepreneurship. Our shared experience working through graduate school, theses, and startups carried me through the more challenging stages of this journey. I also thank Isaac Barchas, Bart Bohn, Pat Condon, and Jason Seats for their insights and wisdom, which have not only cultivated 3DS' growth, but also my own confidence and leadership skill. I also need to thank the countless other 3DS family members who have collaborated, organized, and supported the cause. You have made 3DS a rich educational experience for many.

I sincerely thank my close friends, Ryan and Ashley Crisp, Benjamin and Shannon Delaware, Lewis Fishgold, Thomas Finsterbusch, Andrew Matsuoka, and Donald Nguyen, who contributed to a well-rounded graduate school experience. I reflect warmly on our disc golf and grilling outings, which made life very enjoyable in the early and heavy stages of graduate school.

Last but not least, I am eternally grateful to my family for their tremendous support and unconditional love. My parents, Tom and Lynn, have always offered encouragement, patience, and faith that I would complete the Ph.D. They have made sure that I could continue the journey during challenging times. I also thank my sister, Jodi, for her consistent energy and excitement about my progress. Finally, I thank by best friend, Ginger Pocock, for supporting me through thick and thin. Thank you for keeping me company during those long, weekend afternoons working at the coffee shop, and for leading the way through the Ph.D. I could not have done it without you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Recent developments in internet connectivity and mobile devices have spurred massive data growth. Users demand rapid data processing from both large-scale systems and energy-constrained personal devices. Concurrently with this data growth, transistor scaling trends have slowed, diminishing processor performance and energy improvements compared to prior generations. To sustain performance trends while staying within energy budgets, emerging systems are integrating many processing cores and adding accelerators for specialized computation. The graphics processor (GPU) has become a prominent accelerator by offering fast, energy-efficient processing for data-parallel applications.

For systems like desktops and mobile devices, heterogeneous processors have integrated GPUs onto the same chips as general-purpose cores (CPUs). These integrated processors have introduced new programmability and performance challenges. Similar to systems containing discrete GPU cards, early heterogeneous processors divide the CPU and GPU memory spaces, requiring programmers to explicitly manage data movement between the core types. To simplify this programming challenge, emerging heterogeneous processors provide shared memory and cache coherence between CPU and GPU cores. Still, few applications have been developed to use these new capabilities, and it is challenging to predict how programmers might use them.

We aim to enable programmers to write applications that deftly and efficiently coordinate computation across CPU and GPU cores in shared memory, cache coherent heterogeneous processors. First, we analyze existing GPU computing applications to identify characteristics that can benefit from the new compute and communication capabilities. Generally, application phases with high data-level parallelism (DLP) are a good fit for GPUs, while phases with low DLP are a good fit for CPU cores, especially if they contain high instruction-level parallelism. Further, many GPU computing applications involve multiple software pipeline stages with these varied compute and memory demands.

This thesis proposes and evaluates techniques that improve the performance, pro-

grammability, and energy efficiency of synchronization and coordinated computation in GPUs and heterogeneous processors. Guided by our broad workload analysis, we develop applications that execute concurrently on both CPU and GPU cores. We identify two processor design challenges that limit the performance and energy efficiency of these coordinated work applications.

First, GPU barrier synchronization provides a useful mechanism for programmers to reduce synchronization granularity from numerous GPU threads. However, GPU barriers can still cause high overhead; threads must wait at barriers for lagging threads. Further, to communicate from GPU to CPU cores requires longer latency memory ordering guarantees than commonly used in discrete GPU applications. To reduce barrier overhead, we propose a hardware technique, the Transparent Fuzzy Barrier, which dynamically finds and executes instructions while threads wait at barriers to reduce barrier overhead by more than 50%.

Second, GPU applications often use very coarse-grained producer-consumer communication, which reduces performance due to excessive cache spills and energy-expensive off-chip memory accesses. Existing software transformations can reduce cache spilling, but they tend to be complicated, requiring the programmer to reason about producer-to-consumer mappings and cached data footprint. To ease the cache management burden, we propose a novel hardware technique, called Q-cache, to support concurrent producer-consumer activity. Q-cache measures the cached data footprint and throttles producer or consumer tasks when buffered data might start spilling from cache. Q-cache eliminates cache spills and reduces contention to significantly improve application performance and energy.

1 INTRODUCTION

In the recent past, two major computing trends have emerged that challenge future system designs. First, the rise of internet and mobile connectivity has spurred unprecedented data growth and demand for computing resources that can process large data sets. This growth challenges large-scale systems in their ability to quickly analyze data and extract meaning, and it challenges desktops and mobile systems to leverage data quickly to interactively supply knowledge and entertainment to users. Data-oriented workloads are increasingly intricate, aiming to leverage more data and communication.

Second, recent transistor technology nodes have run into device physics limitations that have slowed Denard scaling and Moore's law, making it difficult to scale transistor sizes down without increasing power density. Processor designers face more challenges to improve the performance of single processing cores without also increasing chip power and energy. To sustain overall performance trends of prior eras while staying within system energy budgets, processor designs are moving toward multiple cores and various accelerators. Large-scale computing systems continually increase the number of components for total compute capacity, while desktop and mobile systems provide increasing compute efficiency by integrating more diverse, targeted cores.

With the confluence of these trends, a prominent accelerator has become ubiquitous over the last few years: the graphics processing unit (GPU). Compared to central processing units (CPUs) designed for general-purpose computation, GPUs offer wide multithreading and in-order, vector execution pipelines that can process wide data-level parallelism (DLP) with better energy efficiency. Depicted in Figure 1.1, systems have commonly integrated GPUs as a separate card connected via a relatively low-bandwidth PCI-express bus.

The architecture of discrete GPUs—like other discrete accelerators—adds significant programming complexity to manage the separate compute cores and memories. With split CPU and GPU memories, programmers must explicitly copy data between CPU and

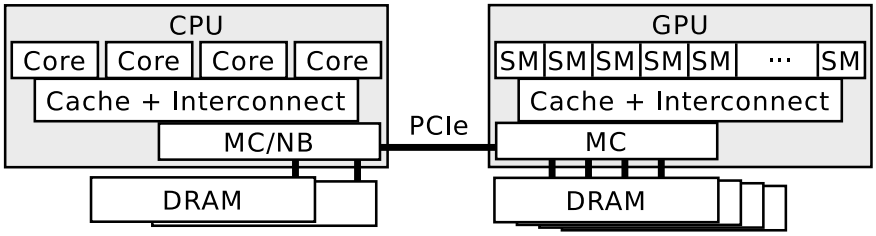


Figure 1.1: Discrete GPU system with separate CPU and GPU chips.

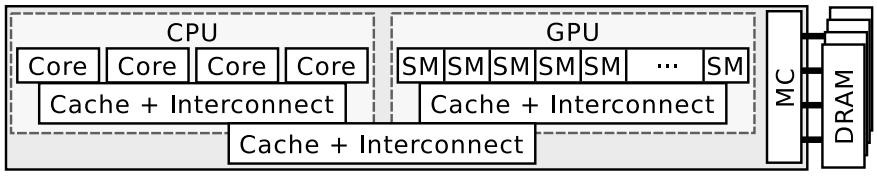


Figure 1.2: Heterogeneous processor with integrated GPU on a single chip.

GPU memory spaces and assign computation to the appropriate cores. As a result, GPU applications must often use simple and regular data structures that can limit algorithmic expressiveness, and programmers must carefully manage data movement to achieve good performance.

Two major advances aim to reduce the programming complexity and overheads of discrete GPUs: unified virtual memories and the development of heterogeneous processors. First, in recent discrete GPU systems, software runtimes have merged CPU and GPU virtual memory spaces to provide a unified view of shared memory [59, 106]. The runtime can manage data movement between CPU and GPU memory spaces, so programmers can use more intricate data structures. However, programmers must still be careful which memories store data to avoid heavy runtime data movement and to optimize performance.

To push these capabilities further, companies have started designing heterogeneous processors [57, 58, 105], which integrate CPUs and GPUs onto a single chip, as diagrammed in Figure 1.2. CPU and GPU cores can share data directly by accessing common physical memory and advanced designs even share on-chip caches. These heterogeneous processors introduce new challenges, such as resolving memory consistency models between the core types [53, 56, 133] and providing cache coherence [134, 119, 75]. These advances allow

CPU and GPU cores to communicate using fine-grained memory accesses. Prior to our work, however, no publicly-available workloads had been developed to leverage these new capabilities.

This thesis offers three major contributions to advance the performance, programmability, and energy efficiency of GPUs and heterogeneous processors. First, we analyze existing GPU computing applications to identify those that have potential to benefit from the new heterogeneous processor compute and communication capabilities, and we develop applications that coordinate CPU and GPU computation. Second, we propose a technique, called Transparent Fuzzy Barriers, that improves the performance of GPU hardware barriers for more efficient synchronization. Third, we propose a hardware technique, called Q-cache, which supports producer-consumer communication in caches to improve performance, programmability, and energy efficiency.

1.1 Characterizing GPU Computing Workloads

Emerging heterogeneous CPU-GPU processors offer shared memory and cache coherence capabilities. A major limitation when investigating them is that the research community lacks applications that concurrently utilize CPU and GPU cores and use the new processor capabilities. CPU and GPU cores provide different performance and power characteristics, so programmers need to know what portions of an application might be most appropriate to run on either core type in a heterogeneous processor. Unfortunately, prior to our work, no open-source applications had been written for heterogeneous processors. In the first part of this thesis, we analyze existing computing applications to compare CPU and GPU cores and to identify application structures that can benefit from coordination in heterogeneous processors.

Comparing CPU and GPU cores: We begin addressing this challenge in Chapter 2, which identifies application-level and microarchitectural characteristics that indicate CPU or GPU

cores can provide better performance. We compare the core types by running the same workloads on both a multicore CPU processor and a GPU. Broadly, the results indicate that the core types share many application and microarchitecture characteristics.

Despite their similarities, CPU and GPU cores have a couple important differences. Compared to GPU cores, CPU cores must mine instruction-level parallelism (ILP) from a small number of threads and often struggle to expose significant memory-level parallelism (MLP). Though they can extract more ILP than GPUs, CPUs are more sensitive to memory latency, and a large portion of run time is spent stalling on memory accesses. In contrast, GPU cores expose significant MLP via their wide multithreading. They are less sensitive to memory latency, but more sensitive to available cache and memory bandwidth.

This microarchitectural characterization also underscores the importance of efficient GPU bulk memory access synchronization. The wide multithreading in GPUs allows hundreds of threads to communicate results using memory fences and barriers. Relative to other GPU computing workload behaviors, barrier synchronization causes significant performance overhead, slowing down applications by more than 14% on average and up to 65%.

GPU computing software pipelines: When developing applications for heterogeneous processors, programmers have opportunity to use cores and cache that might be underutilized in discrete GPU systems. For instance, many existing GPU computing applications move data and computation to discrete GPUs and let CPU cores sit idle for long periods of run time. In heterogeneous processors, this organization underutilizes CPU cores, which share access to data and could participate in computation. We would like to optimize applications to increase resource utilization, and to improve performance and energy efficiency.

Although the CPU and GPU comparison in Chapter 2 identifies application characteristics that can benefit from either core type, the aggregated statistics do not pinpoint

software-level structures that should be optimized or transformed. To further guide our heterogeneous processor application development, Chapter 3 analyzes the software pipelines of a broad set of GPU computing applications. We assess the overheads of moving data between CPU and GPU memories, the common producer-consumer communication structures, and how parallelism varies across separate pipeline stages. With this data, we analytically model the potential performance benefits of software transformations that can better utilize core, cache, and memory.

Our analysis shows that when modifying applications to target heterogeneous processors, GPU computing applications could experience an average 1.7× performance improvement and up to 20×. In addition to improving performance, the software transformations have potential to reduce application energy by reducing component idle time and excess memory access activity. To benefit from the shared caching, existing applications will need to be transformed to reduce the temporal distance between producer and consumer tasks, and these transformations can better leverage the fine-grained communication available in heterogeneous processors. Finally, hardware could help to detect memory access contention and appropriately modulate cache accesses to increase cache efficiency.

1.2 Developing Heterogeneous Processor Workloads

The GPU computing workload characterization highlights the potential performance benefits of restructuring applications to better utilize heterogeneous processor resources. The shared memory and cache coherence capabilities will permit applications to communicate and synchronize data in new and unique ways. Unfortunately, prior to our studies, the research community lacked both publicly-available applications that use these new capabilities, and the simulation infrastructure to test potential hardware configurations of heterogeneous processors. Chapter 4 discusses the important aspects of our efforts to develop heterogeneous processor workloads and simulation.

While the GPU computing characterization results are promising, in practice, programmers face two major challenges when developing applications to run on heterogeneous processors. First, they must identify portions of an application or algorithm to transform that may result in the largest performance benefits when mapped to heterogeneous processors. In Chapter 4, we identify classes of applications that commonly contain specific software pipeline structures that indicate when optimization will be fruitful.

These software structures of interest for heterogeneous processors share common data-level parallelism (DLP) characteristics. Specifically, applications often contain pipeline stages that reduce high DLP input data to narrower output data, and the next pipeline stage consumes this narrower-DLP output. To improve performance and energy, applications can run high DLP stages on GPU cores to leverage the wide multithreading, and use CPU cores for stages with lower DLP and potential for ILP. If producer-consumer communication permits, applications can even execute these CPU and GPU stages concurrently. We commonly find these varying DLP structures in vector and matrix computations, numerical methods, image processing, and graph analytics applications.

Second, to allow concurrent CPU and GPU computation, programmers must select appropriate synchronization mechanisms to ensure correct and efficient data handling. When GPUs participate in coordinated computation, they can cause heavy cache contention and high synchronization overhead, especially when using fine-grained signaling. We discuss options for synchronizing communication between CPU and GPU cores, and techniques to make synchronization granularity coarser to reduce overheads. Guided by these software structure and synchronization learnings, we develop a set of microbenchmarks and applications—some optimized from existing GPU computing suites—to leverage the compute and communication capabilities in heterogeneous processors.

Finally, for our heterogeneous processor design investigations, we faced challenges developing simulation infrastructure capable of testing these processors. Prior to our work, the research community lacked simulators to test their shared memory and cache coherence

capabilities with reasonable fidelity. To flexibly evaluate processor design decisions, we developed a simulator called gem5-gpu [120]. Chapter 4 describes important simulation capabilities we developed to perform high-fidelity testing, including memory consistency and cache coherence, and cache hierarchy modeling improvements for realistic CPU and GPU memory access interaction.

1.3 Techniques for Efficient GPU Barriers

As discovered in the microarchitecture characterization, GPU workloads experience comparatively high overhead from barrier synchronization—a 14% slowdown on average. For heterogeneous processor applications that coordinate work among core types, programmers may use barriers to reduce the granularity of data synchronized by GPU cores. Further, when communicating between cores in emerging heterogeneous processors, synchronization overhead will likely increase; compared to discrete GPU memory hierarchies, heterogeneous processor memory consistency and cache coherence models add latency for operations that ensure correct memory orderings. These emerging system trends indicate that GPU barriers need to be efficient.

Prior studies have aimed to address barrier overhead in both multithreaded CPU environments and GPUs. Unfortunately, they do not significantly reduce overhead in modern GPUs. Prior GPU warp scheduling techniques aim to prioritize warps participating in barriers, but these techniques only reduce overhead by 11% average for a 1.01× application-level speedup. Other prior work proposes fuzzy barriers, which allow programmers to specify independent instructions to be executed during barriers, but adapting fuzzy barriers to GPUs only hides 16% of overhead (1.015× speedup). Programmers or compilers would have to wrestle with relaxed memory consistency and control flow analysis to identify more fuzzy instructions.

To address shortcomings of prior work, we propose Transparent Fuzzy Barriers (TFBs),

a hardware technique that dynamically executes instructions during barriers to hide barrier overhead. By implementing fuzzy barriers in hardware, programmers and compilers need not specify work to execute during barriers. On average, TFBs hide more than 50% of barrier overhead for a 1.061× average speedup and up to 1.14×. Further, TFBs can be implemented with small logic changes or added storage.

1.4 Improving Producer-Consumer Communication

Driven from our GPU computing software pipeline characterization in Chapter 3, we develop applications that share memory and coordinate concurrent work among CPU and GPU cores, as described in Chapter 4. We expect that emerging workloads will similarly try to leverage CPU and GPU cores concurrently and communicate data between the two. This application structure presents two primary challenges to optimizing coordinated work: programmability and cache management.

First, programmers want to use simple paradigms for data communication. Historically, they have structured graphics and GPU computing applications as numerous software pipeline stages that implicitly synchronize data on kernel boundaries between stages, a simple and general-purpose approach. More advanced techniques, such as kernel fission and fusion, allow programmers some control over cache usage, but both structures require significant effort to map producer output to specific consumer tasks and to ensure application correctness. Software queues can simplify the programming interface by allowing any consumer to pull results from any producer, though ensuring communication correctness is still more complicated than kernel boundary synchronization.

Second, producer-consumer communication managed by software is often cache inefficient. When using kernel boundary synchronization, producer kernels processing large data structures push their intermediate results out of cache to off-chip memory before they can be consumed by a following kernel. This cache spilling causes an average of 15%

and up to 34% excess off-chip memory accesses. Software transformations—kernel fission, fusion, and software queuing—all require the programmer to accurately estimate and manage the amount of cached data to ensure it does not spill to off-chip memory.

To ease the programmer's burden for managing data in cache, we propose a novel hardware technique, called Q-cache. Q-cache measures the producer-consumer data footprint in caches, and throttles producer or consumer tasks when buffered data might start spilling from cache. By improving cache footprint estimates, Q-cache provides average performance gains of 1.20× over the best software-only algorithm implementations. Further, Q-cache offers the best energy characteristics by reducing cache spills and trimming core activity to match producer and consumer cache access rates.

1.5 Thesis Organization

This thesis is organized into two major parts. The first part focuses on characterization of existing GPU computing workloads and development of new workloads for heterogeneous processors. Chapter 2 performs a comparison of application and microarchitectural effects when running computing applications either on a multicore CPU or a GPU. Chapter 3 analyzes GPU computing software pipelines to find inefficiencies and opportunities when running them in heterogeneous processors. Chapter 4 describes our efforts to develop workloads that coordinate work across both CPU and GPU cores in a heterogeneous processor. It also describes the simulation infrastructure we develop to evaluate these workloads and our proposed hardware techniques.

The second part of the thesis describes our proposed techniques to improve synchronization and coordinated work in heterogeneous processors. Chapter 5 describes and evaluates Transparent Fuzzy Barriers for GPUs, including a comparison against prior works in fuzzy barriers and warp scheduling. Chapter 6 describes and evaluates Q-cache by comparing it against software techniques to manage data in caches. Finally, Chapter 7 summarizes our

contributions and describes potential extensions to the work.

2 COMPARING CPU AND GPU MICROARCHITECTURE EFFECTS

Current systems-on-chip (SoCs) and heterogeneous processors incorporate core types that, in isolation, are well-researched. However, emerging heterogeneous processors allow CPU and GPU cores to share memory and communicate through coherent caches. These advances present new challenges and opportunities when applications try to leverage both cores types. To understand the differences between multicore CPU and GPU microarchitectures, this chapter performs a holistic comparison of the microarchitectures by running a set of applications written to target both core types.

This comparison establishes unique benefits of each core type under different application characteristics. CPU and GPU applications show significant similarity in application-level characteristics and cache reference locality, and the different cores sometimes perform comparably. However, GPUs, with their wide multithreading, often outperform CPUs during wide data-level parallel (DLP) computation by better hiding memory access latency. Their many threads operate independently allowing them to expose more and burstier memory-level parallelism (MLP). Program phases with low DLP and thread-level parallelism (TLP) are likely to be a better fit for CPUs, especially when the can expose significant instruction-level parallelism (ILP).

Given these unique benefits, programmers are faced with a couple of challenges and opportunities. First, since heterogeneous processors offer more flexibility to move computation between cores without needing to migrate data, the programmer can refine decisions about which cores to use at any point during a computation. Second, when CPU and GPU cores can share memories, they can concurrently process the same data. The programmer can use all compute cores, but will need to be mindful of opportunities to share data in caches and challenges of avoiding potential memory system contention.

The findings in this chapter are published in the Proceedings of the International IEEE Symposium on Workload Characterization (IISWC), 2014 [54].

2.1 Introduction

Over the last few years, GPUs have been added to chips that have historically contained CPU cores. These GPUs offer both high-throughput graphics rendering and accelerated computing. Recent product announcements by AMD [57], Intel [58], and NVIDIA [105] also indicate that emerging heterogeneous CPU-GPU processors will provide a unified virtual address space and cache coherence across these cores. These fused architectures avoid many of the overheads associated with discrete GPUs, and can simplify the programming interface to use both core types.

Although heterogeneous processors offer benefits of decreased communication latencies and increased bandwidth between cores, their integrated memory hierarchies may also result in complicated memory access interactions. When located on the same chip, the heterogeneous cores will share system resources such as the on-chip cache hierarchy, interconnect, and the DRAM, potentially causing contention. Thus, it will be important to understand the different memory access characteristics between the core types.

This chapter presents a complete quantitative analysis of the memory access behavior of multithreaded CPU applications compared to their GPU counterparts with the aim of illuminating the application and microarchitectural causes of memory behavior differences, as well as the common effects of these differences. We focus on a memory system characterization to understand how each core type exposes parallel memory accesses, exploits locality, and leverages capabilities of the memory hierarchy.

The most significant observations include:

- Memory access vectorization and coalescing reduce the number of spatially local accesses to cache lines. Both are fundamental means for exposing MLP to lower levels of the memory hierarchy.

- For data-parallel workloads, CPU and GPU cache hierarchies play substantially similar roles in filtering memory accesses to the off-chip interface.

- Due to threading and cache filtering differences, memory access patterns show substantially different time-distributions: CPU applications typically exhibit regularly time-distributed off-chip accesses, while GPU applications frequently produce large access bursts.

- The bursty memory accesses of GPU cores makes them more sensitive to off-chip bandwidth while the MLP from multithreading makes them less sensitive to off-chip access latency than CPU cores.

Overall, CPU cores must extract significant ILP to find parallel memory accesses to send to the memory hierarchy, and L1 caches often filter locality that could expose MLP to lower levels of the memory hierarchy. On the other hand, GPU cores and caches are organized to avoid MLP limitations, allowing the programmer to focus their efforts on leveraging the available MLP.

The rest of this chapter is organized as follows. Section 2.2 describes the characterization methodology. Section 2.3 measures application-level characteristics including algorithm structure, operation counts, and memory footprint. Section 2.4 details our measurements of memory access characteristics, including access counts, locality, and bandwidth demand. Section 2.5 quantifies and describes the performance effects of these different memory access characteristics. We discuss implications and related work in Section 2.6. Section 2.7 concludes.

2.2 Comparison Methodology

To adjust various system parameters and to collect detailed statistics, we simulate and compare the heterogeneous CPU-GPU processors as described in this section. This section also describes the applications and simulation environment used for this comparison.

2.2.1 Simulated Systems

Table 2.1: Heterogeneous CPU-GPU processor parameters.

Core Type	ISA	Key Parameters	GFLOP/s Per Core	Approx. Area (mm^2)
CPU cores	x86	(4×) 5-wide OoO, 4.5GHz, 256 instruction window	22.5	5.3
GPU cores	PTX	(4×) SMs: 8 CTAs, 48 warps of 32 threads, 700MHz	22.4	6.1

Component	Parameters
CPU Caches	Per-core 32kB L1I + 64kB L1D and exclusive private L2 cache with aggregate capacity 1MB
GPU Caches	64kB L1, 48kB scratch per-core. Shared, banked, non-inclusive L2 cache 1MB
Interconnect	Peak 63 GB/s data transfer between any two endpoints
Memory	(2×) GDDR5-like DIMMs per memory channel, 32GB/s peak

Figure 2.1 diagrams the heterogeneous CPU-GPU processor architecture that we simulate in each of our tests. The baseline parameters of this processor are included in Table 2.1. This basic architecture incorporates CPU and GPU cores on a single chip, and the different core types are allowed to communicate through a unified address space shared memory.

Cores: CPU and GPU cores are at the top of the hierarchies, and we list their configurations in Table 2.1. To control for many of the differences between CPU and GPU cores, we model an aggressive CPU core with peak theoretical FLOP rate comparable to the GPU cores for direct comparison of core efficiency. CPU cores operate at a very high frequency, have a deep instruction window, and 5-wide issue width as a way to observe instruction-level parallelism (ILP) limitations in the per-thread instruction stream. This

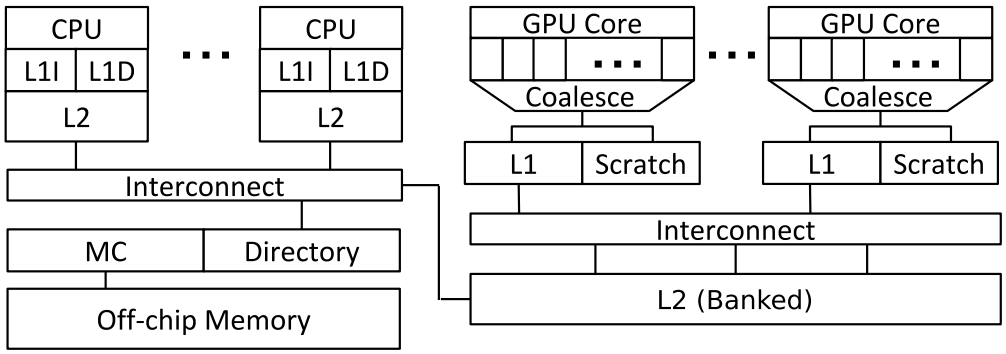


Figure 2.1: Heterogeneous CPU-GPU architecture.

core is a traditional superscalar architecture and contains a 4-wide SIMD functional unit, for which we compiled applications with automatic vectorization.

We compare these CPU cores against GPU cores configured similarly to NVIDIA's Fermi GTX 500 series streaming multiprocessor (SM) with up to 48 warps and a total of 1536 threads per core. With 32 SIMD lanes per core and a frequency lower than the CPU cores, this GPU core is capable of 22.4 GFLOP/s for single-precision computations, consistent with the CPU core.

While core count scalability will be of interest, this study aims to understand the different effects of core microarchitecture on memory system behavior, so these baseline systems model fixed core counts for each type. In particular, we compare 4 CPU cores against 4 GPU cores. For multithreaded CPU applications, we execute a single thread per core, while GPU applications are able to concurrently execute up to the full 1536 threads per core.

Using a modified version of McPAT [83] with CACTI 6.5 [97], we estimate the die area required to implement each core in a 22nm technology. From this data, we observe that the GPU core has a slightly larger size of $6.1mm^2$, which would require it to achieve about 16% higher functional unit occupancy than the CPU core in order to achieve the same compute density.

In addition to the results presented here, we tested varying CPU and GPU core counts and a broader range of cache hierarchy and memory organizations while maintaining similar peak FLOP rate controls. In general, the results from these tests showed expected changes in the magnitudes of memory access demands, but they did not appreciably change the relative or qualitative access characteristics. For presentation and analysis simplicity, we chose to limit our results to the above core configurations.

Cache Hierarchies: CPU and GPU cores pump memory accesses into the top of the cache hierarchy, where each core is connected to L1 caches. The CPU cache hierarchy includes private, 64kB L1 data and 32kB instruction caches. Each core also has a private L2

cache. To maintain a reasonable comparison of the use of caches between core types, we limit the aggregate L2 cache size to 1MB for both core types. Full coherence is maintained over all CPU caches.

The GPU memory hierarchy includes some key differences compared to the CPU hierarchy. First, each GPU core contains a scratch memory that is explicitly accessible by the programmer. Data is often moved into and out of this memory from the global memory space through the caches. Second, GPU memory requests from threads within a single warp are coalesced into accesses that go to either the scratch or cache memories. Each GPU core has an L1 cache, which is allowed to contain incoherent/stale data. If a line is present in an L1 cache when written to, it is invalidated in the L1 before the write is forwarded to the GPU L2. Finally, GPU cores share a unified L2 cache of 1MB. This GPU L2 cache participates in the coherence protocol with the CPU caches.

Interconnects and Memory: We model two simple interconnects for inter-cache communication. The first connects GPU L1 caches and the GPU L2 cache, and the second connects all L2 caches and the directory. These interconnects faithfully model latency, back-pressure and bandwidth limitations consistent with existing GPU interconnects and multicore CPU crossbars, respectively. In all configurations we test, each L2 cache is capable of driving more requested bandwidth than the peak off-chip bandwidth to ensure that achieved bandwidth to any single core can fully utilize off-chip memory resources.

Finally, we model a shared directory controller that manages the coherence protocol and off-chip accesses through the memory controller. To test a range of memory parameters, we use an abstract memory controller design that models first reads, then first-come-first-served memory access scheduling (FR-FCFS) [122], and in appropriate tests, we vary the memory frequency to modulate peak off-chip bandwidth. The memory technology modeled has timing, channel width, and banking parameters similar to GDDR5, but without prefetch buffers, similar to DDR3. Finally, we simulate a range of memory bandwidths in certain tests, but we chose 32GB/s as a baseline as it is representative of bandwidth-to-

compute ratios in currently available systems.

2.2.2 Applications

Selected from the Rodinia suite [24], we use 9 applications to compare CPU and GPU performance and memory behavior. The Rodinia suite includes applications from image processing, scientific workloads, and numerical algorithms, and all applications are designed to exercise heterogeneous computing systems. The suite includes OpenMP multithreaded versions of the applications that we run in multicore CPU systems, and CUDA versions that run their kernels on the GPU. Table 2.2 lists the applications and details about their structure, which we describe in the next section.

In a few cases, we find that the publicly-available versions of Rodinia applications lacked algorithmic mapping and tuning to the target architectures for which they were written. To address this limitation, we refactor and optimize where these transformations are simple to ensure that the behavior we see from the application can be mostly attributed to the underlying core and memory system microarchitecture. Specific optimizations include transposing matrices for reasonable cache strided accesses and GPU coalescing, adding multithreading to unparallelized portions of OpenMP applications, and compiling with optimizations such as loop unrolling and automatic SIMD vectorization.

Further, we choose tuning configurations and application input sets to avoid unfair negative performance impacts for either the CPU or GPU versions. We ensure memory access alignment, and avoid excessive bank contention and unoccupied compute units that can arise from parallel portion tail effects [95]. For most applications, the memory access characteristics change minimally when increasing input set sizes.

Finally, to compare the same portion of each application when run on the CPU or GPU, we annotate each application to collect simulated system statistics over the portion of the application that includes thread or kernel launches and the parallel portion of work executed by these launches. We denote this portion of the application as the region of

interest (ROI). ROIs do not include portions of each application that are common data setup, such as main function variable initialization, or file reads and writes.

2.2.3 gem5-gpu Simulator

To evaluate various heterogeneous CPU-GPU processor designs, we developed the gem5-gpu simulator [120], which integrates the GPU core model from GPGPU-Sim [10] into the gem5 simulator [18]. gem5-gpu uses gem5's Ruby memory hierarchy to model various cache protocols with support for coherence and shared components between the CPU and GPU cores. For our CUDA tests, the GPGPU-Sim cores execute PTX instructions rather than a GPU hardware instruction set, such as SASS. We have validated that using the gem5-gpu memory hierarchy models NVIDIA Fermi hardware more accurately than stand-alone GPGPU-Sim executing PTX.

2.3 Application-level Characteristics

Before delving into memory access behavior, this section briefly discusses the algorithmic structure of the Rodinia applications and their mapping to each core type. In the aggregate, application-level characteristics and statistics for CPU and GPU applications tend to be very similar, but we describe a few notable differences.

2.3.1 Algorithm Structure

The structure of the algorithms that are mapped to each core architecture drive all of the application behavior that we present. Here, we describe a taxonomy of these algorithm structures and how they are mapped to CPU or GPU hardware. Table 2.2 lists the applications considered in our characterization, and their algorithm classifications in this taxonomy.

Table 2.2: Rodinia application algorithm-level characteristics.

Application	Algorithm Structure CPU / GPU	Heap Reads	Iteration Size CPU / GPU	Add'l Structure CPU / GPU
backprop	Pipeline	O(c)	Constant	Reduction
bfs	Iterative	O(c)	Variable	—
heartwall	Iterative	O(I)	Constant	—
hotspot	Iterative	O(I)	Constant	— / Pyramid
kmeans	Iterative	O(I)	Constant	Reduction
nw	Incremental / Iterative	O(c)	Constant / Variable	—
pathfind	Iterative	O(c)	Constant	Pyramid
srad	Pipeline	O(c)	Constant	—
strmclstr	Iterative	O(I)	Constant	Reduction

Table 2.3: Rodinia application-level run time characteristics, single-threaded CPU.

Application	Compute Ops (M)	Memory Ops (M)	Heap Size (MB)
backprop	155	34	35.0
bfs	395	39	6.7
heartwall	12,750	1,924	41.4
hotspot	1,474	242	7.0
kmeans	2,485	571	7.5
nw	303	50	81.0
pathfind	305	76	36.6
srad	780	171	96.0
strmclstr	1,074	248	2.8

First, we describe two classes of applications that employ iterative data processing. The first class of iterative algorithms are listed as "Iterative" and they stream heap data a number of times proportional to I, the number of iterations (O(I)). This structure is generally needed for applications in which either data is transformed over progressive time steps, such as `heartwall` and `hotspot`, or separate iterations successively refine a solution in search of an optimum, as in `kmeans` and `strmclstr`. These algorithms typically spin off parallel computation once or a constant number of times per iteration.

A second class of iterative algorithms divides heap data into chunks, either based on the number of parallel compute units or based on the portions of data that can be efficiently processed per iteration. This iteration structure can be employed when the data for each computation is local to small portions of the overall heap. Wider data dependencies make it difficult to chunk the data to be efficiently processed across separate iterations. These algorithms access heap data roughly a constant number of times, so we denote these

applications as having constant-order ($O(c)$) heap reads.

The structure of work per iteration in iterative algorithms can be variable or fixed. If data dependencies might change across iterations, the amount of available parallel work can also change. We list these algorithms as having "Variable" iteration size, while algorithms that have unchanging parallel work over iterations are denoted as "Constant". It is typically difficult to parallelize varying-work iterative algorithms to use data-parallel microarchitecture constructs such as vectorization and coalescing, because varying or unpredictable data-level parallelism might not fit their widths.

We distinguish one other iterative-like algorithm structure, "Incremental", from iterative algorithms due to the differences in synchronization. The two classes of iterative algorithms described above commonly employ barrier-like synchronization of worker threads between iteration epochs, since the data touched by each thread can typically be isolated from the data of other threads during parallel regions. In contrast, our modified OpenMP version of `nw` employs fine-grained locking to communicate small chunks of data between threads. Compared to the GPU version, this structure elides the barrier synchronization to reduce thread launch overheads and the locking reduces per-thread working set size to reduce cache contention and load imbalance.

The final class of algorithms lacks any iterative nature, and instead, can be parallelized with one or a constant number of parallel work phases. When results from one parallel phase are consumed by a subsequent phase, these applications are referred to as "Pipeline" parallel. `Backprop` and `srad` are both pipeline parallel and access heap data a constant number of times, typically proportional to the number of pipeline stages. Note that both of these applications employ barrier-like synchronization between pipeline stages, but other pipeline-parallel applications could use finer-grained synchronization within pipeline stages.

We note two additional algorithm structures that are employed during or between parallel work epochs: data pyramiding and reduction operations. These are common

structures and tend to have substantial effects on memory access behavior. Data pyramiding is a common technique employed in image processing (iterative) algorithms that access a small neighborhood of data for each computation [139]. By gathering a slightly larger neighborhood of data, a parallel thread is able to compute the result of multiple outer loop iterations, while avoiding the need to stream that data neighborhood multiple times. The "pyramid" term refers to the way that the data neighborhood grows as the number of merged outer loop iterations grows.

Reduction operations are employed in cases where a large set of data, typically produced by multiple threads, must be inspected to find one or a small number of results. Common cases include reduction sums and searches for extrema. These operations often come with extra computation and memory overhead to store intermediate values as the number of parallel threads is increased. In general, as more threads participate in a reduction operation, there is more overhead to synchronizing the handling of intermediately reduced data. While many efficient constructs do exist, we will see that reduction operations are tricky to coordinate between CPU and GPU cores, and can lead to large data communication and run time overheads.

2.3.2 Dynamic Run Time Characteristics

Table 2.3 includes region of interest dynamic run time statistics for each application executed with a single thread on a CPU core. These statistics include the count of dynamic compute and memory operations, and the size of the heap data accessed during the region of interest. In general, these statistics are often similar across core types, so we only briefly touch on the similarities and pay more attention to specific cases that cause the stats to vary across the core types.

For CPU compute operations, we count integer and floating-point micro-instructions that occupy an execution unit rather than the x86 macro-instructions they comprise, and we compare these counts to PTX instructions executed by GPU cores. Memory operations

in x86 count micro-instructions that involve a cache request, and we compare these counts to PTX memory instructions. PTX does not provide a memory-indirect addressing mode, so the count of memory instructions is equal to the number of cache requests. GPU request coalescing reduces the actual number of GPU cache accesses, as described in Section 2.4.

The applications, `nw`, `srad`, and `strmclstr` show compute op count differences of at most 10% across the core types. The applications, `heartwall`, `kmeans`, `nw`, and `pathfind` show memory op count differences of at most 15%. In the geometric mean across all applications, we find that the number of compute and memory ops varies by at most 38% and memory footprint varies by less than 6% across the system configurations. The large geometric mean differences for compute and memory ops indicates that on a per-application basis, large differences can arise. Here, we describe that these differences are due to the use of registers, number of threads, and coordinating work between core types.

A fairly common factor in compute and memory op count differences between system configurations is due to register handling. For x86 CPU applications, the small architected register set (16) can cause register spilling to the stack and recomputation of previously computed values. In contrast, GPU cores have some flexibility in register use due to their core multithreading. By running fewer GPU threads per core and late binding register specifiers to physical registers, there is more flexibility for each thread to access more registers, which can avoid spilling and recomputation.

The CPU versions of `bfs`, `kmeans`, and `strmclstr` require numerous calls to small functions or pointer-chasing in their inner loops, which results in an elevated number of address calculations, memory requests, and register spills. This causes up to 1.33× more cache requests compared to their GPU counterparts. Also, in the CPU version of `heartwall`, each thread executes a large, flat function with numerous local variables. The number of local variables exceeds the number of x86 architected registers, causing CPU threads to excessively recompute values and resulting in almost twice as many compute ops. Both of these differences result in a performance disadvantage for CPU cores.

A less common factor in compute and memory op count differences is the width of multithreading employed by an application. Specifically, in cases such as `backprop`, where each thread must incur some fixed number of ops for setup before performing a share of computation, linearly increasing the number of threads also linearly increases the total number of these fixed ops that must be performed.

For most Rodinia applications, care was taken to ensure that thread counts are kept small to avoid these fixed ops, and the extra threading op counts are often within 8% across the different platforms. However, compared to the single-threaded CPU version, `backprop` employs numerous GPU threads, which increases op counts by approximately 1.5×. On the other hand, `strmclstr`'s numerous CPU thread launches cause approximately 1.7× extra ops. Often, these extra ops can be hidden off the application critical path, because they are executed in parallel by execution units that may not have been fully utilized with fewer threads.

Finally, for applications that coordinate reduction operations between CPU and GPU cores, there is elevated communication and synchronization overhead compared to running the application only on CPU cores. In general, the late stages of reduction operations tend to perform poorly on GPUs, because the limited data-level parallelism causes poor occupancy of the GPU's numerous thread contexts. This limitation suggests that the late stages of the reductions should be performed by a small number of threads, perhaps on CPU cores. However, by transferring control to CPU cores, the GPU versions of `backprop`, `kmeans`, and `strmclstr` incur up to 1.5× as many reduction memory ops to move the intermediately-reduced data from the GPU to the CPU. Reduction performance ends up being a primary factor in performance for these applications.

2.4 Memory Access Characteristics

In this section, we characterize the memory behavior effects of the microarchitectural differences between CPU and GPU cores and cache hierarchies. While we noted in the last section that CPU and GPU applications show many similarities, the different core types expose and leverage MLP in very different ways; CPU cores use a small set of deep per-thread instruction windows, and high-frequency pipelines and caches to expose parallel memory accesses. In contrast, GPU cores expose parallel memory accesses by executing 100s–1000s more threads at lower frequencies, and threads are grouped for smaller per-thread instruction windows and memory request coalescing.

The results here reveal the primary differences in CPU and GPU core memory access. Specifically, while CPU cores rely heavily on L1 caches to capture locality, GPU cores capture most locality with coalescing and lessen the L1 cache responsibilities by providing scratch memory. Beyond the L1 caches, the memory systems tend to capture very similar locality. Further, we see that different core threading and cache filtering result in extreme differences in instantaneous memory access rates; CPU caches tend to filter accesses down to regular intervals, while GPU cores tend to issue bursts of accesses.

2.4.1 Access Counts

Despite large CPU and GPU core threading differences, two core-microarchitecture design characteristics account for the majority of the difference between CPU and GPU cache hierarchy access counts and sizes: scratch memory and address coalescing. We describe their effects here.

Figure 2.2 plots a breakdown of cache access counts for the CPU version (left group of bars) and the GPU version (right group of bars), normalized to a single-threaded execution of each application on a CPU core. We see that the multithread CPU version has small additional memory accesses compared to a single core CPU due to extra threads. Next,

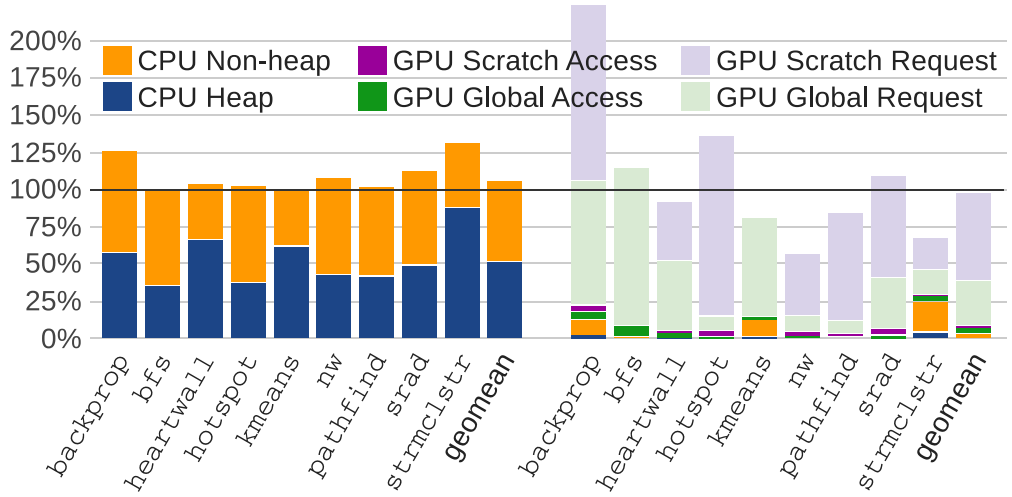


Figure 2.2: Breakdown of memory requests and accesses normalized to single thread CPU execution.

for the GPU, the plot separates memory requests (the absolute number of load/store instructions on the GPU) from cache hierarchy accesses, each of which may be a coalesced set of memory requests. The ghosted portions of the GPU bars represent the number of requests saved by coalescing down to a small number of memory accesses.

Scratch memory: GPU cores provide scratch memory, which can function as local storage for groups of threads to expand the space of local storage with register-like accessibility. In CUDA applications that use the GPU scratch memory, kernels are typically organized into three stages: (1) read a small portion of data from global memory into the scratch memory, (2) compute on the data in scratch memory, and (3) write results back to global memory. Since numerous threads are executing these same stages—often in lock-step—stages 1 and 3 appear as stream operations to the memory hierarchy, and we describe this behavior more deeply below. Besides register files, CPU cores do not have an analogous scratch-like memory, so instead, they typically spill local variables to the L1 caches.

In the common case, CPU stack accesses and GPU scratch memory accesses account for similar portions of memory requests. These similarities are reflected in the proportion of heap versus non-heap (stack or scratch) memory requests depicted in Figure 2.2. Most GPU

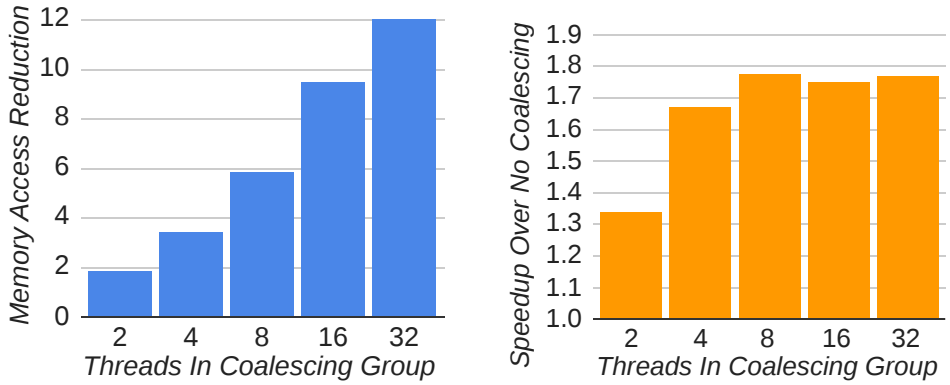


Figure 2.3: Coalescing: memory access reductions and resulting ROI speedup, normalized to no coalescing.

applications employ scratch memory for local data handling, and these requests account for 25–80% of all requests. Moving these requests to scratch memory instead of caches, as on the CPU, reduces the number of requests to the cache hierarchy by a geometric mean of 3.9× before coalescing. Similarly on the CPU, local memory requests account for upwards of 50% of total requests. We also note that CPU stack accesses generally hit in a small number of cache lines in the L1 caches. The high rate of access to CPU stack memory and GPU scratch memory suggests that even a small scratch memory capacity can be useful to mitigate cache hierarchy memory accesses.

A couple applications, `bfs` and `kmeans`, do not use scratch memory and opt instead to access all data from the global memory space through the caches. In both cases, global data and access patterns are such that it would be complex to use scratch memory. We will see later that both of these applications are more sensitive to cache and memory capabilities as a result.

Request coalescing: Second, in contrast to CPUs, which execute up to tens of separate concurrent threads, GPUs maintain contexts for thousands of threads through hierarchical grouping. These groups of threads can exploit spatial locality by coalescing requests from separate threads into a small number of cache accesses when the requests are to neighboring addresses.

To understand the impacts of request coalescing, we ran tests that vary the coalescing

degree from no GPU coalescing to full 32-thread group coalescing. Figure 2.3 shows the geometric mean reduction in number of cache accesses and the ROI speedup for varying coalescing degree normalized to the case with no coalescing. In the common case, full 32-thread coalescing reduces the number of global memory accesses by 10–16× as the GPU is frequently able to coalesce 16 or 32 integer or floating-point requests to a single cache line access.

This data also shows that most of the performance gains come from coalescing across sets of 4–8 threads, as further gains are limited to less than 3% run time in the common case. With wider coalescing, the GPU sends fewer memory accesses to the cache hierarchy and reduces upper-level cache pressure. Thus, memory access bottlenecks tend to move to hierarchy levels below the core-L1 interface. Despite the small potential for performance gains, coalescing beyond 8 threads continues to decrease the total number of cache accesses (up to 2× in the geometric mean), which can result in a decrease in cache access power/energy.

Since GPU request coalescing behaves similarly to CPU single-instruction, multiple-data (SIMD) vectorization, we also ran tests to find the effect of SIMD vectorization on memory access counts. We use the gcc compiler automatic SSE 4 vectorization, which allows the CPU to load, operate on, and store data on up to 4-wide integer or single-precision floating point vectors with single instructions, and has a similar effect to coalescing in terms of the width of memory accesses to the cache hierarchy. Gcc is able to vectorize ROI portions of four applications from our set: `backprop`, `hotspot`, `srad` and `strmclstr`. For these four applications, vectorization reduces the total number of memory accesses by 1.32–1.69× (1.44× geometric mean), and that most of the eliminated accesses are to heap data. Compared to GPU 4-thread coalescing, however, automatic compiler vectorization fails to vectorize as many instructions, so CPU code sees more limited reduction in memory access counts.

Overall, GPU scratch memory and request coalescing reduce the number of global memory accesses by 18–100× compared to CPU applications (27× in the geometric mean).

Table 2.4: Memory access locality metrics by core type.

	OpenMP:		Access	CUDA:		Access
	% L1	% L2	Per	% L1	% L2	Per
Application	Hits	Hits	Line	Hits	Hits	Line
`backprop`	95.6	68.1	42.2	39.6	75.7	5.6
`bfs`	95.5	18.1	4.0	37.6	62.2	3.7
`heartwall`	99.7	89.3	273	87.7	93.2	19.1
`hotspot`	98.9	16.2	57.4	37.1	65.3	4.5
`kmeans`	99.4	0.5	132	44.1	18.9	4.9
`pathfind`	98.8	45.8	35.6	46.6	17.0	2.3
`srad`	97.7	61.0	54.0	19.7	57.3	2.5
`strmclstr`	97.4	17.0	5.3	25.5	66.5	2.8

Compared to CPU cores, this reduction alleviates pressure on caches, which in turn allows GPU cores to operate at lower frequencies while still serving data to threads at rates comparable to or greater than CPU cores.

2.4.2 Spatial Locality

As we observe memory accesses flowing through the cache hierarchy, we see that L1 caches play different locality filtering roles for the CPU and GPU sides of the architecture. Specifically, in contrast to mostly-scalar CPU cache accesses which must hit in cache to provide strong performance, GPU request coalescing dramatically cuts the accesses to each cache line, which in turn alleviates L1 cache pressure and can expose more MLP to lower levels of the hierarchy.

For these data-parallel applications, CPU threads have extremely high spatial locality, typically striding through all elements in a heap cache line in subsequent algorithm loop iterations. These access patterns, which also include accesses to stack/local memory that is persistent over many loop iterations, result in high L1 cache hit rates that even exceed those expected by simple strided read memory access. Table 2.4 lists these hit rates and the average number of accesses per heap cache line per algorithm pipeline stage.

CPU vectorization and GPU coalescing are designed to capture address spatial locality before memory requests are sent to the caches. Thus, these techniques cause a reduction in

the available spatial locality to caches by a factor equal to the effective access width (i.e., up to 1.69× for 4-wide vectorization and more than 14× for 32-wide coalescing in common cases).

To get a sense for the remaining spatial locality in the GPU cache access stream, we observe counts of the number of accesses to each unique global memory cache line during GPU kernels, and we find that lines are typically accessed between 2 and 5 times. Note that most GPU kernels move data from global memory into scratch for local handling, so the few remaining spatially local accesses are likely caused by separate thread groups accessing the same data rather than thread groups being unable to fully coalesce accesses. In either case, there is little spatial locality left to exploit within each kernel without increasing cache line sizes, and GPU thread group and coalescing widths.

Since vectorization and coalescing cut down the number of spatially local accesses to each cache line, they have the effect of exposing wider access parallelism below the L1 caches. This effect is subtle in lock-up free caches: Fewer accesses to each cache line reduces the number of accesses that may occupy miss-status handling registers (MSHRs) queued for a small set of outstanding accesses to lower levels of the cache hierarchy. We find that GPU 32-wide coalescing increases the number of concurrent memory accesses to the L2 cache by 1.3–3× over uncoalesced GPU memory accesses. Hence, vectorization and coalescing are both fundamental means for better exposing MLP to lower levels of the memory hierarchy by decreasing MSHR pressure caused by accesses that queue for a small set of lines.

2.4.3 Temporal Locality

The prior subsection described how core microarchitecture and L1 caches capture a significant portion of access spatial locality for both CPU and GPU applications. This result suggests that little spatial locality is left for L2 caches to capture. Instead, they serve to extract access temporal locality from separate data sharers.

In all OpenMP applications, the CPU L1 hit rate is above 95%, as each heap cache line is accessed numerous times, and many local variables are accessed some number of times across loop iterations. By comparing the L1 cache hit rates with the common number of memory accesses per cache line, we can see that in all applications, the L1 caches must be capturing nearly all of the intra-thread spatially local accesses to each line. For example, strided accesses to 32 consecutive data elements in each cache line would result in a 31/32 = 96.9% hit rate in the absence of intervening accesses. This result suggests that the per-thread working set of most applications fits in the CPU 64kB L1 cache. Given this observation, L1 cache locality leaves the L2 caches mostly responsible for capturing temporally local accesses to data shared across cores rather than temporally or spatially local accesses to data previously evicted from the L1 caches due to limited capacity.

Since GPU coalescing often reduces each warp memory instruction to access just a small number of cache lines, there is diminished importance to ensuring temporally local accesses to cache lines. Observing the GPU L1 cache hit rates and common access counts per line, we note that the L1 caches capture fewer than half of the multiple accesses to each cache line in the common case, and more accesses go to the GPU L2 cache than in CPU applications. To establish whether this behavior is a result of contention for GPU L1 cache capacity or data sharing across GPU cores, we ran tests that vary the GPU L1 cache capacity up to 256kB, and we find that L1 hit rates improve by at most 5% with the extra capacity. This result indicates that instead of competing for L1 capacity, GPU threads from separate cores are generating most of the temporally local accesses to single cache lines, similar to the CPU L2.

Based on the above observations, we find that CPU and GPU L1 caches have very different importance, though their filtering roles are similar. In the aggregate for data-parallel workloads, CPU L1 caches have many responsibilities; they must be designed to capture both the spatial locality for heap data accesses and the temporal locality of stack accesses. Fortunately for data-parallel workloads, these responsibilities rarely conflict given

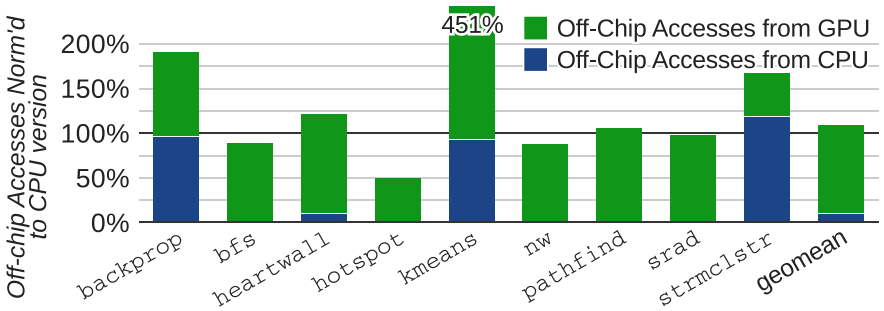


Figure 2.4: Number of off-chip memory accesses, normalized to CPU version.

sufficient L1 capacity, so CPU L1s are quite effective and important for capturing locality.

For GPU applications, register and scratch memory can shift local variable accesses away from the caches, which eliminates the L1 responsibility for capturing temporally local stack requests. Further, GPU coalescing greatly reduces the importance of spatial locality across separate heap accesses, so the L1 caches are mostly responsible for capturing the small number of temporally local accesses from separate GPU threads on the same core, diminishing the overall responsibility of the GPU L1s compared to CPU L1s.

In contrast to L1 caches, L2 caches play a similar role for both CPU and GPU cores. For both core types, the majority of spatial and temporal request locality is captured by components at higher levels of the memory hierarchy, which usually leaves the L2 caches responsible for capturing access locality to data shared among separate cores. We did not find any circumstances in which the L2s function to capture significant temporal request locality from L1 capacity spills.

2.4.4 Number of Off-Chip Accesses

When observing memory request locality, we noted that L2 caches play a similar role in filtering memory requests for both CPU and GPU cores. This filtering causes CPU and GPU applications to have similar spatial locality of memory accesses to off-chip memory. Here, we observe off-chip access counts in support of this hypothesis. Figure 2.4 plots the number of off-chip memory accesses for all GPU applications normalized to the multithreaded

CPU version.

We start by noting some important similarities between CPU and GPU applications. In particular, for applications that execute similar code per output data element (`backprop`, `bfs`, `heartwall`, `pathfind`, `srad` and `strmclstr`), the number of off-chip memory accesses from the GPU is nearly identical to the analogous portions of the CPU version. In addition, we note that the GPU version of `hotspot` uses a pyramid iterative algorithm that completes two timesteps per GPU kernel launch, which cuts the number of times that the GPU streams data on-chip by a factor of precisely two compared to the CPU version. While a non-trivial transformation, the CPU version could also implement pyramiding to gain potential benefits of reduced off-chip data access.

The major differences between CPU and GPU off-chip access counts typically arise due to the overheads of off-loading computation to the GPU cores. The CPU-GPU coordinated reduction operations in `backprop`, `kmeans`, and `strmclstr` require that the CPU stream GPU-generated intermediate data, which is too large to fit in on-chip caches. In each case, these accesses account for roughly as many off-chip accesses as the OpenMP versions of the applications. As we will see later, this extra data streaming, rather than elevated op counts, accounts for the performance overhead in these applications.

Second, given the way that the `kmeans` algorithm is mapped to the GPU, it does not take advantage of local memory in its inner loop (recall, the CPU version spills registers to L1 cache as a local memory). This results in the GPU streaming all data points once for each of the k centers that are being considered in a single iteration. The input set we consider in this work has $k = 5$, and indeed, we see that `kmeans` on the GPU must stream data on-chip nearly 5 times more than the CPU version. The CUDA version could be modified to store the same local variables as the CPU version to eliminate these extra off-chip accesses, though the transformation would be non-trivial.

Finally, we find that memory footprint can contribute to second order effects on off-chip accesses. In particular, `heartwall` and `nw` double-buffer data in their GPU and CPU versions,

Table 2.5: ROI Requested Off-chip Bandwidth (GB/s).

	CPU cores			GPU cores		
	Avg	Stdev	Max	Avg	Stdev	Max
backprop	7.0	4.1	20.3	11.1	10.4	83.9
bfs	7.0	6.8	30.0	13.8	12.0	94.0
heartwall	0.3	0.9	20.8	1.1	4.8	77.3
hotspot	3.6	1.8	16.3	2.9	4.0	35.6
kmeans	1.0	0.6	11.7	23.8	6.0	80.8
nw	4.3	1.7	16.8	7.5	7.8	82.4
pathfind	5.0	1.6	16.3	6.2	4.2	79.3
srad	4.9	3.0	20.3	10.1	8.1	87.9
strmclstr	5.2	3.0	16.8	16.5	13.1	114.9

respectively. These extra buffers result in up to 10% differences in memory footprint for their respective cores, and we find that access counts are analogously affected.

We also note that we find little difference in the volume of off-chip memory accesses ($< 1\%$) for GPU applications as we vary the coalescing degree. This result indicates that, like their CPU counterparts, the GPU cache hierarchy is able to capture the majority of spatially local accesses to heap cache lines when coalescing is unable to, albeit with possible overheads in performance or power.

2.4.5 Bandwidth Demands

While we have noted that CPU and GPU L2 caches play similar roles in capturing memory access locality, here, we demonstrate that CPU and GPU cores expose very different MLP over time that results in a substantial difference in their requested memory bandwidth rates.

To evaluate differences in off-chip memory bandwidth demand, we simulated both application versions with a 32GB/s off-chip memory interface and collected memory access interarrival times at the memory controller. From these interarrival times, we calculated instantaneous requested bandwidth over time intervals of 500 memory controller cycles and used them to estimate the average, standard deviation, and maximum requested bandwidth as listed in Table 2.5.

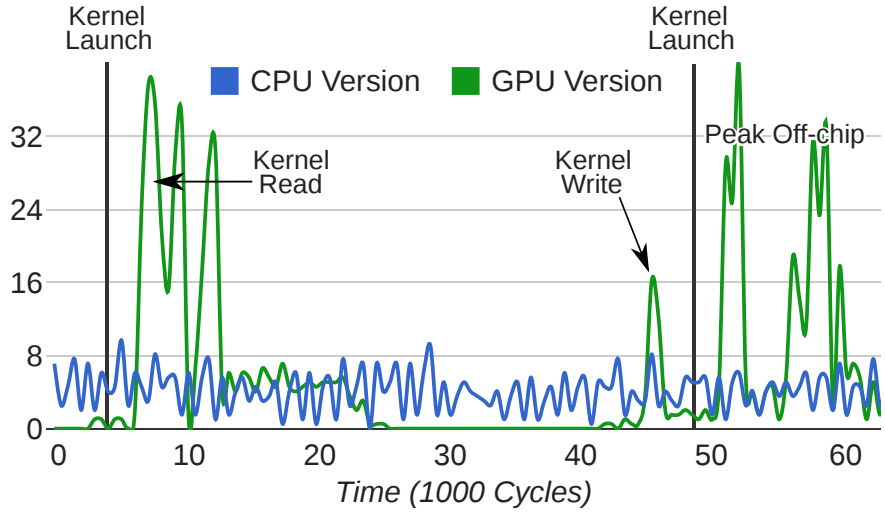


Figure 2.5: `nw` requested off-chip bandwidth (GB/s) over time.

This data shows that GPU cores nearly always demand greater average bandwidth and greater variance over time than the CPU. Compared to the CPU versions, all GPU applications here show a greater frequency of high instantaneous requested bandwidth. Further, the GPU's instantaneous requested bandwidth regularly exceeds the peak theoretical limit of the off-chip interface by more than 2.5×, while CPU cores only request up to the peak.

To tie requested bandwidth back to application-level characteristics, we observe requested bandwidth over time for a representative portion of the `nw` application. Figure 2.5 plots the requested off-chip bandwidth from both core types for this region. The blue line shows the average requested bandwidth over 100-cycle time windows from the CPU cores. The behavior in this plot is common to most CPU applications, namely CPU cache accesses miss in fairly regular intervals, which cause regular accesses through time and consistent requested bandwidth through each phase of the application.

By contrast, GPU cores expose very bursty memory accesses, frequently exceeding the peak available bandwidth. The green line plots the GPU requested bandwidth with annotations for the start/end of GPU kernels ("Kernel Launch"), and it depicts the common GPU kernel stages we described previously. Shortly after a kernel launch, numerous thread blocks issue parallel memory accesses that begin reading data from the global memory

space into the core's scratch memory. These reads are issued en masse, causing the cache hierarchy and memory system to fill with buffered accesses. After reading data, the GPU often accesses data in scratch—a time period with few or no global accesses from each thread block—and then the data is written back to global memory as a burst ("Kernel Write").

Burst access behavior is common to all GPU applications we tested, and the character tends to be largely similar to `nw`. We chose to plot `nw` because it clearly demonstrates the read-compute-write character of GPU thread blocks. However, `nw` has low GPU thread occupancy, which limits its ability to further push bandwidth limitations while thread blocks are computing on data. Many applications execute more concurrent thread blocks/groups or have lower ops-to-byte ratios, which often result in larger or more frequent access bursts. When near bandwidth saturation, the distinction of kernel read/write bursts can blur as buffers fill and dependent memory accesses modulate the issue of further outstanding accesses.

The key takeaway here is that GPU burst access behavior results from the way that GPUs group and launch threads. Specifically, at the beginning of a kernel, all capable thread block contexts begin executing at roughly the same time, which can cause very large bursts of independent accesses. Following this initial burst, smaller but still significant access bursts occur each time a new thread block begins executing or when thread groups pass synchronization events. By contrast, CPU cache access filtering tends to modulate the core's ability to issue nearly as many parallel accesses to off-chip memory.

2.5 Application Performance Comparison

The last section showed CPU and GPU memory access similarities and differences, and we want to understand their application-level effects. Here, we show that the majority of performance differences between CPU and GPU versions is, in fact, directly attributable to

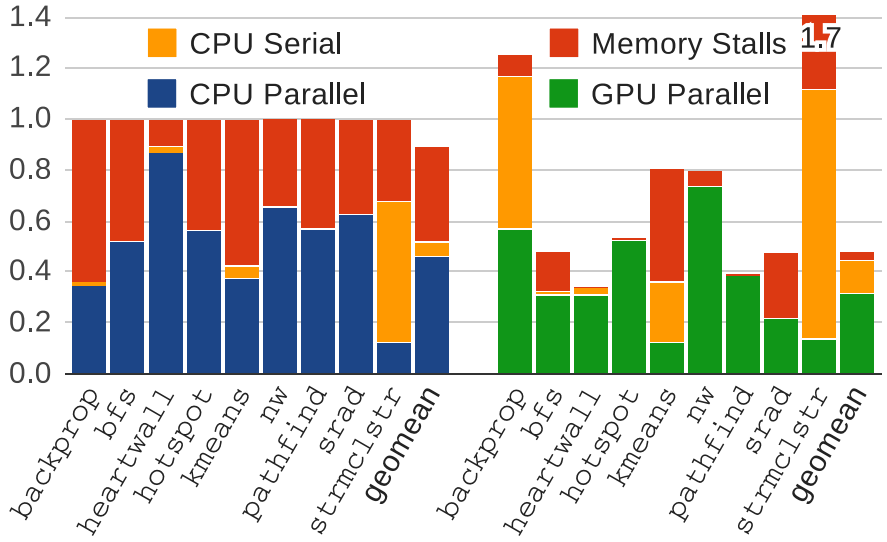


Figure 2.6: ROI run time normalized CPU version.

the memory access behavior differences. Specifically, the CPU versions struggle to keep up with GPU versions due to MLP limitations which cause memory stalling. This memory stalling results in elevated memory access latency sensitivity for CPU cores, while GPU cores are better able to leverage available bandwidth.

2.5.1 Performance Depends on Memory Stalling

First, we show that the primary difference in performance between CPU and GPU applications is a result of memory stalling. Figure 2.6 plots the ROI run times of applications run on CPU cores (left cluster of bars) and GPU cores (right cluster) normalized to the CPU's run time. The figure indicates that CPU versions tend to struggle to keep up with the GPU versions despite comparable peak FLOP rates.

To establish that memory access is the substantial portion of the difference in run time between CPU and GPU versions, we ran tests which cut the memory hierarchy latency to nearly zero cycles as a way to estimate the portion of run time attributable to memory hierarchy performance. Cache and memory bank conflicts were minimal, so we describe that the resulting stalls come largely from each core's ability to expose MLP. The run time gains shifting from the realistic memory hierarchy design to the optimal hierarchy are

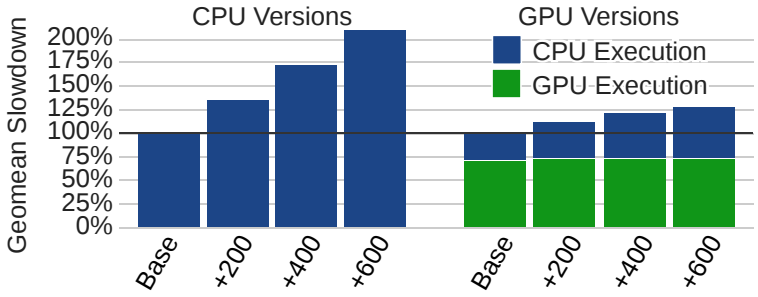


Figure 2.7: Geometric mean slowdown over all applications from 200, 400, and 600 additional off-chip memory access cycles.

reflected in the "Memory Stalls" portion of each run time bar.

These memory stall estimates indicate that in the common case, CPU applications suffer memory stalls for 30–50% of run time. If these stalls could be removed from the CPU version run times, four CPU applications—`bfs`, `hotspot`, `nw`, and `srad`—would come within 25% of their respective GPU versions, and the geometric mean across all applications would be within 8%. Remaining second-order performance differences are attributable to data communication overhead (`backprop`, `kmeans`, `strmclstr`), elevated CPU compute ops (`heartwall`), and control-flow ILP limitations (`pathfind`).

2.5.2 Latency Sensitivity

Ultimately, memory request dependencies in CPU thread instruction streams regulate their ability to issue many concurrent outstanding memory requests below the L1 caches and to hide memory access latency. In contrast, GPU cores leverage their deep multithreading to expose wide MLP and issue access bursts to lower levels of the cache hierarchy. It is a common belief that this difference allows GPU cores to hide very long memory access latency.

We confirm this belief by running simulations with additional no-load, off-chip access latencies of 200, 400 and 600 memory cycles, which increase the baseline off-chip access latency roughly 3–7×. Figure 2.7 presents, for both versions, the geometric mean ROI slowdowns over all applications normalized to the baseline memory. The CPU applications

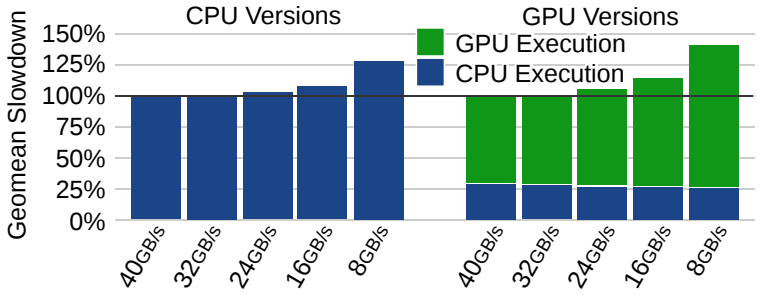


Figure 2.8: Geometric mean slowdown over all applications from limiting bandwidth, normalized to 40 GB/s off-chip.

show immense sensitivity (>2×) to additional latency compared to the GPU versions. Even in the GPU applications, the vast majority of latency sensitivity comes from the CPU portion of execution time, while the GPU portions of execution show a maximum slowdown of 5% across all applications.

2.5.3 Bandwidth Sensitivity

Since GPU applications tend to demand greater instantaneous bandwidth from off-chip memory than CPU applications, we would expect that GPU cores should be more sensitive to changes in the peak theoretical bandwidth of the off-chip interface. We quantify the difference in bandwidth sensitivity by running tests that vary off-chip bandwidth from 8 to 40 GB/s. Figure 2.8 plots, for each system, the geometric mean ROI slowdown over all applications normalized to a system with 40 GB/s peak off-chip bandwidth. These tests confirm that for the complete ROI, GPU applications are 6–10% more sensitive to available memory bandwidth. Further, the GPU parallel portion of these applications is up to 60% more sensitive to bandwidth than the CPU applications.

2.5.4 Discussion

The performance and memory sensitivity of these applications is directly related to a core's ability to expose MLP. If we follow CPU and GPU memory access paths, we see where constraints on MLP arise. CPU cores executing fewer threads must extract most MLP from

ILP, which poses challenges for parallel accesses below L1 caches, while GPU cores tend to extract greater MLP across a large set of concurrent grouped threads, and to spread memory accesses over a larger instantaneous set of cache lines.

Given the latency sensitivity of CPU cores, there is performance incentive for programmers to aim for high L1 cache hit rates as we see with these applications. These high hit rates typically result from tight code loops and strided memory access, which result in many sequential cache hits to each cache line. Unfortunately, these sequential cache hits limit the core's ability to expose MLP to lower levels of the cache hierarchy. Accesses to the L1 cache are spatially local, so they trigger infrequent cache misses to cause accesses to lower levels. As our tests show, this cache access behavior even exists when using aggressive out-of-order CPU cores, which are capable of continuing execution well beyond waiting cache misses.

By restructuring loop iterations as separate threads and gathering spatial locality across threads, GPU applications tend to avoid the sequential accesses to single cache lines. GPU request coalescing and the use of scratch memory substantially reduce the number of memory accesses that go to the caches, and we described how this reduced number of accesses can increase MLP. Specifically, fewer, less-local accesses reduce pressure on MSHR queuing and increase the number of concurrent parallel memory accesses that can proceed to lower levels of the memory hierarchy.

There are a few options for multicore CPUs to mitigate the effects of low-exposed MLP. First, programmers can parallelize the memory access portion of instruction streams by managing multiple strided access streams in each loop iteration. This structure tends to be complicated, so other hardware techniques have been developed. Simultaneous multithreading (SMT) gives the perspective that there are more CPU threads, and indeed, our test show that adding threads can help hide some of the memory stalls. However, SMT elevates contention for the sequential access to L1 caches. CPU SIMD vectorization can mitigate the number of L1 cache accesses, but does not reduce the number of accesses as

dramatically as GPU request coalescing. Finally, CPU cache access prefetching can eliminate much of the latency spent waiting for regular or strided memory access. However, we expect that applications, such as `bfs`, with irregular memory access will still tend to perform better on GPU cores by being able to issue many parallel accesses to hide latency.

2.6 Microarchitecture Key Implications and Related Work

We expect that most of the challenges and opportunities in heterogeneous system design will arise in shared components that will need to balance GPU bursty access behavior against CPU access latency sensitivity. We expect that applications will find it useful to share data between CPU and GPU cores, and this could mean sharing caches, on-chip interconnect, and off-chip memory.

2.6.1 Cache Hierarchy Sharing

Trying to share caches between CPU and GPU cores will likely require techniques to ensure latency-critical CPU data remains on-chip as appropriate. GPU cache filtering behavior and bursty memory access results in high volumes and high instantaneous rates of data invalidations and writebacks. If shared caches give equal capacity priority to CPU and GPU data, the GPU's operation can turn over cached data very quickly and erratically. Assuming there are applications that could benefit from sharing caches, we expect that new techniques will look to more advanced partitioning schemes, capacity prioritization, and replacement policies to address this challenge.

2.6.2 Interconnect and Off-chip Memory Scheduling

Sharing interconnect and off-chip memory among CPU and GPU cores will require quality-of-service (QoS) techniques to appropriately balance bandwidth and prioritize memory accesses. Many prior QoS techniques (e.g., [72, 39]) consider longer-run average statistics as

proxies to approximate the application-level performance impact of bandwidth allocation options or delaying memory accesses to prioritize others. However, given the substantial differences in CPU and GPU core architectures and instantaneous memory bandwidth demands, these existing proxies may be insufficient to adequately predict and compare CPU and GPU application performance impacts. Further, most prior techniques are applied in the context of symmetric multicore processors in which there is an assumption that even bandwidth allocation is likely to impact application performance evenly. Our results show that this assumption is not likely to hold up in the context of general-purpose heterogeneous processors, because GPUs tend to be more sensitive to effective bandwidth, while CPUs tend to be more sensitive to access latency. Prior work does investigate scheduling techniques to meet real-time graphics processing constraints in heterogeneous processors [61, 9], but it is likely that estimating appropriate bandwidth balance and finer-grained access prioritization for general-purpose heterogeneous processors will be a broad area of future work.

2.6.3 Emerging Memory Technologies

In addition to the challenges posed above, we see opportunity to leverage emerging memory technologies to improve memory latency, bandwidth, and power efficiency. Stacked and on-package DRAM options promise small latency and power improvements, and are likely to provide bandwidth comparable to existing discrete GPUs [89]. We estimate that these improvements will increase CPU application performance by upwards of 5% and GPU performance by as much as 15%.

A shortcoming of available stacked and on-package DRAM options is their small capacity, which is limited by area and thermal constraints. To effectively use these memories, it will be important to store latency- and bandwidth-critical data in them. System designers will need to consider methods for allowing data to be mapped and migrated between memories, while programmers will desire means to intelligently control data placement.

2.6.4 Related Work

To the best of our knowledge, this work is the first to present a quantitative characterization comparing the application and memory system behavior of applications mapped to both CPU and GPU cores with the express aim of illuminating their microarchitectural similarities and differences. We perform this analysis for data-parallel applications.

Memory System Characterizations: A wide range of papers have characterized memory system behavior for parallel applications [17, 150], cloud/server applications [13, 90], GPU applications [21, 62], and mobile/embedded applications [30, 50, 51]. While these prior studies characterized the applications on a single type of platform, our work examines different implementations of the same applications for two different core types with the express goal of understanding the differences in architectural mapping.

Performance Comparisons: Another class of related work seeks to compare the performance of applications implemented for different architectures or using different parallelization techniques [74, 65, 46]. Of particular significance, Lee et al. compare the performance of a set of applications parallelized to run on either CPU or GPU hardware [79]. That work focuses on the application tuning required to push hard performance limitations: peak FLOP rates and bandwidth limitations. Our work echoes many of the findings of [79], but goes beyond by using tightly controlled simulations to illuminate the precise memory system microarchitecture behavior differences that can cause performance differences.

2.7 Summary

As heterogeneous cores become more tightly integrated onto the same die and share the same system resources, understanding the memory system requirements of the cores becomes more critical. This chapter presents the first detailed analysis of memory system behavior and effects for applications mapped to both CPU and GPU cores.

Our results show that while the applications are designed with similar algorithmic

structures, their mapping to different core types can result in dramatically different memory system characteristics. Specifically, GPU coalescing and scratch memory greatly reduce the importance of L1 caches compared to CPU L1s, which must capture immense spatial locality to ensure performance. GPUs, with their deep multithreading, more readily expose wide bursts of parallel memory accesses to the off-chip interface, which results in greater sensitivity to bandwidth and diminished sensitivity to memory access latency. These memory behaviors are the primary factor in their performance differences.

3 CHARACTERIZING GPU COMPUTING SOFTWARE PIPELINES

Chapter 2 gives us an understanding of differences between CPU and GPU core behaviors, and identifies application-level characteristics that indicate which core type will perform better on CPU or GPU cores. As we start modifying GPU computing applications to target heterogeneous processors, the primary challenge is to identify application stages that show the characteristics could be optimized to better utilize CPU or GPU cores.

To decide which portions of an application to modify, we find it very valuable to collect metrics about higher-level algorithm and application characteristics. Specifically, we investigate their software pipeline structures, the dynamic location of computation and data (i.e., in CPU or GPU memories), and the way that they communicate data between pipeline stages. Using these metrics, we can estimate the potential performance benefits of better core and cache utilization.

This chapter presents an analysis of GPU computing software pipelines to further guide our heterogeneous processor application development. We identify software pipeline structures that signal opportunities to overlap computation and communication, and to migrate work between core types. First, as a result of the split between CPU and GPU memories in discrete GPU systems, many existing GPU computing applications move data and computation to GPUs and let CPU cores sit idle for long periods of run time. In heterogeneous processors, this organization often underutilizes CPU cores, which share access to data and could participate in computation. Overlapping CPU and GPU computation would better utilize the cores.

Second, although many GPU computing applications contain pipeline stages with wide data-level parallelism (DLP) fit for GPUs, many classes of applications also show varying DLP across separate pipeline stages that is fundamental to the algorithmic structure. Existing applications can leave GPU cores idle during narrow-DLP stages. Often, CPU cores could perform better, but in discrete GPU systems, moving data to and from the

CPU's memory can perform worse than just using the GPU. In a heterogeneous processor, the programmer can consider migrating these stages to CPU cores.

Finally, there is significant opportunity for applications to better leverage caches in heterogeneous processors. Applications usually employ producer-consumer communication between pipeline stages, and the communicated data often exceeds the size of cache. When pipeline stages execute sequentially, data communicated between them spills to off-chip memory. Performance and energy suffer from the excess off-chip memory access caused by these spills.

Overall, our findings show that there is potential to gain 1.25–10× performance when executing applications in heterogeneous processors. Further, the performance improvements will reduce application static energy and caching improvements will reduce total memory access energy. We identify particular applications with the greatest potential and describe the types of software transformations that will be required to offer these gains. The findings in this chapter are published in the Proceedings of the International IEEE Symposium on Workload Characterization (IISWC) 2015 [55].

3.1 Introduction

Heterogeneous systems are evolving to allow tighter CPU and GPU interaction. Many new systems allow GPUs to access CPU physical memory and both cores to reference memory with the same virtual addresses. These emerging capabilities can reduce the effort to develop applications that correctly access complex data structures used by both CPU and GPU cores [59, 106]. However, since data must still move across the PCIe bus in discrete GPU systems (previously shown in Figure 1.1), programmers must still carefully manage data movement to achieve good performance.

Emerging heterogeneous processors integrate coherent communication fabrics among CPU and GPU cores [8, 109, 149]. Further, tightly-integrated processors even include cache

coherence among cores [57, 129], an area of active research [119, 75]. These architectures can mitigate costly memory transfers and allow CPU and GPU cores to perform fine-grained communication and synchronization in cache.

Currently, it is unclear how programmers may try to use these emerging system features. These new processors and their application programming interfaces (APIs), such as OpenCL 2.0 [69], are seeing growing adoption. However, few current benchmarks exercise the performance capabilities of unified virtual memory architectures, and no publicly-available benchmarks exercise cache coherent heterogeneous processor capabilities.

To better understand the potential evolution of applications and architectures for heterogeneous CPU-GPU processors, this chapter compares a broad set of existing, publicly available GPU computing applications against ported versions that remove memory copies. We then analytically quantify the compute and cache inefficiencies to identify likely optimization targets. Heterogeneous processors are likely to benefit from three major optimizations:

- Reducing the temporal distance between data producer and consumer tasks using finer-grained communication and concurrently executing compute stages.
- Identifying task data-independence and leveraging mechanisms to migrate independent work to underutilized cores.
- Detecting memory access contention and appropriately modulating access to increase cache efficiency.

Overall, this analysis shows that heterogeneous processors offer greater compute and cache efficiency opportunities compared to discrete GPU systems. While removing copy overheads from current applications results in modest performance improvement, still half of all memory accesses result from cache contention caused by residual GPU kernel-granularity synchronization. Most applications with high cache inefficiency also show

bandwidth limitations, so improving cache utilization should directly decrease bandwidth demand and increase application performance.

After reviewing potential gains from these optimizations, this chapter discusses software and hardware directions that may improve programmability and performance of applications executing on heterogeneous processors. Software constructs should offer more flexible compute and data granularities. Hardware mechanisms should support lighter-weight thread handling and producer-consumer communication in caches.

The rest of this chapter is organized as follows: Section 3.2 presents a case study motivating deeper exploration of GPU computing pipeline structures. Section 3.3 articulates the simulation methodology used to compare discrete GPU and heterogeneous processor systems, and Section 3.4 quantifies the comparison. Section 3.5 describes and quantifies analytical estimates of run time and memory access improvements in heterogeneous processors. Section 3.6 discusses implications and related work, and Section 3.7 concludes.

3.2 Motivating Producer-Consumer Support

To motivate pipeline structure investigation, we begin with a case study of the `kmeans` application running in our simulation environment. `Kmeans` shows significant compute and caching inefficiency due to the bulk-synchronous pipeline structure, and up to 77% of run time can be recovered by restructuring the application to run on a heterogeneous processor. The optimizations tested here on `kmeans` have widely varying potential benefits for other applications and input sets, but further results show they are broadly applicable optimization targets for heterogeneous processors.

3.2.1 Example `Kmeans` Application

From the Rodinia suite, the `kmeans` application iteratively analyzes a set of n-dimensional points to find the k points that characterize clusters of the points. Each iteration involves

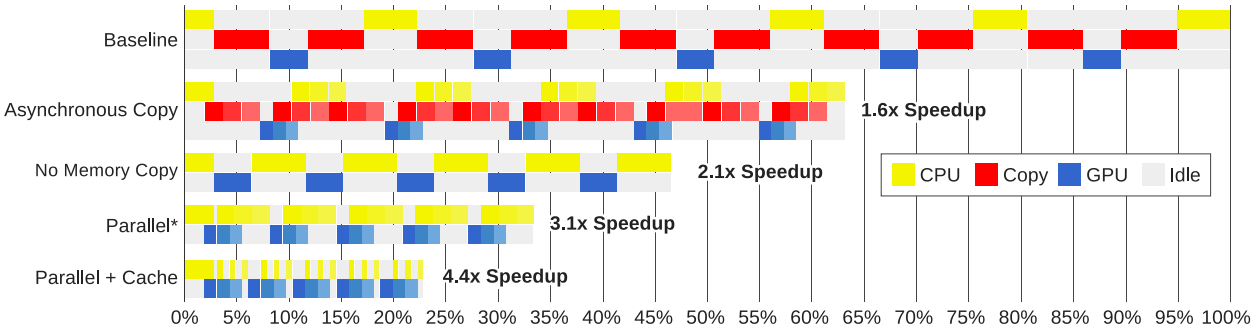


Figure 3.1: `Kmeans` simulated and estimated (*) run times for various application organizations.

calculating the distance between each of the points and the current `k` centers, assigning each point to the closest center, and then replacing poor centers with new candidate centers. Distance calculations and center assignments have wide thread-level parallelism (TLP) and so are performed on the GPU. The center replacement algorithm has limited TLP, so assigned centers are copied back from the GPU memory to perform the center adjustment on the CPU.

Baseline: When using copies in the discrete GPU setting, `kmeans` serializes nearly all of the work and copies. Run time component activity is depicted in Figure 3.1 as "Baseline". Despite only transferring a small amount of data between CPU and GPU memories in each iteration, over 50% of `kmeans` run time is spent copying data. This overhead is due to the asymmetry of PCIe bandwidth (8GB/s) compared to the CPU and GPU memory bandwidth (24 and 179GB/s), which allow the CPU and GPU to process data substantially faster than a PCIe copy.

Bandwidth asymmetry in discrete GPU systems encourages programmers to minimize data transfers often resulting in wide, bulk-synchronous pipeline stages. For `kmeans`, the GPU sits idle for a substantial portion of run time (82%) though the GPU completes 95% of the compute operations, indicating that `kmeans` incurs very high GPU FLOP opportunity cost[1] for bulk transferring work between CPU and GPU.

[1]We refer to "FLOP opportunity cost" as the portion of compute FLOPs that go unused due to a core being inactive

3.2.2 Optimizing `Kmeans`

Programmers can use a number of existing techniques to restructure application pipelines and improve performance. However, to date, little analysis has compared different inefficiencies and opportunity costs of optimizing GPU application structure for discrete GPUs versus heterogeneous processors. For `kmeans` operating on this particular input set, the programmer's incentive to optimize increases substantially when running on a heterogeneous processor. Specifically, removing memory copies provides a 2× run time improvement, but 2× more improvement can come from further CPU-GPU parallelism and effective cache management. It is difficult or impossible to employ these optimizations in current discrete GPU systems.

Asynchronous Memory Copy Streams: In the discrete GPU system, `kmeans` performance is hamstrung by the need to copy data back and forth between CPU and GPU memories. One option to reduce this overhead is to use kernel fission and asynchronous streams [104, 151]. Kernel fission requires the programmer to explicitly divide independent data and compute chunks of a kernel into separate kernels that can be overlapped with asynchronous memory copies. The "Asynchronous Copy" bars of Figure 3.1 show the run time activity for a 3-wide asynchronous stream organization.

While a non-trivial code transformation, kernel fission and streams can improve `kmeans` run time by 37%. Memory copies can be overlapped with CPU and GPU execution, though there are data dependencies that the limit overlap. Despite the data dependencies, `kmeans` run time could improve up to the point that the PCIe link is saturated for the full execution.

Emerging unified virtual memory architectures exist that allow coherent data synchronization between CPU and GPU over the PCIe link. However, we expect that the latency to perform these on-demand synchronizations will be too prohibitive to allow data handling as efficiently as streams. For `kmeans`, performance is likely to still bottleneck on copies, because the total data copied would remain the same.

Eliminating Memory Copies: In Figure 3.1, the "No Memory Copy" bars show the

CPU and GPU activity of `kmeans` running on a cache-coherent heterogeneous processor without the need for memory copies. Without the copies, run time can improve over the baseline execution by nearly the total baseline copy time, and GPU utilization improves from 18% to 39%.

Unfortunately, this organization is still quite core and cache inefficient. First, this organization leaves either CPU or GPU cores idle throughout the complete execution. In terms of available compute operations, this organization incurs an opportunity cost of nearly 60% unused FLOPs. Further, this `kmeans` implementation was designed for a discrete GPU and minimal copy overhead, which encouraged GPU kernel-granularity synchronization. This residual structure results in very inefficient use of cache. Each GPU kernel streams input and output data, and the total size of this data exceeds the size of cache, causing all produced data to spill off-chip before they are consumed. This spilling results in roughly 9.5% more memory accesses than if these data could be passed in cache.

Parallel Producer-Consumer Compute: With a discrete GPU, overlapping memory copies and computation requires techniques like kernel fission to split the copies and kernels into parallelizeable chunks and synchronize data. In a heterogeneous processor, however, data synchronization between CPU and GPU can happen in coherent memory, possibly eliminating the need for kernel fission by allowing CPU-GPU communication with simple memory reads and writes.

In Figure 3.1, we estimate the run time if CPU consumer code runs immediately after GPU producers generate their output ("Parallel"). This estimate assumes similar core execution times as the asynchronous streams version, and analogously, performance improves up to the point that some component bottlenecks run time. In the heterogeneous processor, the CPU becomes the bottleneck, but the overlapped execution results in a 40% run time improvement over the no-copy case, and GPU utilization rises to nearly 65%.

Improved Heterogeneous Processor Caching: While the parallel producer-consumer estimate showed improved core utilizations, in actual simulation of this application orga-

nization, performance improves still more due to caching. If the GPU and CPU work is chunked to synchronize small enough intermediate data between cores, the CPU is able to access all of its data out of cache. These cache hits dramatically reduce CPU memory access latency, which dominated the CPU execution time. Figure 3.1 ("Parallel + Cache") shows that these caching benefits improve `kmeans` run time by another 32%, and GPU utilization reaches 80%.

3.2.3 Motivation Summary

Overall, `kmeans` exhibits the major optimization opportunities that may become common in heterogeneous processors. First, overlapping core activity can reduce the opportunity cost of underutilized cores. Such an optimization is likely to be more straightforward when cores can communicate through memory rather than using PCIe transfers. Second, bringing producer and consumer tasks into closer temporal proximity has potential to greatly improve the use of cache, an optimization that is difficult in current discrete GPUs. The final optimized `kmeans` here leaves CPU cores underutilized, and further optimization could focus on migrating computation to those cores.

3.3 Software Pipeline Characterization Methodology

3.3.1 Simulated System Configurations

This chapter compares memory access and performance effects of optimizations for discrete GPUs and forward-looking heterogeneous processors. To control performance capabilities and allow flexible system architectures, we simulate the systems with configuration parameters defined in Table 3.1. Both systems use the same CPU and GPU cores, and their compute capabilities are comparable to current mid-range discrete GPU systems or aggressive heterogeneous processors. We configure CPU and GPU cores the same way as in the microarchitecture characterization in Chapter 2.

Table 3.1: Heterogeneous system parameters.

Component	Parameters
CPU Cores	(4×) 4-wide out-of-order, x86 cores, 3.5GHz
CPU Caches	Per-core 32kB L1I + 64kB L1D and exclusive, private 256kB L2 cache, 128B lines
GPU Cores	(16×) Fermi-like SMs, 700MHz, Greedy-then-oldest warp scheduler [123], 8 CTAs, 48 warps of 32 threads, 48kB scratch memory, 32k registers
GPU Caches	24kB L1 per-core. GPU-shared, banked, non-inclusive L2 cache 1MB, 128B lines
Discrete GPU System	
Interconnects	CPU L2s/MCs: 6-port switch, GPU L1/L2: Dance-hall, GPU L2s/MCs: Direct links
CPU Memory	(2×) DDR3-1600 channels, 24 GB/s peak
GPU Memory	(4×) GDDR5 channels, 179 GB/s peak
PCI Express	v2.0 x16, 8 GB/s peak
Heterogeneous CPU-GPU Processor	
Interconnects	GPU L1/L2: Dance-hall, All L2s/MCs: High-bandwidth, 12-port switch
Memory	(4×) shared GDDR5 channels, 179 GB/s peak

The discrete GPU system models the split CPU and GPU caches and memories. The CPU chip accesses DDR3 memory capable of up to 24 GB/s peak, while the discrete GPU chip has 4 GDDR5 memory channels capable of up to 179 GB/s peak. Consistent with many current discrete GPU systems, memory copies are performed using a PCIe link with peak bandwidth of 8 GB/s between CPU and GPU memories. When data is copied between CPU and GPU memories, any coherent cache lines containing data for the destination addresses are written back or invalidated.

To limit performance effects resulting from memory bandwidth differences, the heterogeneous processor CPU and GPU cores share access to the same GDDR5 memory as the GPU in the discrete system. The bandwidth of GDDR5 is likely to be comparable to emerging memory technologies, such as 2.5D/3D stacked DRAM, which may be used with heterogeneous processors. For the tests in this study, CPU and GPU memory access contention has marginal effect compared to other application-level differences described later.

Table 3.2: Producer-consumer relationships in applications.

					Producer-Consumer Constructs		
Suite	Tested Here	Number Benches	P-C Comm.	Pipeline Parallel	Regular Accesses	Irregular Accesses	Software Queue
Lonestar	Y	14	14	13	14	13	10
Pannotia	Y	10	10	10	10	10	0
Parboil	Y	12	8	8	8	3	1
Polybench		15	10	10	9	0	0
Rodinia	Y	22	19	18	19	6	0
Total		73	61	59	60	32	11
Portion		100%	84%	81%	82%	44%	15%

3.3.2 gem5-gpu Simulator

To perform tests with well-controlled and flexible system architectures, we use the gem5-gpu simulator [120]. gem5-gpu offers full-system simulation of discrete GPU systems and heterogeneous processors with flexible memory hierarchies and PCIe configurations. The GPU model is from GPGPU-Sim v3.2.2 [10], and the CPU cores are the out-of-order model in gem5 [18]. All tests use Linux kernel 2.6.28.4.

3.3.3 Applications

Depending on their application-level pipeline structure, GPU computing applications can see widely varying memory copy overheads and potential optimization targets. As we aim to explore a broad range of these effects, this study tests applications from four open-source GPU computing suites. Table 3.2 summarizes details about the application-level structures of all applications in these suites.

The Lonestar GPU suite [21] contains many applications with irregular control flow and memory access behaviors (“Irregular”). Many of these applications operate on graph-like data structures, and many use software queues, or “worklists”, for tracking available work (“SW Queue”). Similar to Lonestar, the Pannotia applications [23] perform various graph analyses, though each is structured to expose available work without software queues. Pannotia applications are implemented with OpenCL, but ported to CUDA for this study. Representing some more traditional GPU computing workloads, this chapter also

characterizes the Parboil [138] and Rodinia [24] suites. These suites contain many image and signal processing, machine learning, and scientific numerical workloads, as well as two graph handling applications. Though we do not test applications from Polybench [118], we list the suite here to note its applications contain regular vector and matrix math operations, many of which have potential to benefit from heterogeneous processors. Of the total 73 applications in these five suites, we examine 46 that work fully in gem5-gpu and perform non-trivial computations.

Table 3.2 also lists counts of application pipeline characteristics. Most applications (84%) contain multiple producer-consumer pipeline interactions ("P-C Comm."), including CPU execution, GPU kernels, or memory copies between CPU and GPU memories. Of these 61 applications with producer-consumer relationships, all but two could be parallelized to run pipeline stages concurrently or in closer temporal proximity than the unmodified applications ("Pipeline Parallel"). Section 3.5.1 investigates the potential gains from such parallelization.

3.3.4 Application Configurations

Memory Copies: To characterize the differences between applications running on discrete GPUs and heterogeneous processors, we run applications with two different memory copy configurations. In discrete GPU simulation, we run applications largely unchanged from their publicly available versions[2], which use CUDA to allocate and copy memory between CPU and GPU memory spaces.

To model potential gains of heterogeneous processors with unified virtual and physical memory spaces, the second application configuration removes data copies between memory spaces. We use a combination of CUDA library and manual application modifications to eliminate separate CPU and GPU copies of data. Specifically, for memory allocations that mirror CPU allocations into the GPU memory space, we allow the GPU to access the CPU

[2]Application versions working in gem5-gpu are available open-source with the simulator: http://gem5-gpu.cs.wisc.edu/repo.

allocation directly and eliminate the GPU memory allocation. The CUDA library identifies and eliminates many allocations by observing runtime dynamic CUDA calls.

We also manually modify some applications to eliminate memory copies. Many allocations only serve to double-buffer mirrored data. In these cases, the GPU has multiple versions of the same mirrored CPU data, but runtime analysis cannot safely eliminate redundant allocations or copies. Using consistent, explicit copies between CPU and GPU allocations is often sufficient to allow the CUDA library to eliminate copies.

These two techniques eliminate the substantial majority of memory copy overheads. Specifically, all but one application (Lonestar `bh`) see reduced number of copies, and 24 of the 46 applications have at most 1 remaining memory copy. Because some copies still remain, we refer to this application version as "limited-copy". The next section characterizes improvements from removing copies.

Data Location Assumptions: For each of the applications, we delineate the portion of run time during which data handling and computation occur as the region of interest (ROI). The ROI begins after the CPU sets up all necessary data to be resident in its physical memory, but before the CPU transfers any data to the GPU memory space or is allowed to launch GPU kernels. The only exception is Rodinia `mummer`, which also reads data from disk while the GPU is executing. Before the ROI begins, the application is allowed to allocate memory to be used by the GPU, but these allocations cannot have been accessed prior to the start of the ROI. We define ROI completion as the time at which all output data produced by the application is once again available in the CPU memory. In the discrete GPU case, this definition means that resulting outputs are copied back to the CPU memory space within the ROI. In the heterogeneous processor, the ROI can end after the last CPU or GPU activity completes to generate final output.

This ROI definition is important for a couple reasons. Prior work shows that data copies must be accounted for when analyzing application-level performance [46], so especially when directly analyzing memory copy overheads. Further, many open-source GPU comput-

ing applications actually perform subsets of full GPU computing applications. Real world applications may shuttle data through other compute kernels rather than reading from disk or randomizing inputs, which are common input methods for applications observed here. In this case, compute kernels executed prior to and following the kernels within our ROIs would require that data end up in a designated location such as CPU-accessible memory.

GPU Memory Management: Besides the architectural differences between discrete GPUs and heterogeneous processors, a notable difference between these systems is GPU memory management. In the discrete GPU setting, GPU memory is not accessible from CPU cores, so the CPU need not have access to GPU address translations. A GPU-specific memory allocator is allowed to map memory for address translations while the PCIe copy engine or GPU are executing, and GPU minor page faults are handled completely by the GPU.

In the heterogeneous processor, however, memory mappings must be consistent across CPU and GPU. Thus, both CPU and GPU address translation hardware must access a common page table, and page table updates must be simultaneously visible to both. To achieve these translation capabilities, gem5-gpu implements GPU page faults in a manner similar to IOMMU page faults available in recent Linux kernel versions (e.g., v3.19). Specifically, the GPU raises an interrupt to the CPU, which performs memory page mapping and returns the mapping to the GPU.

Prior studies in GPU address translation [121, 117] have described implementations of efficient GPU address translation, including TLB and page table walk structures. However, they have not explicitly studied the performance impact of GPU page faults. The results in this chapter show that CPU-handled GPU page faults can cause significant performance degradation, and we suggest this area for future research.

Input Set Selection: Application input sets were chosen to ensure that the ROIs meet the following criteria. First, all applications execute at least 1 billion instructions combined

for both the CPU and GPU, and total instructions typically exceed 2.1 billion. The most instructions executed by any application is 90 billion. Second, since the GPU completes a majority of work, input sets were chosen to ensure that total GPU execution time is at least 5ms and typically more than 30ms. The longest running ROI is 1.535s. Finally, input sets were chosen so that total memory footprint is at least 6MB and usually greater than 42MB for copy application versions. Limited-copy memory footprints are at least 3.5MB and usually greater than 24MB. While these footprints are smaller than common applications, they are larger than L2 caches, so they show memory access behaviors consistent with real applications.

3.4 Eliminating Memory Copies

Most existing GPU computing applications were developed for discrete GPU systems and thus use explicit memory copies to move data between CPU and GPU memories. As a baseline for further core and cache inefficiency analysis, this section observes how memory copies affect some basic application characteristics: memory footprint, memory access counts, and ROI run time.

The statistics below show that the GPU-kernel-synchronous structure of existing applications limit the immediate use of available cores and cache in heterogeneous processors. Removing memory copies often decreases application memory footprint and total memory accesses, and run times typically decrease by the total time of eliminated memory copies. However, the aggregate number of memory accesses from CPU and GPU cores tends to remain similar after removing copies, and CPU and GPU cores see limited utilization improvements.

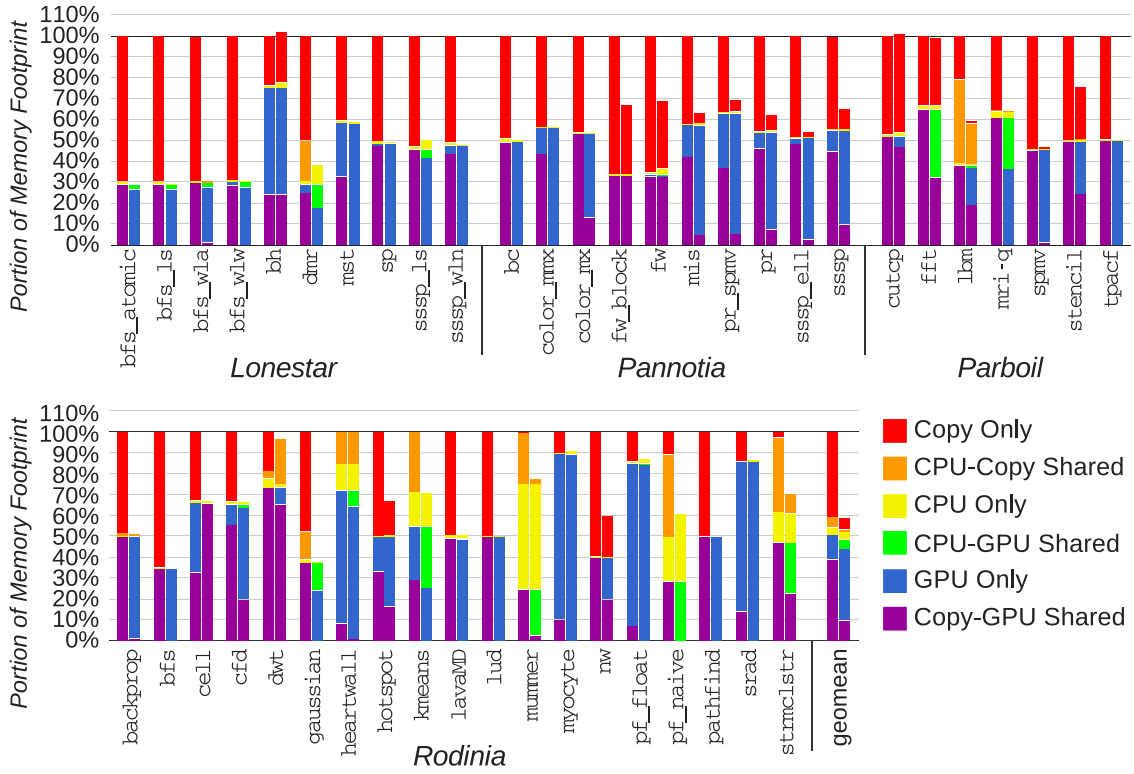


Figure 3.2: Breakdown of memory footprint touched by component type for copy (left bars) and limited-copy (right bars) versions normalized to the copy version.

3.4.1 Memory Footprint

The limited-copy application versions typically just mirror data between CPU and GPU memories, so eliminating mirrored data can significantly reduce the total memory footprint. We measure application memory footprint by observing the addresses of all memory accesses from CPU and GPU cores, and in the discrete GPU system, the PCIe copy engine. Figure 3.2 breaks down these footprints into mutually exclusive subsets touched by one or more components for the copy and limited-copy versions of the applications. The plot is normalized to the total memory footprint of the copy version of each application (left bar of each pair) to show how the memory footprint decreases when eliminating mirrored data copies in the heterogeneous processor setting (right bars).

First, most copy applications replicate data from the CPU to GPU memory. Copy portions of the bars (red, orange, purple) make up nearly all of each bar, indicating that

most of a application's data is copied at some point. In a few other applications, such as Lonestar `bh` and Rodinia `srad`, the GPU uses substantial temporary data that is only ever resident in GPU memory. This GPU-temporary memory often stores large sets of intermediate data that are passed between GPU kernels and cannot be statically bound to GPU scratch memory.

Second, some object-oriented and graph-based computations do not touch their whole data sets though their copy versions need to move that data to GPU memory. For example, in Lonestar `bfs` and Pannotia `fw` applications, the copy engine touches nearly all of the data, but the CPU and GPU combined touch less than one-third of that data. Here, the CPU and GPU traverse the data's structure, but do not necessarily need to touch all data for the desired computations. For applications with this character, prior work shows that memory copies can be saved using smart page placement or on-demand page migration to the GPU rather than memory copies [2, 3].

Of the remaining limited-copy memory footprint, applications use the GPU to process the majority of data—more than 70% on average. In few cases (Lonestar `bh`, Parboil `cutcp` and `fft`, and Rodinia `dwt` and `heartwall`), our memory copy elimination techniques are unable to remove the majority of copied footprint. However, more extensive manual application modification should be able to remove all of these copies.

3.4.2 Memory Accesses

As expected, the copy applications also incur excess memory accesses for moving data between CPU and GPU memory spaces. Figure 3.3 shows each application's total memory accesses broken down by component type for the copy (left bars) and limited-copy application versions (right bars). Most commonly, copy accesses account for 4–10% of total memory accesses, but in a substantial subset of applications, copies account for more than 20% of total memory accesses.

For applications with a small portion of copy accesses, CPU and GPU cores often

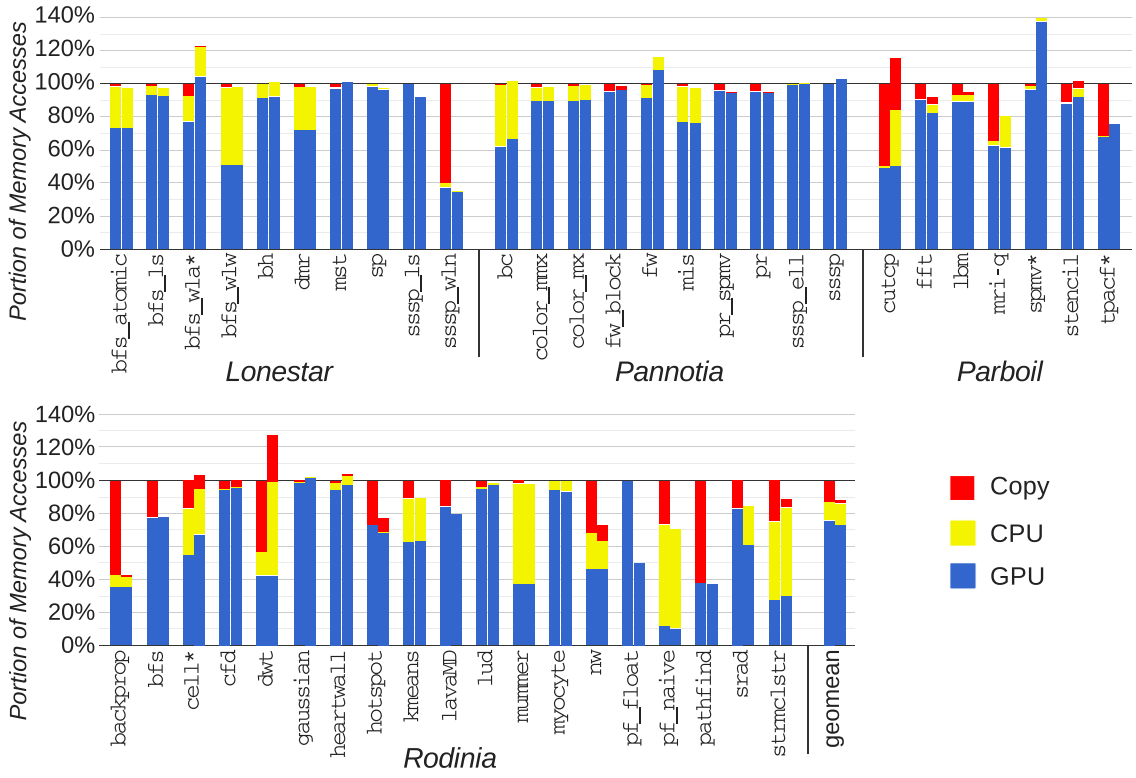


Figure 3.3: Memory access breakdown by component type for copy (left bars) and limited-copy (right bars) versions normalized to copy version.

perform multiple accesses per data element. For most Lonestar and Pannotia applications, memory copies account for at most 5% of total memory accesses, because CPU and GPU work perform multiple traversals of and modifications to irregular data structures, such as graphs. Similarly, applications, such as Rodinia `gaussian` and `lud`, perform iterative refinement to the majority of their data, so copies account for a small portion of memory accesses.

When memory copies are removed, Figure 3.3 shows that typically all copy memory accesses are eliminated. In the geometric mean, the number of total copy accesses declines by more than 11%. Further, it is common that the CPU and GPU memory access counts remain substantially similar in the limited-copy case. Despite the CPU and GPU being able to share data in caches of heterogeneous processors, the structure of the applications is such that little cache efficiency can be gained simply from removing memory copies. We speak more to this in Section 3.5.3.

Though memory access counts do not indicate any systematic caching improvements when moving to a heterogeneous processor, there are two uncommon conditions under which removing memory copies can significantly change GPU memory access counts. First, when CPU and GPU share memory, the GPU must rely on the CPU for page fault handling, which can upset memory access orderings. For Rodinia `srad`, page fault handling causes accesses to be shifted from the GPU to the CPU, which clears memory pages during page mapping. For Rodinia `pf_float`, the page fault handler causes serialization of some GPU memory accesses, which substantially reduces GPU cache contention and cuts off-chip accesses by 50%. In Pannotia `fw`, GPU access serialization limits L1 cache locality, which results in an increase in off-chip accesses. These behaviors are exceptional, though the affected applications share other characteristics common to other applications.

Another cause of increased GPU memory access counts is memory allocation misalignment, which affects GPU coalescing and cache contention. Specifically, since memory allocations are no longer managed by the CUDA library, which cache-line-aligns GPU allocations, CPU-GPU-shared allocations can lack good alignment. As a result, GPU access coalescing can result in more memory accesses to caches, and stress the ability of the cache to capture temporal locality while streaming data. The applications marked with '$*$' in Figure 3.3 experience this elevated cache contention. Nearly all of the extra memory accesses result from misalignment and could be avoided by using an aligned memory allocator.

3.4.3 Run Time

On average, removing memory copies results in modest performance improvement. Figure 3.4 shows the run time for copy and limited-copy application versions broken down by the portion of run time during which each component is active. Overall, only CPU-side work can benefit from heterogeneous processor caching and copy removal. In the aggregate, removing memory copies results in a 7% run time improvement, which we break down

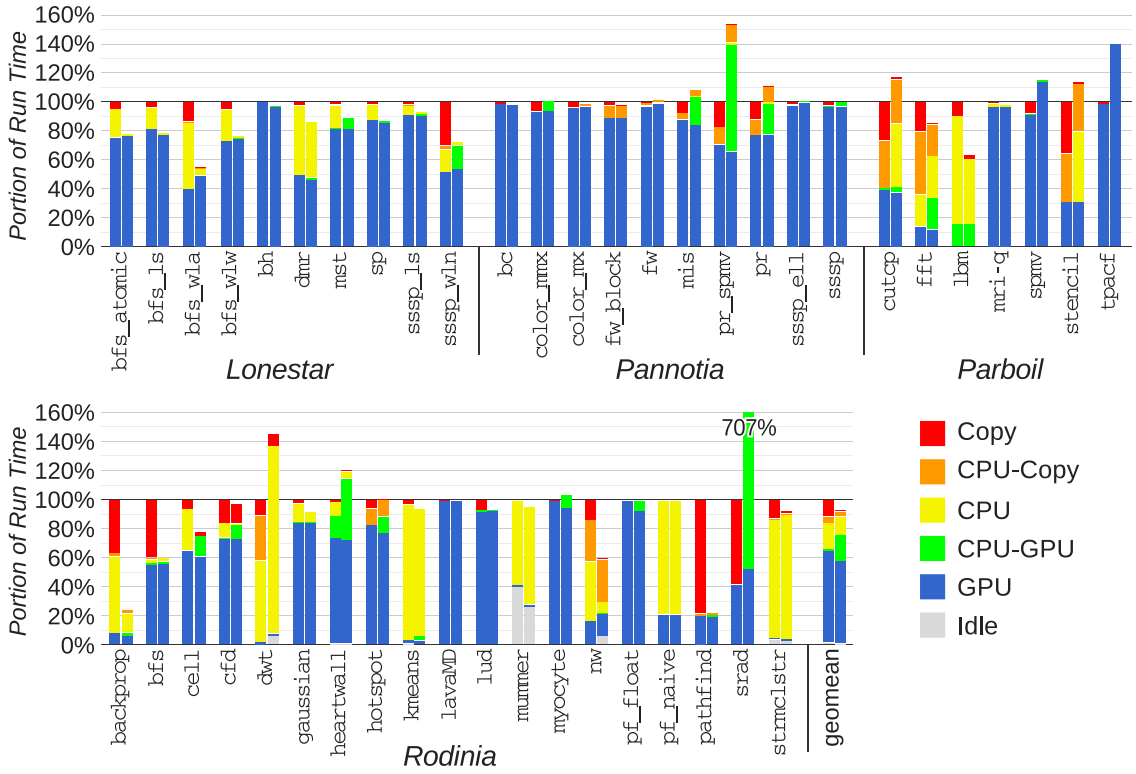


Figure 3.4: Run time component activity breakdown for copy (left bars) and limited-copy (right bars) versions normalized to copy run time.

below.

To the first order, limited-copy application run times improve due to reduced memory copy and CPU execution time. The reduction in memory copies eliminates much of the run time during which the PCIe copy engine is exclusively handling data, as demonstrated by applications like Rodinia `bfs` and `pathfind`. This benefit is common for many applications and in the geometric mean accounts for an 11% run time improvement.

As a first major caching benefit in heterogeneous processors, applications with significant CPU execution time in their copy versions often see run time improvement due to improved caching and memory copy removal. As exemplified by most Lonestar applications, even small memory copies invalidate data from CPU caches when moving that data between CPU and GPU memories. Since CPU execution is often latency-sensitive, CPU progress is slow as it reads data back into caches from off-chip memory after PCIe transfers. This cache invalidation is often avoided in the limited-copy applications, resulting in a

geometric mean 6% run time improvement.

This run time plot also illustrates detrimental first order performance effects of address translation. As mentioned, GPU page faults cause serialization of some GPU memory accesses, which would be executed in parallel if not waiting on the CPU page fault handler. On average, this serialization causes a 9% GPU slowdown, though the majority of this slowdown is experienced by just three applications: Pannotia `pr_spmv`, and Rodinia `heartwall` and `srad` (7× slowdown). In these applications, numerous would-be parallel GPU writes go to unmapped global memory and must wait on the serialized CPU page fault handler for address mappings.

3.5 Software Pipeline Optimization

As noted in the case study, the bulk-synchronous pipeline structure of GPU computing applications leads to core and cache inefficiency. In fact, it is widely believed that most existing GPU computing applications perform serialized CPU and GPU portions of run time. Figure 3.4 confirms this belief by showing that most execution time for both copy and limited-copy applications is exclusively running the copy (red), CPU (yellow), or GPU (blue). This result indicates that there is potential to improve performance by overlapping execution of these components.

In this section, we estimate how eliminating memory copies can change the potential gains of application restructuring to better leverage available parallel resources. We identify three opportunities to improve resource utilization and compare them in the following subsections. The estimates indicate that better core utilization could result in run time gains greater than 20%, and we find significant room to improve cache efficiency in the heterogeneous processor setting.

3.5.1 Overlapping Communication and Computation

For both copy and limited-copy application versions, a reasonable first cut at improving performance is to run the same code, but try to expose more overlap of CPU, copy, and GPU activity. For the discrete GPU, such parallelism might be achieved with kernel fission and asynchronous streams as described previously. In the heterogeneous processor, data could be passed in memory between CPU and GPU. Specifically, CPU or GPU consumer threads could be launched, and set to wait for in-memory signals indicating when data is ready to be consumed. The producer threads can set these signal variables as their generated results become available. This software organization could use similar data blocking structure as kernel fission + asynchronous streams, but may avoid the need to split GPU kernels and manage separate kernel streams since threads synchronize in memory.

To test these application transformations broadly would be a tremendous amount of work. However, we can employ analytical modeling to get a sense for their benefits. Using an Amdahl's Law-like calculation, we estimate potential performance gains from overlapping component activity for all applications. The following formula estimates component-overlap run time, R_{co}:

$$R_{co} = C_{serial} + max(C - C_{serial}, P, G) \tag{3.1}$$

Here, C, P, and G are the CPU, copy, and GPU portions of run time, respectively. Since the CPU acts as the control component, some of its activity strictly cannot be overlapped. C_{serial} accounts for non-overlapped kernel and memory copy launch portions of CPU run time. We estimate this limitation by iterating through pipeline stage statistics and identifying copies and kernel launches that occur while no other kernels or copies are executing to mask the launch latency. C_{serial} can account for up to 9% of application run time, but only for a few applications with numerous, serialized kernels and copies, such as Lonestar `sssp_wln` and Rodinia `bfs`.

We briefly validate the component-overlap model by applying application transfor-

mations to three copy and limited-copy application versions: `backprop`, `kmeans`, and `strmclstr`. Each of these applications is structured with wide data-level parallelism per kernel instance, and this structure is common to more than half of the applications in this study. We chunk the kernel input and output data either to apply kernel fission + asynchronous streams in the discrete GPU system, or in-memory "data ready" signal variables in the heterogeneous processor. By chunking the data to execute at least four concurrent streams, the run time of each application improves to within 3.1% of the component-overlap estimate.

While the component-overlap model is accurate for some applications, it can still be optimistic for a couple of reasons. First, the estimate does not account for wide data dependencies from one pipeline stage to the next that may limit parallelism (e.g., Lonestar `dmr` or Parboil `fft`), or memory access contention that may occur from overlapping component activity. Second, often to extract more core activity parallelism, programmers must add more pipeline control code or synchronization primitives, possibly increasing run time. If, however, the heterogeneous processor sees improved caching effects, performance could improve beyond this estimate, as observed in `kmeans`.

Analysis: Component-overlap run time estimates for copy and limited-copy application versions are depicted in Figure 3.5, and normalized to the baseline copy run time. These results suggest that overlapping communication and computation has the potential to eliminate much of the performance difference between copy and limited copy versions.

Two classes of copy applications are likely to benefit substantially from component-overlap optimizations. First, many applications with regularly structured data/computation have significant memory copy overheads. Due to their regular structure, it is straightforward to use kernel fission and asynchronous streams to overlap copies with computation, and the potential performance gains often bring them in line with the limited-copy applications. Applications in this class include Parboil `cutcp` and `stencil`, and Rodinia `backprop`, `cell`, `cfd`, `hotspot`, `kmeans`, `nw`, `srad`, and `strmclstr`.

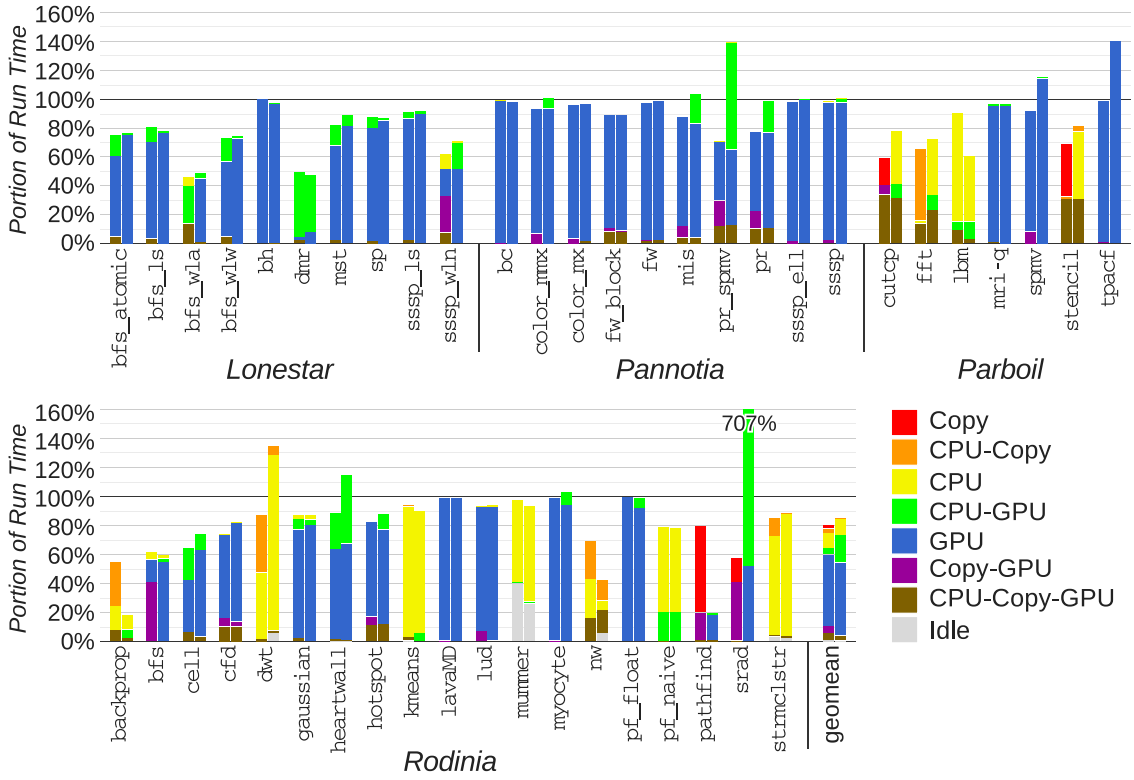


Figure 3.5: Estimated: Component-overlap run time breakdown for copy (left bars) and limited-copy (right bars) versions normalized to baseline copy run time.

The second class of applications has a common outer-loop structure executed by the CPU. Examples include Lonestar applications and Rodinia `bfs` in which the CPU launches GPU kernels and then waits to decide whether to continue loop execution until kernels complete and results are copied back. Often the deciding factor is whether the GPU generated more work during the last GPU kernel, and this factor can be triggered at any time during the kernel. This structure could be optimized with mechanisms to signal the CPU thread as soon as the loop condition became true to allow the CPU to overlap its control code with the running kernel.

3.5.2 Migrating Compute Between Cores

Though overlapped execution could improve application run times by 10–15%, many applications still experience significant FLOP opportunity cost. The majority of this under-

utilization results from poor balance of work across CPU and GPU cores. Most frequently, CPU cores are left idle for nearly all application run time, but for some applications, GPU cores are left idle for significant portions of run time (e.g., Rodinia `dwt`, `kmeans`, and `strmclstr`).

To quantify the effect of this core underutilization, this subsection estimates potential performance gains from migrating compute between CPU and GPU cores. Migration can be achieved through a number of mechanisms, including identifying portions of computation that can be hoisted from the beginning/end of one pipeline stage to the prior/next stage, or identifying data-independent portions of a pipeline stage that can be computed on underutilized cores. In discrete GPU systems, work can be migrated between CPU and GPU cores, though such transformations are often unwieldy since they also require migrating data. In heterogeneous processors with shared physical memory, on the other hand, splitting parallel work across core types to improve core utilizations may be easier.

This optimistic migrated-compute estimate assumes that all application compute phases can be effectively distributed across CPU and GPU cores. Since performance would not be able to exceed hard resource limitations, the migrated-compute estimate uses two limiting factors. First, computation cannot exceed the FLOP rate of cores to which it is migrated, so we estimate a peak-FLOP/s-relative run time, $R_{mc_{core}}$. Second, effective memory bandwidth cannot exceed the peak bandwidth to off-chip memory, so we estimate the limit on run time improvement running up against this bound ($R_{mc_{BW}}$). Perfect migrated-compute run time, R_{mc}, is estimated from these as follows:

$$R_{mc_{core}} = \frac{C * F_{cpu} + G * F_{gpu}}{F_{cpu} + F_{gpu}} \tag{3.2}$$

$$R_{mc_{BW}} = M/BW_{mem} \tag{3.3}$$

$$R_{mc} = max(P, R_{mc_{core}}, R_{mc_{BW}}) \tag{3.4}$$

As above, C, P, and G are the CPU, copy, and GPU portions of run time. F_{cpu} and F_{gpu}

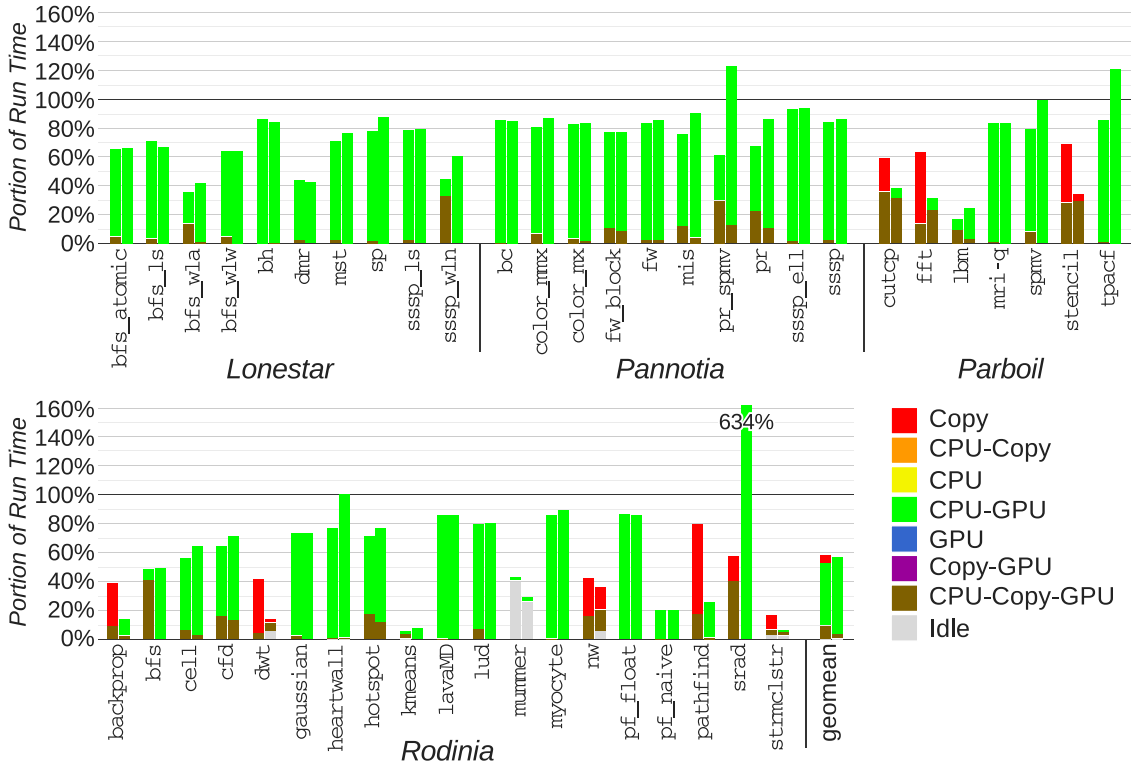


Figure 3.6: Estimated: Migrated-compute run time breakdown for copy (left bars) and limited-copy (right bars) versions normalized to baseline copy run time.

are the CPU and GPU peak FLOP rates, respectively. M is the total number of memory accesses, and BW_{mem} is the peak achieved memory bandwidth, which generally tops out at about 82% of peak pin bandwidth.

Validating the migrated-compute model is difficult compared to the compute-overlap model, because few programming constructs exist for such transformations. However, we applied manual program transformations to the `kmeans` and `strmclstr` copy application version that indicate that migrated-compute estimated gains are plausible. Specifically, both applications perform matrix-vector and reduction-like operations on CPU cores, and we rewrote these operations to run on GPU cores as part of preceding kernels. Where appropriate, we further utilized GPU atomic operations to bring effective FLOP rates near the GPU theoretical peak. These transformations reduced the amount of data transferred between GPU and CPU memories, and improved run time by more than 2.5× to within 35% of the compute-overlap estimates.

Figure 3.6 shows the results of the migrated-compute run time estimates for copy (left bar in each pair) and limited-copy (right bar) application versions. In the common cases, such as in Lonestar and Pannotia, the results indicate that fully utilizing compute resources could improve performance by another 4–13% by moving GPU work to idle CPU cores. Such migration may be relatively straightforward if the CPU executes tasks indexed similarly to GPU threads. On the other hand, when CPU execution dominates baseline run time (e.g., Rodinia `dwt`), the potential gains are substantially larger, because shifting more parallelism to the GPU could reduce the substantial, unused GPU FLOPs.

Applications with substantial baseline CPU execution time often contain pipeline organization or memory handling inefficiencies that result from the need to copy data between CPU and GPU memory spaces. For example, inefficient pipeline organizations sometimes contain medium-to-high TLP-capable work that the CPU performs using a single thread. Migrating this compute to the GPU would often require copying data to the GPU in the discrete GPU setting. Many of these applications already have high memory copy overheads in the discrete GPU setting, so migrating data may actually degrade performance. These inefficiencies indicate that migrating compute is likely to be easier in heterogeneous processors. Parboil `cutcp` and `lbm`, and Rodinia `backprop`, `dwt`, `kmeans`, `pf_naive`, and `strmclstr` contain such inefficiency.

Other applications with substantial CPU execution time have high data movement overheads. Parboil `fft` and `stencil`, and Rodinia `mummer` contain memory copies for double buffering data, or clearing memory regions, which are costly CPU operations. We expect these could each be optimized for heterogeneous processors by improving their data structures to eliminate the need for these memory operations.

Finally, it is likely that remaining copy-dominated applications (20% of applications with red portions in Figure 3.6 bars) will be difficult to optimize on discrete GPUs. In most of these cases, this performance bottleneck is largely fundamental to the computation being performed: Significant data must be moved relative the amount of computation

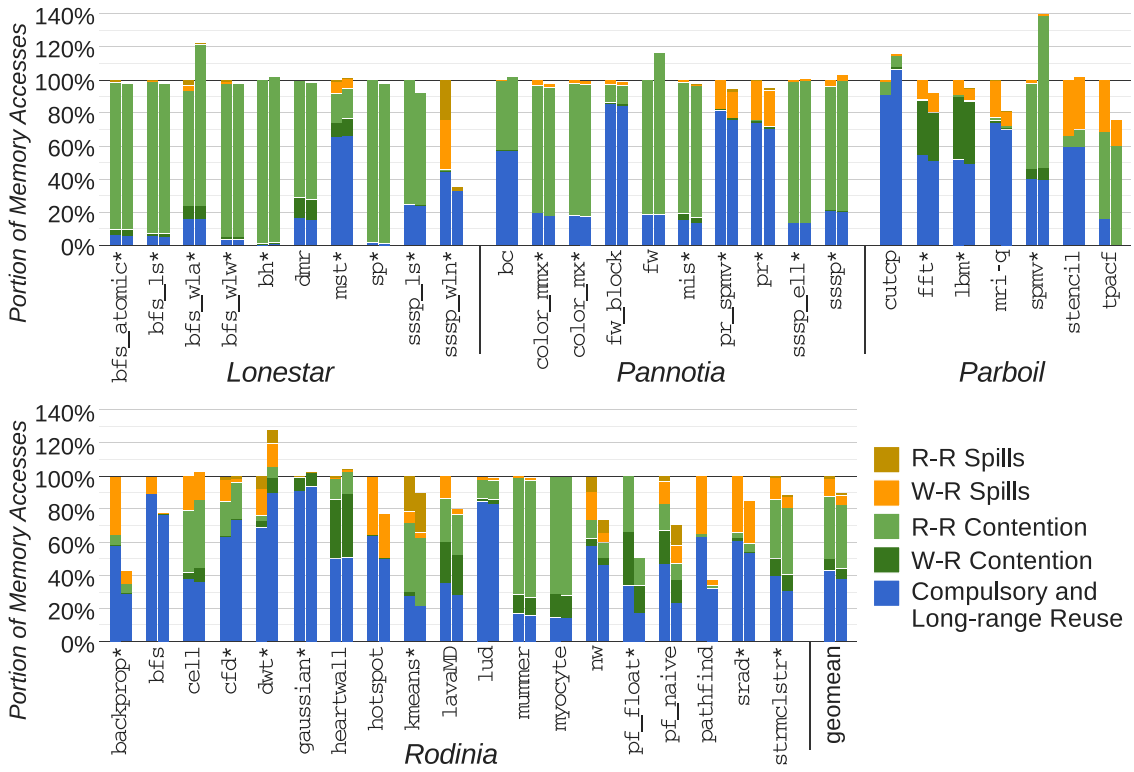


Figure 3.7: Memory accesses broken down by cause for copy (left bars) and limited-copy (right bars) versions normalized to copy application versions.

completed on that data. Further, of these applications, Parboil `fft`, Rodinia `backprop`, `nw`, and `strmclstr` contain many-to-few data dependencies between pipeline stages, suggesting that inter-stage optimization will be difficult in the presence of memory copies.

3.5.3 Coordinated Use of Cache Capacity

The performance gain estimates above do not account for an important feature of heterogeneous processors not available in discrete GPU systems: shared CPU-GPU caching. The GPU-kernel-synchronous pipeline structure of existing applications causes large per-pipeline-stage memory footprints, which frequently push data out of cache before it can be reused. Without memory copies to work around, heterogeneous processors have flexibility to improve shared cache management and increase performance.

To quantify opportunities for better cache data reuse, we inspect and categorize off-chip memory accesses based on their relationship to other memory accesses. At the off-chip

interface, we record whether a cache block was previously accessed off-chip, the type of the access (read or write), the type of the previous access, and the reuse distance in terms of pipeline stages since previous access. We identify four classes of memory access that may be reduced or eliminated through better application organization or caching. Figure 3.7 shows the breakdown of these memory accesses.

Required memory accesses: Compulsory memory accesses include the first off-chip read from and last write to a piece of data. Such accesses must occur to complete the computation, so they cannot be eliminated. Grouped with compulsory accesses (blue), long-range reuse accesses occur to data that has been previously accessed, but the time between the accesses spans multiple pipeline stages. Reducing these accesses may be possible, but would probably require substantial application restructuring to improve temporal locality.

Spills: Two categories of memory accesses are caused by cache "spills" from one pipeline stage to the next (orange shades in Figure 3.7), and they represent about 10% of memory accesses on average. First, in bright orange, "W-R Spills" count memory writes from one stage that are read in the next stage, and thus represent producer-consumer relationships between stages. In darker orange, "R-R Spills" are reads from the same data in consecutive pipeline stages, indicating that the stages share input data. Inter-stage spills commonly result from GPU kernel stages that produce or consume data quantities in excess of cache capacity, another symptom of the GPU-kernel-synchronous structure of these applications. As a result of their pipeline organizations, most applications experience little reduction in cache spills when removing memory copies.

Eliminating inter-stage cache spills can result in significant performance improvement, especially when shifting work between CPU and GPU cores. In the `kmeans` case study, overlapping CPU and GPU execution in the heterogeneous processor eliminated the 9.5% of accesses that resulted from W-R spills, and subsequent cache hits increased CPU performance by 2.6×. CPU cores tend to be more sensitive to memory access latency than

GPU cores [54]. Shifting accesses from off-chip memory to cache hits can decrease CPU run time proportionally to the reduction in access latency [27].

Contention: When a pipeline stage accesses a large concurrent memory footprint, cache capacity contention can occur causing data to be evicted from cache before it can be reused. Further accesses must pull the data back from off-chip. Most frequently, these repeated contention accesses are reads ("R-R Contention"), which account for 38% of total accesses and upwards of 80% for many applications. Other contention accesses begin with a writeback of the data ("W-R Contention"), but the data is read again during the same pipeline stage, indicating that the writeback occurred before all uses of the data were complete. These accesses can account for up to 36% of a application's accesses. The substantial portion of contention accesses indicates that pipeline stage working sets often greatly exceed the available cache capacity.

Figure 3.7 also indicates significant potential performance gain by reducing cache contention. As denoted by '$*$' in the figure, many applications bump against off-chip memory bandwidth limitations during cache contentious pipeline stages. Most of these bandwidth-limited applications (e.g., Lonestar and Pannotia) also show significant cache contention memory accesses. Reducing the excess accesses is likely to proportionally reduce the memory bandwidth demand and run time.

3.6 Software Pipeline Key Implications and Related Work

3.6.1 Key Implications

The results in the last section indicate substantial opportunity to optimize core and cache utilization in heterogeneous processors. Potential research directions that will be applicable include producer-consumer parallelism, compute migration, and shared/cooperative cache management.

Software and hardware should provide efficient means to move producer and con-

sumer tasks into closer temporal proximity. To provide this proximity, prior work has proposed kernel fusion [151] and data pyramiding [139] for GPU-GPU producer-consumer relationships. However, these tend to be complicated program transformations, which can encounter resource limitations, such as GPU register and scratch memory capacity. These methods can still result in cache spills, as experienced by Parboil `stencil`, `cell`, `hotspot`, and `pathfind`.

More recently, CUDA and OpenCL have added methods to improve producer-to-consumer programmability. CUDA 5.0 introduced dynamic parallelism [103], which allows GPU code to dynamically launch consumer kernels. While this technique can provide programmability benefits for dynamic and irregular applications, prior work shows that kernel launch overheads can outweigh performance benefits [147]. OpenCL 2 added support for work queues, which allow producer tasks to queue generated work for other tasks to consume [69]. While queues may also provide producer-to-consumer programming flexibility, programmers will need to carefully pack data to maintain good GPU memory access coalescing for performance and efficiency.

Moving producer and consumer tasks into closer temporal proximity may raise new caching challenges. If producers generate data more quickly than consumers can pull cached data, spills will still occur. Software or hardware techniques could modulate the rate of data production and consumption to keep performance-sensitive data on chip. Further producer-consumer analysis techniques should improve identification of a task's live data and estimation of concurrent memory footprint to aid the programmer in placing data in available cache to avoid existing cache contention. Such techniques could prove very valuable for applications with irregular memory accesses (e.g., most Lonestar and Pannotia applications).

For heterogeneous processors, compute migration could unlock further fusion-like optimization opportunities. When CPU cores provide reasonable compute resources, migrating short-running GPU kernels to CPU cores could increase pipeline compute

overlap and increase effective cache capacity when CPU cores have private, non-inclusive cache. Such optimization may work in applications with multiple, varying complexity GPU kernels, as in Lonestar `dmr`, `mst`, or `sp`.

Finally, this chapter discusses ways that existing applications may be optimized for heterogeneous processors, and identifies constructs that may be directly used to develop new applications for heterogeneous processors. Forward-looking application development will likely adopt light-weight task handling and data dependency tracking early to ease the effort required to leverage available heterogeneous processor caching, since it is a primary benefit over discrete GPU systems.

3.6.2 Related Work

In addition to the many studies cited throughout this chapter, we identify a few classes of related work. First, while prior application characterizations focus mostly on the GPU-side resources and behaviors [21, 23, 138, 24], this study focuses on the application pipeline structures and interaction of all components in the systems. We are not aware of any prior studies that compare such results across many suites.

Prior studies propose methods to manage concurrency and optimize software pipeline structure. These mostly target multicore CPU and SMP systems [44, 143, 156] rather than systems containing GPUs, and few characterize potential optimizations to whole application pipeline structures.

A broad class of prior work aims to model and predict application run time when run on CPUs or GPUs. These works base their estimates on an application's required memory copies, compute operations, and memory accesses, as well as the specifications of the hardware (e.g., [19, 94, 11]). Similar models are used in large-scale distributed system runtimes to decide where to store memory to minimize data movement overhead [1, 32].

Finally, many prior studies look at GPU application optimization (e.g., [102, 125]) and run time comparisons [41], but we are not aware of any prior studies that quantify potential

optimizations focused around the elimination of memory copies in heterogeneous CPU-GPU processors. There are studies that describe results from mitigating redundant data and copies in other settings, such as OS data pipes to GPUs [124], and networking [157, 4].

3.7 Summary

This chapter compares GPU computing application optimization opportunities for discrete GPU systems and heterogeneous CPU-GPU processors. We modified applications to remove memory copies and run in cache-coherent heterogeneous processors. When comparing core utilization efficiency, the potential gains for improving core parallelism in either system type are similar. However, we expect programming models to adapt to make it easier to capture parallelism gains in heterogeneous processors.

Our memory access characterization indicates that the bulk-synchronous nature of GPU computing applications causes poor cache efficiency. Data often spills off-chip before it can be reused. The layout and use of data indicates that heterogeneous processors are likely to provide caching opportunities that can improve application performance beyond the capabilities of discrete GPU systems. To capture these opportunities will require flexible data/compute granularities and migration, and coordinated/intelligent caching.

In the following chapters, we explore these challenges and opportunities in more depth. We specifically develop applications suited to utilize the cores and cache in heterogeneous processors, and we study hardware techniques to support these applications.

4 DEVELOPING WORKLOADS AND SIMULATION INFRASTRUCTURE

This thesis aims to anticipate how emerging applications may stress the compute and memory resources in heterogeneous processors. The shared memory and cache coherence capabilities permit applications to communicate and synchronize data in new and unique ways. Unfortunately, prior to our research, the research community lacked both publicly-available applications that use these new capabilities, and the simulation infrastructure to test potential hardware configurations of heterogeneous processors. This chapter discusses the important aspects of our efforts to develop heterogeneous processor workloads and simulation.

First, to address the lack of heterogeneous processor applications, we develop new applications and modify existing GPU computing applications to test heterogeneous processor memory systems. The first part of this chapter expands on the characterizations in the last two chapters to describe our development process for targeting applications at heterogeneous processors. Chapter 2 identified application-level characteristics that indicate whether a computation will perform better on CPU or GPU cores. Chapter 3 broke down software pipelines of existing GPU computing applications to identify optimization strategies—migrating computation and overlapping work—when targeting heterogeneous processors. These guides suggest avenues that programmers will follow when writing new applications.

To select existing applications to modify and target at heterogeneous processors, we more deeply analyze application characteristics. Many applications contain separate portions that could run better on CPU or GPU cores, but often, these portions do not constitute significant portions of run time. As a result, migrating these computations without other algorithmic optimization will not substantially improve performance. On the other hand, many classes of applications, such as image processing and graph analytics, contain longer algorithmic stages each with a different amount of parallelism that may be targeted at the

different core types. We select applications with the most potential performance benefit from overlapping these CPU- and GPU-targeted stages.

As we modify applications to migrate stages between core types and to run stages concurrently, we must take care to preserve any communication ordering requirements between the stages. GPU computing applications often use kernel synchronization to implicitly synchronize producer-consumer communication. To run producer and consumer tasks concurrently, however, applications must introduce explicit and finer-grained synchronization. This chapter discusses the benefits and disadvantages of potential CPU and GPU synchronization mechanisms when coordinating CPU and GPU computation.

The second part of this chapter discusses our efforts to develop simulation infrastructure for evaluating heterogeneous processor architectures. Prior to our work, the research community lacked simulators capable of modeling shared memory and cache coherent heterogeneous processors. To fill this gap, we developed gem5-gpu [120]. Many of gem5-gpu's capabilities were published in IEEE Computer Architecture Letters in 2014, so this chapter discusses other important modeling details we developed for this thesis. Further, our newly-developed applications place unique demands on heterogeneous processor memory systems, such that prior simulation infrastructure can fail to accurately model CPU and GPU memory access interactions. We briefly describe these challenges and the simulation enhancements we made to ensure accurate memory system modeling.

4.1 Workloads for Heterogeneous Processors

The software pipeline analysis in Chapter 3 estimated that if applications could effectively use heterogeneous processor resources, they could see substantial performance improvement. However, when modifying existing applications in practice, programmers encounter numerous challenges to achieve these improvements. First, programmers must decide which software transformations could provide the largest performance benefits, but some

transformations, such as migrating work between core types, show small potential for performance gain. We describe potential optimization pitfalls, which indicate that programmers should focus attention on transforming relatively wide parallel compute stages and overlapping CPU and GPU work.

Second, existing GPU computing applications contain wide parallel compute stages, but programmers must decide which stages to overlap for largest benefit. For best utilization, the programmer must find successive pipeline stages to overlap that have similar run times when mapped to their subsets of cores and reasonable communication behaviors. Since GPUs are designed to process wider parallelism, programmers should map the widest parallel stages to GPU cores, while narrower parallel stages could be executed concurrently on CPU cores. We discuss the common software structures that result in this varying DLP across pipeline stages, and the classes of applications that commonly contain these structures. Finally, we describe one class of emerging applications, irregular algorithms, in more detail to highlight their interesting structures and opportunities in heterogeneous processors.

This section describes algorithm-level characteristics, software transformations, and synchronization for targeting heterogeneous processors. We summarize application characteristics for open-source benchmark suites in Table 4.1, and Appendix A includes a more complete classification of characteristics for each workload we study.

Table 4.1: Summary of software pipeline characteristics for open-source benchmark suites.

		Algorithm Organization			Implementation Details					
Suite	Num. Apps	Pipe P-C Comm.	Pipe Paral. -able	Vary-DLP P-C	Reg. P-C Comm.	Irreg. P-C Comm.	Topo.-driven	Data-driven	Fine Sync: Atomic	SW Work Queue
Lonestar	14	14	13	13	14	13	8	10	13	9
Pannotia	10	10	10	9	10	10	10	0	2	0
Parboil	12	9	9	7	9	3	12	0	3	1
Polybench	15	10	10	8	9	0	15	0	0	0
Rodinia	22	19	18	13	19	6	22	0	0	0
Totals	73	62	60	50	61	32	67	10	18	10
		85%	82%	68%	84%	44%	92%	14%	25%	14%

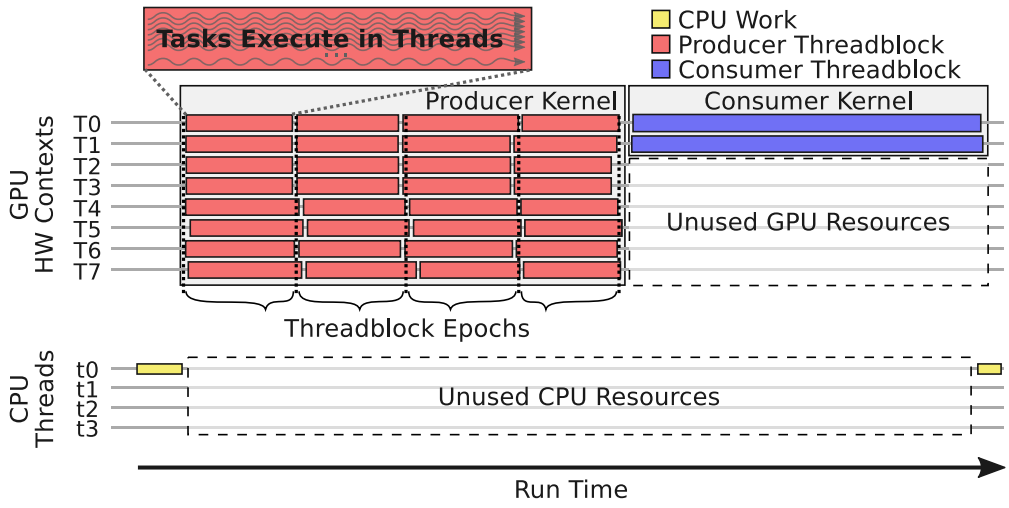


Figure 4.1: Utilization of processor resources during GPU computing application.

4.1.1 Specific Task Handling Characteristics

This subsection investigates pipeline stage task handling characteristics common in some existing GPU computing applications, and describes the practicalities that programmers face when optimizing these structures. Most applications contain producer-consumer communication that could be parallelized, but programmers must identify parallelization that can provide best performance. We identify two potential pitfalls—marked by coarse task granularity relative to the overall computation—that indicate when optimizations for heterogeneous processors can still leave cores underutilized. We conclude that programmers should instead optimize pipeline stages with relatively high DLP that can be structured as numerous task epochs relative to the stage's run time.

GPU applications contain hierarchical task structure to map computation to hardware. Figure 4.1 shows how stages of an application software pipeline map to CPU and GPU cores during run time. GPU kernels execute many tasks in threads grouped into threadblocks. For simplicity, this discussion assumes that each thread in the threadblock executes a single task. When a threadblock completes, another takes the hardware context until there are no threadblocks left in the kernel. For many graphics and early GPU computing applications, tasks in a single kernel perform roughly the same computation and have similar run times.

As a result, threadblocks often proceed in epochal sets that start and end around the same time.

Pitfall 1: Not Enough GPU Work: Many applications contain pipeline stages with limited parallelism. As depicted in Figure 4.1, the consumer kernel only contains enough work for two threadblocks, and the rest of the GPU contexts are idle during the consumer's run time. Frequently, applications contain these kernels when producer stages reduce the DLP for the consumer, or when a stage is inherently DLP-limited because it performs data or synchronization setup. Such DLP-limited stages might perform better on CPU cores. However, when applications are targeted at discrete GPUs, migrating data back-and-forth to the CPU memory can be costly. Prior work has observed a similar difficulty that applications need to have sufficient DLP to see performance benefit from GPUs [52].

When targeting heterogeneous processors that do not need to copy data between CPU and GPU memories, it can be straightforward to migrate work to CPU. These DLP-limited stages are common, but in most cases, they are not critical to optimize. In our application survey, we find that more than 15% of applications contain at least one GPU kernel that uses less than 50% of the GPU's contexts, even when the application processes large input sets. These stages are often very short, accounting for just 1–9% of application run time. Performance may improve by migrating work to the CPU, but the expected gains are small.

Pitfall 2: Few GPU Threadblock Epochs: Sometimes, the number of GPU threadblocks in a kernel is such that there are few threadblock epochs during the run time. Figure 4.1 depicts this situation: the producer kernel executes only four threadblock epochs. This "limited-epoch" structure often indicates the stage has limited DLP, and further coordination with CPU cores may still result in core underutilization.

First, tasks from a single pipeline stage could be executed on CPU cores in parallel with GPU activity, but may result in asymmetric progress. Task runtimes, such as OpenMP [84] and OpenACC [112], have added support to compile code to run on either CPU or GPU cores, and could be extended to execute a stage's tasks across all heterogeneous processor

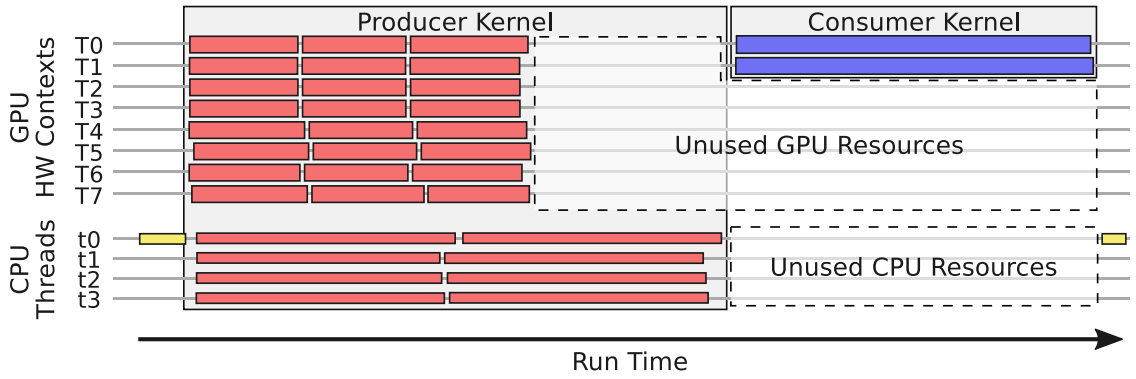


Figure 4.2: Progress asymmetry of different cores can cause cause limited-epoch pipeline stages to run longer when spreading tasks across all cores.

cores. We show that this parallelization scheme in Figure 4.2, which splits producer tasks out of GPU threadblocks and migrates them to CPU cores.

Though this structure could reduce CPU core underutilization, asymmetric task progress might still cause underutilization, especially for limited-epoch stages. Our results in Chapter 2 indicate that tasks completing the same computation on CPU cores can progress at rates up to 2–3× faster or slower than GPU cores. Due to relatively coarse threadblock granularity in limited-epoch kernels, threadblocks running on either core type might run long after the other cores have completed, causing unoccupied cores to be idle until the stage completes. Figure 4.2 highlights potential effects of this asymmetry, leaving GPU cores idle at the end of the producer stage. Task runtimes might evolve to adaptively and efficiently execute tasks from single pipeline stages across CPU and GPU cores, so programmers may not need to be involved in such scheduling. Instead, we focus on another form of component-overlap parallelism in which successive pipeline stages execute on the different core types.

Second, producer and consumer stages could be parallelized to execute concurrently on the separate core types, but this structure can also cause core underutilization due to limited-epoch stages. Generally, producer tasks generate outputs and synchronize these results to consumers implicitly using kernel boundaries (e.g., Figure 4.1 depicts back-to-back producer and consumer kernels). However, producer task outputs must be ready by

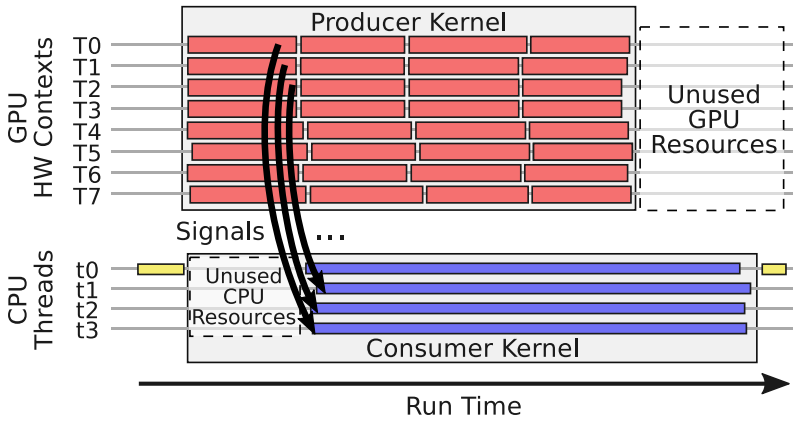


Figure 4.3: Concurrent producer-consumer activity requires signals from producer tasks to consumer tasks and can cut down underutilized resources, but can still result in underutilization during limited-epoch stages.

the time a task exits, so consumers could pull the data long before kernel boundaries if the producer tasks signal their completed work. Figure 4.3 shows this concurrent producer-consumer structure.

If producer and consumer stages execute concurrently, coarse-grained synchronization in limited-epoch producer stages can result in core underutilization. As shown in Figure 4.3, consumer tasks must wait through roughly 25% of the producer kernel's run time before consuming ready data, because there are only four producer task epochs. Further, consumers may still run long beyond the end of the producer, leaving the producer's cores idle. For producer pipeline stages with more task epochs relative to their run time, the potential underutilization can be much smaller.

Finally, we note an important practicality about GPU threadblock epochs: GPU threads can each execute multiple tasks to appear as though fewer threadblocks execute during a kernel, but tasks often still execute in epochs. Some applications execute multiple tasks per GPU thread in loops as it traverses large data structures. In this setting, a threadblock's run time is the sum of the sequential task times. The extreme case is called "persistent threads", in which threadblocks persist through the full kernel run time, and threads execute tasks by looping until all are complete [47]. Even when GPU threads execute multiple tasks per thread in a threadblock, tasks often still execute roughly in epochs. For the purposes

of investigating concurrent producer-consumer activity, we consider these task-to-thread mappings to be equivalent.

Although many applications have limited-epoch stages, they tend not to be performance critical. 15% of GPU computing applications contain stages with fewer than five task epochs. Another 14% of applications use persistent threads, but they usually have fine task granularity relative to kernel run time so we do not consider them limited-epoch. For most of these applications, these limited-epoch stages account for less than 16% of run time, and migrating them between core types is unlikely to improve run time significantly.

To summarize, pipeline stages that process narrow DLP or have relatively coarse task granularity are likely to cause core underutilization, even if parallelized to use all core resources in heterogeneous processors. Fortunately, such stages often account for a small portion of run time, suggesting programmers may find larger benefit from parallelizing application stages with greater DLP.

4.1.2 Algorithmic Structures with Varying DLP

The pitfalls described in the prior subsection guide our attention toward software pipeline stages that have specific DLP characteristics. Specifically, for successive pipeline stages, we can utilize producer-consumer parallelism when one stage has significant DLP that can benefit from GPU cores while the other stage has less DLP or more ILP for which CPU cores can perform well. Three common algorithmic structures can result in pipeline stages with these varying characteristics. When selecting applications to modify and target at heterogeneous processors, we look for these structures as indicators an application may be a good fit. As shown in the "Vary-DLP P-C" column of Table 4.1, roughly half of applications in open-source suites contain structures that cause varying DLP from one pipeline stage to the next. Below, we describe how we measure DLP variance associated with the structures.

Reduction-like Operations: We began our study of coordinated work in heterogeneous processors by deeply investigating reduction-like operations, because they can be flexibly

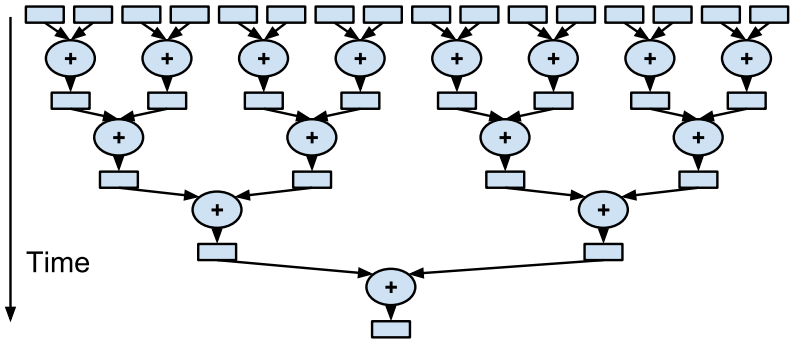


Figure 4.4: Example logarithmic reduction-sum on 16 data elements.

structured to test a spectra of DLP (and thus TLP) and ILP configurations. Reduction operations are constructed from a set of associative operators on pairs of data elements. The operator's associativity allows data elements to be combined sequentially or logarithmically to expose DLP (see Figure 4.4). Algorithm implementations can use combinations of linear and logarithmic stages as appropriate for best use of parallel hardware resources.

By simulating different reduction configurations across cores in heterogeneous processors, we glean a couple high-level insights about coordinating work. First, GPUs have in-cache execution units that can execute atomic operations, and these units should be considered as extra compute resources for certain reduction operations. Some reduction operations—such as addition/subtraction, multiplies, and logical and/or—can be completed using GPU in-cache atomic units, and to perform these reductions, GPU tasks send atomic operations to units in global memory caches. Current GPUs contain one to four pipelined atomic units per way of the cache, and atomic operations on a single data location (address) can proceed once every 2-8 cache cycles. In total, atomic units can contribute up to roughly 100 GFLOP/s to the total compute capabilities—comparable to 2-6 CPU or GPU cores.

Although atomic units offer significant extra compute resources, it is challenging to write applications that can fully utilize atomic resources. An application can easily send numerous atomic operations to a single data location from multiple threads, but these operations will be serialized at the atomic unit and proceed at low throughput. Instead,

threads can hierarchically accumulate results across separate data locations to use separate atomic units and improve throughput, but the programmer must manually structure the hierarchy and distribute operations across the separate data locations.

Second, reductions usually involve simple operations, which make them easy to structure to run efficiently on GPU cores or in atomic units. Unless a reduction operation involves more intricate computation, the portion of work that should be migrated to CPU cores is generally small. GPUs can effectively process a reduction's DLP until there are slightly fewer parallel operations than the GPU has execution units. At that point, the GPU's compute resources can become underutilized due to insufficient parallel work. For very large reductions of simple operations, the GPU should perform the majority of work.

Reduction-like operations are very common in many computing applications. Specific examples include vector and matrix operations such as multiplications and convolutions that are found in numerical methods and image processing applications. Other examples include histogram and cumulative distribution calculations that arise in statistical algorithms, and distance comparisons used in physics calculations. Most Polybench applications, and Rodinia `backprop`, `kmeans`, `lud`, `nn`, and `strmclstr` contain reductions.

Subset Data Processing: Some numerical methods and graph analytics applications contain pipeline stages that process subsets of data, and DLP narrows from one stage to the next. As an example, consider the Rodinia application, `lud`, which performs matrix LU decomposition. This algorithm iterates multiple times over three software stages that processes a matrix's perimeter, then its diagonal, and then its interior. The perimeter and diagonal stages process smaller portions of data than the interior stages, and the perimeter and diagonal computations could be processed concurrently with interior processing as long as data is appropriately synchronized. Analogously, graph analytics applications often progress such that each successive stage processes a subset of the data from the previous stage, and sets of stages could be executed concurrently. Such graph applications include Lonestar `bh` and `mst`, and Pannotia `color`.

Feature Extractions: Finally, we see varying DLP across pipeline stages of applications that involve data feature detection and extraction. A notable example is an application that uses image processing to detect edges of objects in an image. Early pipeline stages use image filters centered at each pixel of an image to detect color or contrast gradients, and they mark pixels that are candidates to be included as part of an object's edge. Further stages process candidate pixels to stitch them together and generate edges. The early image filtering stages can proceed in wide parallel with regular data access that is a good fit for GPUs. However, the later stages can witness more irregular data access and contain more sequential and low-DLP work, which can be a better fit for CPU cores.

A couple classes of applications perform feature extractions. Image processing pipelines are common in Rodinia applications, including `heartwall`, particle filtering (`pf_float` and `pf_naive`), and `srad`. Feature extractions are also common when detecting exceptional results in stream processing, such as Rodinia `strmclstr`, and signal processing, such as analog-to-digital radar and sonar [135].

Estimating DLP: Most applications contain varying DLP pipeline stages, and emerging applications show greater DLP variance. To estimate the DLP of separate pipeline stages, we collect two metrics when running an application in simulation. First, the thread organization often indicates task structure and DLP as long as each thread in a threadblock executes a single task. To collect thread organization, we record GPU kernel launch configurations, which include the number of threadblocks and the number of threads per threadblock. Second, since tasks may not always map directly to threads, we also collect memory footprint of each pipeline stage as a proxy for DLP. Typically, each task processes a small, independent chunk of a data structure, so the total data footprint size processed during a stage often correlates closely with DLP, and comparing footprint across pipeline stages can indicate whether DLP is varying.

We compare the number of threads and memory footprint across successive stages of each application to estimate how DLP changes. For some applications, these factors

remain constant (e.g., Rodinia `cell`), but more frequently, they change across pipeline stages by up to 10,000× or more. We classify applications as having varying DLP when successive per-stage thread counts or footprints change by more than 50%, and inspection of the code confirms the changing DLP. We find that 68% of open-source applications have varying DLP across pipeline stages. In Section 4.1.3, we describe the DLP differences for applications with regular and irregular task structures. In more regular applications—such as most in Rodinia, Parboil, and Polybench—28 of the 49 (57%) contain varying DLP across software pipeline stages. In contrast, varying DLP is very common among the irregular applications in Lonestar and Pannotia suites (22 of the 24 applications, 92%), and we describe why in more detail below.

4.1.3 Regular and Irregular Applications

Finally, after identifying applications with software structures that are likely to benefit from execution on heterogeneous processors, programmers must work through practical details of modifying the organization of tasks and data structures to better map to heterogeneous processors. GPU computing applications typically organize tasks around their DLP, and the regularity of the DLP often dramatically affects the way tasks can be organized. To give the reader a flavor for these practicalities, we discuss task organizations for regular and irregular applications. Compared to existing regular applications, future-looking irregular applications show increased DLP variance throughout execution, indicating they may be even better suited for processors with heterogeneous compute capabilities. We summarize the number of applications with regular and irregular structures for communicating producer-consumer data in Table 4.1 ("Reg. P-C Comm." and "Irreg. P-C Comm." columns, respectively).

Burtscher et al. define regular and irregular applications [21]. A computation is considered to be regular if the data can be regularly structured, as in arrays or matrices, and tasks process the data in regular or statically predictable ways. Example applications include

matrix manipulations and image filtering. Programmers can often easily reason about the way these applications access memory. In irregular applications, on the other hand, tasks operate on irregular data structures, such as graphs or priority queues, and tasks may need to be spawned dynamically based on the data. The dynamic, data-dependent nature of these computations often make it difficult to reason about their memory usage characteristics. Irregular computations are often used in physics simulations, and for graph analytics and manipulation.

Compared to regular applications, irregular applications frequently involve deeper software pipelines with greater DLP variance as a result of how they must manage tasks to operate on their irregular data. As an example, consider a common graph traversal algorithm that moves through frontiers, like Lonestar `bfs_wla` and `sssp_wln`. In each software stage, a task processes edges from a node in set called the frontier, and it updates the frontier by removing its source node and adding destination nodes along each edge. The amount of work available in each stage depends on the frontier size, which is an intrinsic characteristic of the graph's structure. Graphs with high degree will result in pipeline stages with high DLP, and a graph's diameter dictates the number of pipeline stages required to traverse it.

Topology-driven and Data-driven Irregular Applications: Finally, irregular applications often require intricate software transformations to perform well on GPUs, but the resulting task management structures are very useful in heterogeneous processors. Nasre et al. describe processes for transforming irregular applications from topology-driven to data-driven task handling [101]. In each stage of computation, topology-driven applications spawn a task for every data element in a structure, such as each node or edge in a graph. Tasks first check whether there is work to do at their respective element. If so, the task completes the work, but if not, the task exits immediately, completing no useful work. When small subsets of the data must be processed in a stage, few tasks do useful work, so a topology-driven approach can be very wasteful of compute resources.

An alternative approach is called "data-driven" irregular computation [101], because it accounts for whether a data element must be processed in the next computational stage. When a task updates a data element during one stage, it can often check whether its activity will trigger further work on that data in the following stage. If so, it can signal this requirement by creating a work item (request) in a queue. The following stage of computation can just spawn tasks for each work item in the queue rather than for all data elements.

If a data-driven approach is appropriate for an application, it can yield significant benefits for running in heterogeneous processors. First, work queuing narrows the total amount of computation, directly reducing run time to just the useful work in each stage. This narrowing also has derivative benefit on caching; since fewer tasks execute in each stage, they also access less data, which can reduce the pressure on cache capacity and reduce conflict misses. Applications tested by Nasre et al. and in our work have witnessed performance gains of 1.1× to over 100×.

Finally, using work queues for task management is very useful for structuring efficient computation in heterogeneous processors. Work queues act as concentration points for computation control, because threads can pull and perform work items simply by knowing the location of the queue in memory. As described in Section 4.2, this queue structure is very powerful for migrating work between CPU and GPU cores that share memory.

4.2 Coordinating CPU and GPU Computation

Section 4.1 describes pipeline stage characteristics that indicate an application may perform well when targeted at heterogeneous processors. We look for producer-consumer pipeline stage pairs, for which producers have high-DLP, and consumers complete lower DLP or higher ILP computation. We aim to fully utilize all heterogeneous processor compute resources by executing such pipeline stage pairs concurrently, with producer tasks on the

GPU and consumer tasks on CPU cores.

Unfortunately, as we will show in Chapter 6, naively executing work on CPU and GPU cores concurrently often causes excessive contention for cache and memory resources. The characterization in Chapter 2 showed that differences between CPU and GPU microarchitectures can cause significant differences in MLP and the temporal characteristics of memory accesses. GPU burst memory behavior often causes significant contention with CPU cache and memory accesses. If programmers aim to efficiently coordinate CPU and GPU concurrent work, the cores will need to communicate and synchronize data without causing excessive contention for cache and memory resources.

This section briefly documents our experience with and assessment of synchronization options for coordinating work between CPU and GPU cores. Signal variables and atomic data updates are useful mechanisms for synchronizing the common producer-consumer behaviors in GPU computing workloads. When the GPU participates in such synchronization, however, the synchronization granularity must usually be coarser than per-GPU-thread, which causes excessive cache accesses and contention.

4.2.1 Synchronization Primitives in Heterogeneous Processors

Existing multicore CPUs and discrete GPUs each offer a number of synchronization primitives for coordinating work among their threads. These same primitives can be valuable when coordinating work between CPU and GPU cores in a heterogeneous processor. Here, we describe the benefits and disadvantages of each.

We preface this discussion with a memory consistency note. CPU and GPU cores provide different memory consistency models, so when synchronizing across core types, each core must have mechanisms to ensure memory ordering guarantees on shared variables. Our tests simulate processors with scoped memory consistency closely resembling the OpenCL v2 specification [69]. To correctly synchronize between CPU and GPU threads, each must execute appropriate fence or atomic operations that make data visible in the

chip-wide coherence domain (called "global level" in CUDA/PTX or "device-scope" in OpenCL).

Signal Variables: Signal or flag variables are commonly used to indicate when data becomes available for consumption, so they are a natural fit for concurrent producer-consumer activity. After writing output data, a thread executes a fence to ensure the output is visible to other threads, and can then complete a standard or atomic write to the signal variable. Commonly, GPU producer pipeline stages generate outputs in regular data arrays, and threads can signal in a separate associated array or in work queues, as described in Chapter 6.

Signal variables have desirable performance properties. Variable signaling tends to be low latency, mostly for the fence between output data and the variable (e.g., 2–10 cycles from CPUs, 150–250 cycles from GPUs). Signals can also be very high throughput: CPU threads can vectorize parallel signal writes and GPU threads can coalesce their writes.

Naive algorithm implementations with signal variables often have high memory space and performance overheads. Since tasks often process one or a few data elements, it is often straightforward to write code that uses signal variables to indicate when each task is complete or after each data element has been modified. Unfortunately, this fine-grained signaling often requires large memory space overhead of one signal variable per task or data element, making the signals size proportional to the original data set size. This storage can cause cache capacity contention and high signal read overhead, reducing performance. The next subsection discusses ways to change the granularity of signaling to mitigate storage overheads.

Atomic Data Updates: In some producer-consumer pipeline pairs, consumer computation can be completed using atomic operations, such as increments/adds or setting extrema with min or max. Unfortunately, most consumer pipeline stages perform non-trivial work that is not amenable to atomic operations. Reduction-like operations are often a good target for atomics, but producer-consumer pairs that perform subset data processing or

feature extractions rarely need to accumulate results as in reductions. In Chapter 6, we test application implementations that use atomic operations when possible.

Locks: Applications could also use fine-grained locking, though the technique is uncommon in GPU computing applications. CPU and GPU threads can use atomic operations to set and reset lock variables that indicate exclusive access to shared data. Similar to signal variables, when GPU applications use locks (e.g., Lonestar `bh` or `mst`), they tend to store locks in arrays that are associated with regularly-structured data that requires locking.

Coordinating CPU and GPU work using locks can result in unfair data access due to differences in CPU and GPU atomics. GPU threads can use in-cache atomics (e.g., at the L2 cache) to set and unset lock variables, and these operations can proceed with high throughput. CPU atomic operations, on the other hand, are often implemented by pulling the cache line into the CPU L1 data cache, setting a bit that forces the line to remain in exclusive state there, performing the atomic computation in the CPU core, and then writing the line. Even without cache contention, CPU atomics tend to be longer latency and lower throughput than atomics in current GPUs due to the latency for moving data from cache to the CPU core and back. As a result, if CPU and GPU threads contend on locks, CPU threads can experience starvation for access to the data.

Spin-wait Loops: When using signal or lock variables, threads must have a mechanism to wait for the variable, and spin-wait loops are a first-cut approach. CPU and GPU threads can spin-wait using similar code (e.g., Figure 4.5), which checks the variable and executes a memory fence in a loop until the variable is set. When signaling between CPU and GPU cores, threads must be careful to access data that is visible to all threads synchronizing on the variables. Thus, GPU threads must use device-scope fences to access these variables and ensure they do not read from stale cached data. Further, when GPU threads spin-wait, the programmer must take care that all threads in each warp proceed through the loop together to avoid deadlocks.

We commonly use spin-wait loops in our coordinated work applications, because our

```
1  void signal_wait(bool *_signal_variable) {
2      while (!(*_signal_variable)) {
3          // Perform memory fence
4          __memory_fence();
5      }
6  }
```

Figure 4.5: Spin-wait pseudocode. Threads wait until a signal variable in memory gets set. Code is similar on x86 CPU cores as on NVIDIA GPU cores.

testing shows that CPU and GPU spin-wait loops provide good performance. Typically, the latency between when a producer writes a variable and when a consumer detects it is only 5–40 cache cycles longer than the latency to transfer the variable between writer and reader caches.

Thread Yield/Sleep: When many threads spin-wait in loops, they can contend with other threads for execution and load-store unit slots. We did not encounter instances in which spin-wait loops caused significant contention. However, we considered two thread sleep mechanisms that can pause a thread's execution while it waits for a signal variable: thread yield and microsleep.

First, the CPU thread can yield its context to other threads (e.g., x86 `hlt` instruction), but this process can be a very long latency operation. The operating system can context switch out the executing thread and when it later receives an interrupt to wake up the thread, it switches it into the next available thread context. Context switches and waiting for an available thread context typically incur latencies upward of 10µs, which is incompatible with our desire for low latency coordinated work.

Though we do not use it frequently, a useful CPU mechanism pauses a thread's execution while waiting without yielding the context, called thread microsleep. CPU ISAs include instructions to monitor a memory address for activity and for a thread to pause execution until activity is detected (e.g., x86 `monitor` and `mwait` instructions, respectively). This mechanism makes spin-waiting more energy efficient, because threads no longer need to poll for changes to a memory location. Microsleep is a good candidate for CPU-GPU coordinated work signal waiting, because compared to spin-waiting, it can eagerly detect

memory access activity, and restarting the waiting thread adds just 8–20 cycles latency.

Prior work investigates GPU thread sleep [140] and threadblock draining and preemption [141]. Threadblock draining is when the GPU core allows a threadblock to complete execution, but reclaims the threadblock context for another kernel rather than scheduling another threadblock from the same kernel. GPU threadblock schedulers can also be designed to preempt running threadblocks. In this case, the threadblock must save its GPU core state (e.g., registers and scratch memory) to a location that can be re-read later (e.g., global memory), and then, the threadblock releases the context. Our Q-cache architecture in Chapter 6 makes use of threadblock draining, but not for the purposes of synchronization. Further, our coordinated work applications execute producer pipeline stages on the GPU, and the baseline organization with fine-grained signaling does not require the GPU to wait (spin or sleep).

4.2.2 GPU Synchronization Granularity

When GPU cores participate in coordinated computation with CPU cores, new synchronization challenges arise as a result of the GPU's wide multithreading, fine-grained tasks, and memory access coalescing. As GPU producer tasks near completion, they can collectively generate outputs in bursts of hundreds or thousands of data writes within tens of cycles. If each task also signals that its outputs are ready, they can double the total producer writes and consumer reads. Further, coalesced memory writes from GPU warps can each set up to 32 signal variables and can proceed at a rate of one per cache cycle. This write rate can cause extreme cache contention.

To enable efficient producer-consumer communication, we will need to reduce the GPU's synchronization granularity. Consumer tasks need to be able to keep up with high throughput producer writes, but fine-grained signaling can cause very high consumer read overhead. Prior work [71, 25] and our tests reveal 2–5× performance overheads, caused by the extra lock or signal variable reads, and cache contention misses due to the extra signal

```
1  __global__ void producer(Data *input, Data *output, Signal *signals){
2      // All threads in block perform producer computation
3      process(input, threadIdx.x, output);
4      // All threads fence to ensure data ordering
5      __memory_fence();
6      // All threads wait until all threadblock results are written
7      __threadblock_barrier();
8      // 0th thread signals threadblock's results are ready
9      if (threadIdx.x == 0)
10         signal(&signals[blockIdx.x]);
11 }
```

Figure 4.6: GPU pseudocode demonstrating the use of barriers for per-threadblock data-ready signals.

storage. This problem is common across all coordinated work applications we consider.

To address this challenge, we test and use thread barrier techniques to narrow GPU synchronization from per-task or per-data-element to per-threadblock. Code for this process is shown in Figure 4.6. Specifically, after tasks in a threadblock process their input and generate outputs, they execute a fence, but then perform a threadblock barrier to wait for all other threads in the threadblock to write their results. After the barrier, a single task can signal that all of the threadblock's results are ready.

We test two barrier implementations for this coarser granularity signaling: GPU hardware barriers, and barriers constructed from atomic operations. GPUs offer hardware barriers that pause progress of threads at the barrier until all other threads in the threadblock have arrived at the barrier. Hardware barriers can be very energy efficient and low latency. However, we later show that despite their efficiency, current barrier implementations have inefficiencies, which can cause high performance overheads as many threads must wait for other lagging threads. This overhead motivates our following proposal to improve hardware barriers, Transparent Fuzzy Barriers, in Chapter 5.

We use GPU hardware barriers for our coordinated work applications, because they provide the best performance, but we also tested barriers constructed from atomics. In an atomic barrier, threads atomically increment a counter variable when they arrive at the barrier. Then, they wait until other threads increment the counter to be equal to the total number of participating threads before they continue execution. Atomic barriers allow

more or fewer threads to participate than hardware barriers, which pause execution of threads in a threadblock. This difference allows atomic barriers to provide even coarser grained signaling. However, we do not find a need for this flexibility.

Threadblock barriers reduce the synchronization granularity and eliminate more than 96% of performance overhead caused by per-thread signaling. Threadblocks often contain 128–1024 threads, so using threadblock barrier signaling reduces the number of signal variables and storage by more than 100×. Further, commonly, GPU threads heavily contend for cache access, so the extra signal variable accesses can cause more than 2× performance overhead. The improvement from coarser grained signaling makes it much easier for consumer tasks to keep up with producers.

These synchronization advances greatly improve concurrent producer-consumer performance and allow the programmer or system to focus attention on making efficient use of on-chip caches in heterogeneous processors. The potential benefits of improved cache communication motivate our proposal for Q-cache in Chapter 6. Q-cache is a cache management technique to avoid spilling producer-to-consumer queued data from caches.

4.3 gem5-gpu: A Heterogeneous CPU-GPU Simulator

Our workload characterization and heterogeneous processor application development have identified opportunities for hardware to better support coordinated work between CPU and GPU cores. Unfortunately, prior to our work, the research community lacked simulation infrastructure to test applications that leverage the shared memory and cache coherence capabilities in heterogeneous processors. To address this challenge, our research group has developed the gem5-gpu simulator through its formative stages, and we use gem5-gpu to model heterogeneous CPU-GPU processors for characterizations in Chapters 2 and 3. We document gem5-gpu's broad capabilities in IEEE Computer Architecture Letters [120].

The challenges of modeling heterogeneous processors are many, and this section de-

scribes the most important modeling details we developed specifically to test proposals in this thesis. Heterogeneous processors allow CPU and GPU memory accesses to interact in the cache hierarchy, and prior simulation infrastructure had not been adapted to model their differing access characteristics. Our research focuses on efficient communication and synchronization between CPU and GPU cores, so these cache access interactions need to be modeled with sufficient fidelity to anticipate real cache hierarchy bottlenecks. The important microarchitectural components we developed or enhanced include GPU memory access coalescing and queuing, GPU cache coherence, and buffering and fairness in memory access paths.

4.3.1 Table Stakes Simulator Development

gem5-gpu integrates the CPU models and Ruby memory hierarchies of gem5 [18] with the GPU core model from GPGPU-Sim [10]. To accurately model state-of-the-art GPU cores accessing global memory through the Ruby memory hierarchy, we implemented four major "table stakes" simulated system features: GPU load-store queuing to global memory, atomic memory accesses, scoped memory fences and barriers, and GPU L1 caches that do not require read-for-ownership.

When we integrated gem5 and GPGPU-Sim models, GPGPU-Sim modeled global memory accesses with approximately timing-correct global memory accesses and functional data access. However, the Ruby memory model requires both data and timing correctness. To resolve these modeling differences, we developed a cycle-accurate GPU load-store queue (LSQ). This LSQ accepts memory requests, including addresses and data as appropriate, from the separate threads (lanes) of a warp executing on a GPU core in GPGPU-Sim. It models four stages. First, it coalesces requests into accesses to a minimal set of cache lines, and then models stages that queue and issue the associated memory accesses to the connected Ruby cache hierarchy. Finally, when a memory access completes, it decoalesces data to be returned to requesting threads.

By moving global memory accesses to the Ruby memory model, we were unable to use GPGPU-Sim's modeling of atomic memory operations, so we augmented the GPU LSQ and Ruby memory model to perform GPU atomic operations. Similar to standard accesses, when GPU threads issue atomic requests, they send addresses and data to the LSQ, but they also send a flag indicating the type of atomic operation to be performed. The LSQ coalesces and packs atomic operations differently than standard accesses, because they will be performed by Ruby caches. Ruby models these operations by functionally executing the atomic functions on the data after the access attains write permissions at the appropriate cache.

Next, the relaxed memory consistency model used with NVIDIA GPUs required that we correctly model memory request orderings with respect to fences and barriers. The gem5-gpu LSQ handles memory requests in the order they were received, but Ruby caches are allowed to reorder accesses, which can cause data to appear reordered when memory accesses complete. To track fences, the GPU core sends fences (implicitly included with barriers) to the LSQ, which places them in the warp's memory request queue. To enforce release consistency, when the fence reaches the head of the instruction queue, it blocks until all prior memory accesses from the warp have completed. If it needs to enforce acquire consistency (e.g., global-scope fence), the fence's acquire operation is forwarded to the Ruby caches, which flush any stale data out to global scope.

Fourth, since Ruby had been previously used to model CPU cache hierarchies, it assumed that all caches would be coherent and implement read-for-ownership (RFO) coherence protocols. To enforce these assumptions, Ruby's cache hierarchy input port—the "Sequencer"—would block multiple outstanding memory accesses to a single cache line, requiring the sending core to buffer accesses until its cache had ownership of the line. Unfortunately, cache hierarchies in state-of-the-art GPUs permit stale (incoherent) data to be read from some caches, and memory stores to be buffered without obtaining ownership through some cache hierarchy levels. We modified the Ruby Sequencer to permit multiple

outstanding memory accesses per cache line, and we developed a GPU L1 cache controller that accurately models NVIDIA's no-RFO cache design.

Finally, although the GPU cores in current hardware functionally perform similar request coalescing and cache accesses, different microarchitectures often have very different access latencies and throughput. To accurately model different microarchitectures, we parameterized the LSQ in gem5-gpu to model flexible LSQ pipeline latencies, atomic memory access packing widths, and fence operation latencies.

4.3.2 Modifying GPGPU-Sim to Model NVIDIA Maxwell

As gem5-gpu matured, GPGPU-Sim's GPU core microarchitectural model—based on NVIDIA Fermi—became stale compared to state-of-the-art GPU models, such as NVIDIA Kepler and Maxwell. To ensure that our tests reflect potential advances on the state-of-the-art, we modified GPGPU-Sim's core model to more accurately simulate NVIDIA Maxwell cores.

First, in testing, we found that compared to NVIDIA Fermi cores, Maxwell cores show reduced stalls due to instruction fetches. Consistent with Fermi [142], GPGPU-Sim GPU cores send fetch requests lazily when a warp becomes eligible to issue but its instruction buffers are empty. This lazy fetch behavior causes high overheads when few warps are ready to issue, and instruction execution is unable to hide instruction fetch latency. To improve Maxwell modeling, we adjusted GPGPU-Sim to send fetch requests more eagerly, when a warp issues the last instruction from its instruction buffer. This improves performance consistent with Maxwell, and more recent prior work has confirmed the benefit of eager instruction fetch [88].

We also adjust GPGPU-Sim's configurations to more accurately model Maxwell compute operation throughput. Maxwell cores, called the Maxwell Streaming Multiprocessors or "SMMs", contain two sets of components that comprise a Fermi core ("SM"). Specifically, Maxwell cores contain two pairs of warp schedulers, two scratchpad memories, and two

sets of execution pipelines. To model a Maxwell core, we adjust GPGPU-Sim's configuration to include two cores per cluster, so each cluster acts as a single SMM connected to the cache hierarchy. Further, we add minor fixes and configuration adjustments to GPGPU-Sim's scratch memory and execution unit pipelines to better model Maxwell's throughput and latencies. Finally, we use parameters of the GPU global LSQs, described above, to accurately model global memory access, atomics, and fence/barrier latencies.

4.4 Improving Ruby Cache Hierarchy Modeling

Prior to our work, gem5 only modeled CPU cores interacting through the cache hierarchy. CPU memory accesses tend to be regularly distributed through time with little contention for buffering and bandwidth. To accurately model CPU memory access, gem5 often just needed reasonable cache and memory access latencies, but did not need to model accurate cache buffering and throughput. In contrast to CPUs, GPU memory accesses place heavy stress on memory hierarchy buffering and cause contention. To handle these stresses, we needed to improve gem5's Ruby memory model. Specifically, we improve Ruby's ability to model finite buffering throughout memory hierarchies, and we fix memory access arbitration in gem5's Ruby interconnect model that caused unfair scheduling.

4.4.1 Finite Memory Access Buffering: "Nature Abhors a Vacuum"

By default, the Ruby memory model does not impose buffer capacity limitations in its cache hierarchies, and this unlimited buffering causes unrealistic queuing and latency under heavy memory access load. Figure 4.7 depicts an example memory hierarchy in which unlimited buffering can cause problems. Consider the case where many GPU cores all issue numerous read requests that miss in cache requiring them all to access off-chip memory. With unlimited buffering throughout the cache hierarchy, these accesses will proceed nearly uncontended to the point that bandwidth bottlenecks. In the diagram, the

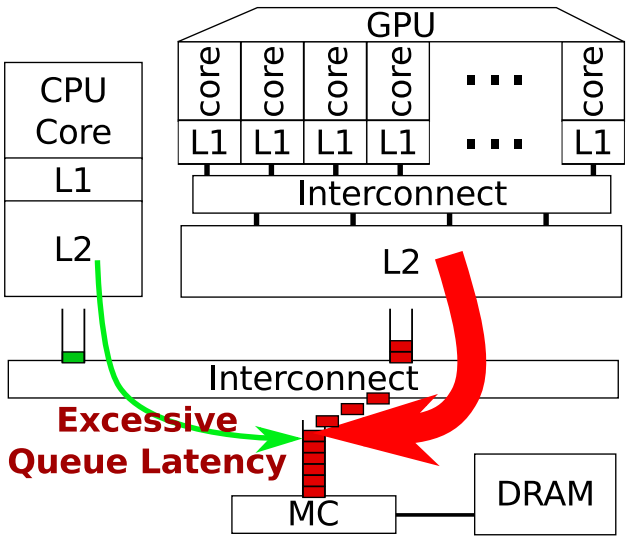


Figure 4.7: Example memory hierarchy in which unlimited buffering causes unrealistic queuing and latency for CPU memory accesses.

DRAM controller bottlenecks the accesses due to its limited channel bandwidth to the DIMMs, so accesses bloat the DRAM controller's input queue.

The bloated DRAM input queue causes unrealistic CPU memory access latency. Suppose a CPU memory access traverses the interconnect and enters the the DRAM input queue behind the large group of GPU accesses. In the unlimited buffering scenario, each GPU core may have tens of accesses waiting in the queue, so the CPU access could be behind more than a thousand prior requests, potentially causing the CPU access to wait tens of thousands of cycles. Real system cache hierarchies, on the other hand, commonly limit CPU access buffering time to hundreds of cycles, even under heavy load. We find that the excessive latencies of unbounded buffering caused unrealistic simulated system performance, sometimes degrading performance by 3–8×.

To ensure that our simulation model provides realistic buffering latencies, we enhanced Ruby's cache hierarchy to correctly model finite buffering. These enhancements involved fixing Ruby's cache controller and interconnect queues to correctly enforce buffer capacity, and augmenting simulation configuration scripts to set buffer capacities throughout the caches and interconnects. Further, we modified gem5's DRAM memory controller models

to add finite request and response queuing. After making these changes, we compare memory access latencies and bandwidths to existing CPU and GPU hardware to validate that they behave realistically under heavy memory access load and fix performance degradation.

4.4.2 Fair Network Arbitration

Finally, as a consequence of adding finite memory access buffering in Ruby, cache hierarchy components can cause back pressure to other components when buffers fill. When buffers experience back pressure through a routing point, such as an on-chip interconnect switch, memory accesses must be fairly prioritized to ensure predictable latencies and fair progress among cores. Unfortunately, gem5's Ruby and Garnet interconnect models use a static round-robin arbitration scheme that has been shown to be unfair in prior work [73].

To address this arbitration issue, we implemented the batching assured access arbiter described by Vernon et al. [144] in Ruby's interconnect router. As accesses that are destined for a common output port get buffered in the router's input queues, the router creates a batch of accesses that must all traverse the router before the next batch can be formed. Each batch contains up to one access from each input queue if the queue has at least one access waiting when a batch is started. Batches ensure fairness; no input port is allowed to issue twice before other ready input ports issue once. Further, to ensure that separate batches do not systematically prioritize particular input ports, each new batch randomly selects one of the ready input ports to be the leader of the round-robin ordering within the batch. This arbitration scheme eliminated GPU core access unfairness that caused up to $3\times$ progress asymmetry.

4.5 Workload and Simulation Summary

Overall, our infrastructure development efforts for this thesis comprise applications and simulation infrastructure targeted at exercising the unique memory and caching capabilities of emerging heterogeneous processors. Applications that have the most opportunity for optimization contain varying DLP pipeline stages that can be transformed to overlap CPU and GPU computation and fully utilize the available resources. When coordinating CPU and GPU work, programmers must use GPU coarse granularity synchronization to avoid extreme synchronization overheads. The need for coarse synchronization motivates our proposal to improve GPU hardware barriers in Chapter 5. Further, when CPU and GPU cores coordinate work, they have significant opportunity to better leverage available on-chip caches. In Chapter 6, we propose a technique to better support producer-consumer communication in cache.

Finally, in order to deeply study cache and memory access behaviors during CPU and GPU coordinated work, we need simulation infrastructure that provides realistic memory access latencies and bandwidths. Prior simulation infrastructure comes up short of these fidelity requirements. To address this need, we developed the gem5-gpu simulator, which includes accurate GPU simulated memory system components, and enhanced memory hierarchy buffering and access arbitration modeling compared to prior simulators.

5 TRANSPARENT FUZZY BARRIERS FOR GPUS

While developing coordinated work applications, we find that sharing cache and synchronizing data among CPU and GPU cores can cause high overheads. Bursty GPU memory access can cause significant cache contention. This contention can interfere with synchronization between CPU and GPU cores, especially when using fine-grained signaling. Further, when threads generate results to be communicated to other cores, they need to use fences with appropriate scope that ensure access ordering visible to the other cores. These fences can add latency and perturb cache hit rates compared to fences that synchronize data visible to a single core.

As Section 4.2.2 describes, applications can reduce performance overheads by using coarser-grained GPU synchronization. GPU threadblocks can execute barriers to ensure correct memory ordering of each thread's results and signal their results collectively. Although coarser-grained synchronization improves performance of coordinated work applications, state-of-the-art GPUs have conservative hardware barrier implementations that can cause high synchronization overhead. In a survey of GPU computing applications, we find that barriers often add more than 20% to application run time. Further, the majority of applications use barriers, and they do so in diverse ways.

This chapter evaluates existing techniques to reduce barrier overhead, but shows that they run into programmability and performance limitations. To address these challenges, we propose Transparent Fuzzy Barriers for GPUs, a hardware technique that allows threads to safely execute further instructions while waiting at a barrier to improve performance without requiring programmer involvement or changing existing GPU architectures.

5.1 Introduction

GPUs depend on significant memory-level parallelism (MLP) to tolerate their long memory access latencies, but MLP can be limited in the presence of data synchronization. Graphics

and GPU computing workloads both frequently share data among threads, and existing synchronization primitives can cause significant performance overhead. In state-of-the-art GPUs, fences and barriers can limit in-flight instructions ("occupancy") and MLP. For existing GPU computing applications, fences and barriers frequently cause more than 20% run time overhead and up to 65%.

On top of existing synchronization overhead, emerging systems are likely to see increased overhead. Compared to discrete GPU memory hierarchies, new heterogeneous processor memory consistency and cache coherence models often add cache operation latency to enforce consistency or coherence. Similarly, large scale systems integrating many GPUs aim to offer increasing shared memory capacity, but data synchronization across physically separate memories will come with higher latencies.

This chapter shows that barriers in current GPUs cause four sources of lost occupancy that can harm performance. The biggest source of occupancy loss is load imbalance, when early warps to a barrier must block to wait for lagging warps. Prior warp scheduling proposals [7, 88] target barrier overhead, but only reduce issue slot contention between threadblocks. They fail to address load imbalance, fence latency, and after a barrier completes, contention for execution units. Warp scheduling only recovers 11% of occupancy loss for 1.01× average speedup.

Prior work proposes that multicore CPU systems use fuzzy barriers [48, 49], which allow programmers to specify work that threads can perform while waiting at a barrier—during the "barrier region". GPUs have many threads and should expose numerous instructions to execute during fuzzy barriers to recover occupancy, but programming fuzzy barriers is complicated due to GPU relaxed consistency and multiple memory spaces and scopes. Compilers could specify barrier regions, but unfortunately, they must end fuzzy barrier regions before complicated control flow. As a result, they only hide 16% of lost occupancy for 1.015× speedup.

To improve on fuzzy barriers, we propose Transparent Fuzzy Barriers (TFB), a hardware

technique that can dynamically execute instructions during barriers to increase execution unit occupancy. On average, TFBs hide more than 50% of lost occupancy during barriers for a 1.061× average speedup and up to 1.14×. TFBs do not require architectural or software changes; they exploit side-effect visibility requirements of architected barriers (e.g., in OpenCL 2 [69]), which permit hardware to execute instructions while threads wait at a barrier.

This chapter makes the following contributions:

1. Shows that in GPU computing workloads, current hardware barriers cause significant occupancy loss, largely due to load imbalance.

2. Analyzes GPU and barrier architectural specifications to show how hardware can reorder instructions during barriers while meeting side-effect visibility requirements.

3. Proposes three Transparent Fuzzy Barrier variants and compares them to warp scheduling and fuzzy barriers to show their capabilities to restore occupancy.

Overall, we conclude that TFBs hide barrier overhead better than warp scheduling and fuzzy barrier techniques, and they only require small logic changes or 1kB added storage per core. TFBs expose more instructions during barriers to the warp schedulers, which can optionally issue them to maintain occupancy and warp progress divergence that tends to improve parallelism. Performance of warp scheduling and TFBs compose with further opportunity to co-optimize the two.

The rest of this chapter is organized as follows. Section 5.2 motivates the need for efficient barriers by breaking down their substantial overhead. Section 5.3 shows that the prior fuzzy barrier technique, when implemented in GPUs using compiler instruction reordering, often encounters static instruction ordering requirements that limit potential performance gains. Section 5.4 identifies the instruction ordering that the OpenCL architectural specification allows hardware to perform near barriers. Section 5.5 describes existing GPU fence and barrier microarchitectures as lead-in to our three TFB implementation

proposals in Section 5.6. We evaluate and discuss TFBs in Section 5.7, and observe related work in Section 5.8.

5.2 Hardware Barrier Overhead

Existing hardware fences and barriers can impose substantial run time overhead. We introduce a method for estimating fence and barrier overhead in hardware, and show that NVIDIA Maxwell hardware often experiences more than 20% run time overhead. Barrier overhead is caused by the reduction of in-flight instructions ("occupancy") as warps wait for others at a barrier, and we break down sources of occupancy loss during barriers. Prior warp scheduling proposals cannot significantly improve occupancy, since they do not reduce the major source of overhead: load imbalance.

5.2.1 Barrier Use is Diverse and Frequent

GPU computing workloads heavily rely on barriers. Table 5.1 lists all applications containing barriers from the Lonestar, Pannotia, Parboil, and Rodinia application suites. More than half of the 54 applications in these suites execute barriers, indicating that they are common in GPU computing.

These applications use barriers for very diverse communication behaviors. Specifically, they use barriers to synchronize access to scratch and global memories, and for most possible inter-thread communication patterns. Most applications involve control-flow between barriers, so synchronized data updates can be irregular and even hard to statically predict. GPUs implement hardware barriers in an effort to efficiently address this diversity of barrier use.

In addition to diverse use, most applications execute barriers frequently. For applications with mostly deterministic execution, the table lists the number of total dynamic instructions and dynamic memory instructions per barrier executed. Applications with

Table 5.1: Open-source applications containing barriers.

Suite	Application	Dynamic Insts. Per Bar	Memory Insts. Per Bar	Data-Dependent Control
Lonestar [21]	bfs_wlc	107	15	Heavy
	bfs_wlc_gb	108	15	Heavy
	bh	—	—	Minimal
	dmr	—	—	Heavy
	mst	71	11	Minimal
	pta	—	—	Heavy
	sp	302k	43k	Minimal
	sssp_wlc	60	10	Heavy
Pannotia [23]	fw_block	13	5	Minimal
Parboil [138]	bfs	11	2	Heavy
	cutcp	257	31	None
	histo	6,379	430	Minimal
	mri-gridding	—	—	Heavy
	sad	859	150	None
	stencil	67	10	None
	tpacf	48	7	Minimal
Rodinia [24]	backprop	31	6	None
	btree	28	5	Minimal
	cell	241	13	Minimal
	dwt	61	23	None
	heartwall	26k	2,800	Minimal
	hotspot	39	5	None
	kmeans	37	5	None
	lavaMD	4,432	522	None
	lud	183	77	None
	needle	32	8	None
	pf_float	8,289	817	None
	pathfind	42	7	None
	srad	41	8	None

frequent barriers tend to experience greater overhead, though particular barrier instances can suffer extreme thread wait times. On average, applications execute fewer than 48 dynamic instructions per barrier, and most have 5–15 memory instructions per barrier. However, barriers are often used to synchronize short segments of code that contain just 1–3 memory accesses for quick inter-thread communication. These code segments, which start and end with a barrier, often occur inside loops, so barriers execute just a few hundred cycles apart.

5.2.2 Pre-finalizer Barrier Swapping (PBS)

To measure fence and barrier overhead, we introduce a method, pre-finalizer barrier swapping (PBS), which exchanges particular GPU instructions during application compilation. When compiling a CUDA application, the NVIDIA compiler, `nvcc`, generates intermediate assembly code, PTX ISA, before the assembler/finalizer pass generates NVIDIA machine code, SASS. By saving `nvcc`'s generated files (`--keep` option), the generated PTX code can be manually modified before generating SASS. To investigate barrier overhead, PTX barriers (`bar.sync`) can be removed or swapped with other PTX fence or barrier instructions. After generating PBS binaries with different fences/barriers, we run them on hardware to compare kernel run times.

PBS-generated binaries may not produce correct application output, since changing or removing barriers may not provide correct memory access ordering. However, PBS binaries frequently execute the same code paths, so they can often be used to accurately measure fence and barrier overhead. PBS does not introduce any overhead into kernel execution for timing. Further, it avoids complications from code optimization, which occurs before `nvcc` generates PTX. The NVIDIA assembler/finalizer, `ptxas`, assembles PTX into SASS, with little or no code movement.

Although most applications have no or minimal data-dependent control, more complicated and irregular applications, like Lonestar `dmr` and `sssp`, have heavy data-dependent control, so we are currently unable to estimate their barrier overhead due to incorrect behavior. For PBS performance results presented next, we use detailed GPU profiling to validate that binaries execute the same mix of instructions besides fences and barriers, indicating that hardware exercises the same compute and memory operations during PBS tests.

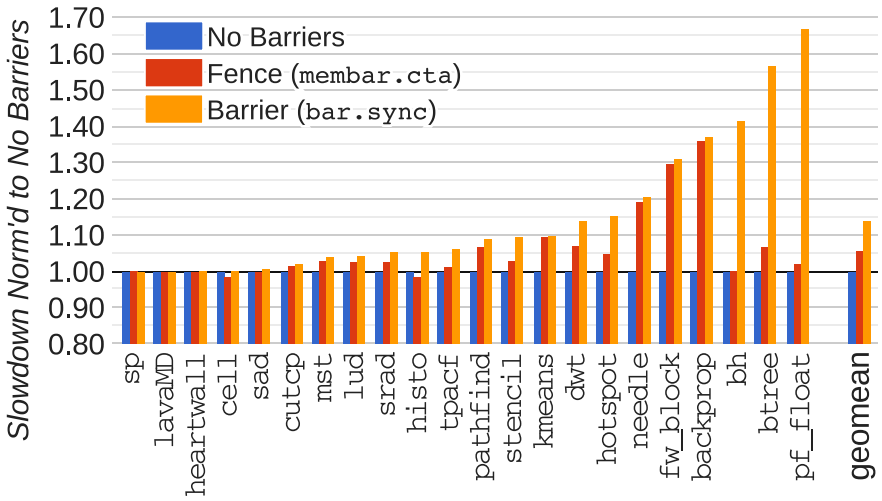


Figure 5.1: NVIDIA GTX860M: PBS-estimated run time slowdown from PTX fences and barriers for LonestarGPU, Pannotia, Parboil, and Rodinia applications.

5.2.3 NVIDIA Hardware Barrier Overhead

Figure 5.1 shows PTX fence and barrier overheads for an NVIDIA Maxwell GTX860M normalized to the PBS application version with barriers removed (first bar in each group). Many applications experience more than a 20% slowdown when using barriers (`bar.sync`) compared to the 'No Barriers' version, and barrier overhead can be as high as 55–65%.

GPU barriers execute an implicit memory fences (`membar.cta`) that order per-thread memory instructions, but Figure 5.1 results also indicate that barrier fences only account for a small portion of total overhead. If fence overhead accounted for all barrier instruction overhead, `membar.cta` and `bar.sync` run times would be similar. However, in the majority of cases, barriers cause significantly more slowdown due to the load imbalance across threads.

5.2.4 Breaking Down Barrier Overheads

As we detail in Section 5.5, current state-of-the-art GPUs implement barriers by draining and committing all of a threadblock's memory instructions before allowing any warps to proceed, an implementation we call "drain barriers". Draining activity during barriers

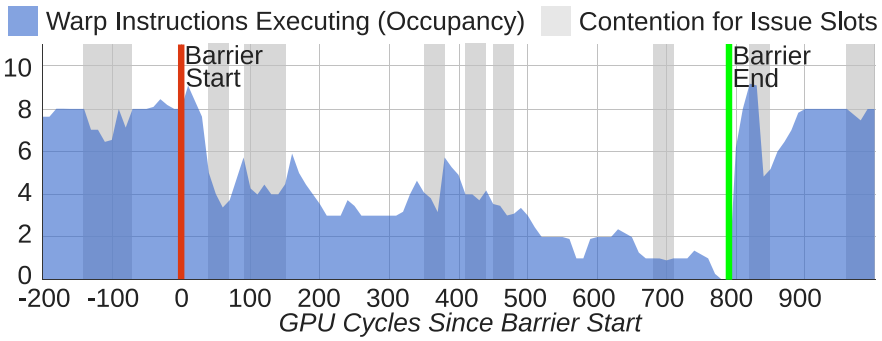


Figure 5.2: `backprop` threadblock warp instruction occupancy during a drain barrier. GTO warp scheduler.

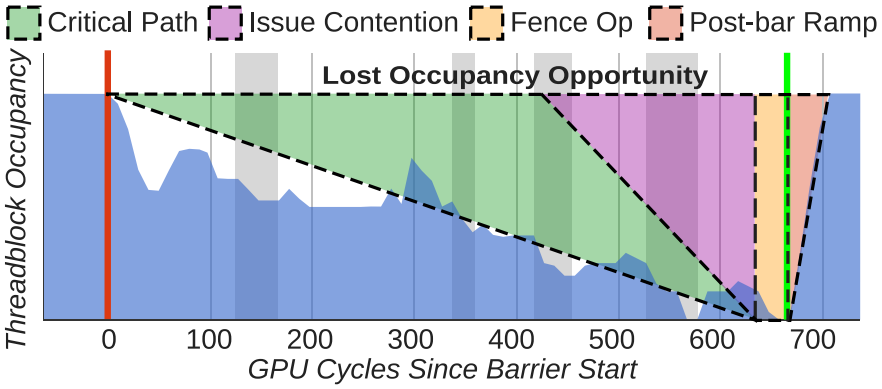


Figure 5.3: Occupancy loss break down during drain barriers.

limits a threadblock's parallel in-flight instructions, or "occupancy". To dissect occupancy loss during barriers, we use simulation to collect data on four factors limiting occupancy and describe them below.

Barrier Overhead is Lost Occupancy: A trace of a threadblock's in-flight instructions depicts occupancy loss near drain barriers. Figure 5.2 shows the number of parallel warp instructions executing for all warps in a threadblock during a representative portion of the Rodinia `backprop` application. As warps reach the barrier instruction, starting with the first one at the red bar, they must stall instruction issue until all the threadblock's warps have completed the barrier (green bar). Occupancy declines as more warps get stuck waiting at the barrier, especially when warps experience load imbalance or contention with other threadblocks for instruction issue slots (shaded cycles in the plot). Commonly, threadblocks can lose 200–700 cycles of in-flight instructions.

In simulation, we count the number of warps with no in-flight instructions during barriers and categorize them to four groups depicted in Figure 5.3. The shaded shapes in the plot approximate these loss factors, but are often accurate estimates. First, threadblocks always experience some load imbalance, so generally there is significant critical path execution time between the earliest and latest warps to arrive at a barrier. Second, during a barrier, a threadblock may contend with other threadblocks for issue slots, causing extra critical path time or ramp up time after the barrier. Third, all warps execute a fence operation at the end of the barrier to ensure memory access ordering. Finally, after a barrier completes, it may take some time for a threadblock to ramp up its occupancy, especially if warps must compete for issue slots.

Note that a threadblock's lost occupancy during barriers does not always directly result in performance degradation. When multiple threadblocks execute on a single GPU core, some may gain occupancy while other threadblocks perform barriers. Using more threadblocks each with fewer warps will tend to reduce load imbalance for quicker barriers. However, GPU cores frequently have insufficient independent instructions from other threadblocks to maintain the core's occupancy during barriers. This challenge is compounded when multiple threadblocks perform barriers concurrently.

Prior Warp Scheduling Proposals Fall Short: Recent prior works aim to reduce barrier overhead with warp scheduling, but unfortunately fail to address the major sources of occupancy loss. Anantpur and Govindarajan [7] and Liu et al. [88] both propose warp schedulers that prioritize warps in threadblocks with the most warps waiting at a barrier, a technique they call Most-Waiting-First (MWF) [88]. MWF scheduling can reduce a threadblock's contention for issue slots during barriers, which in turn reduces barrier latency.

Unfortunately, MWF does not address occupancy loss caused by factors other than issue slot contention. Figure 5.4 shows how MWF reduces the total barrier occupancy loss for the `backprop` and `btree` applications. In `backprop`, all occupancy gain comes from reducing

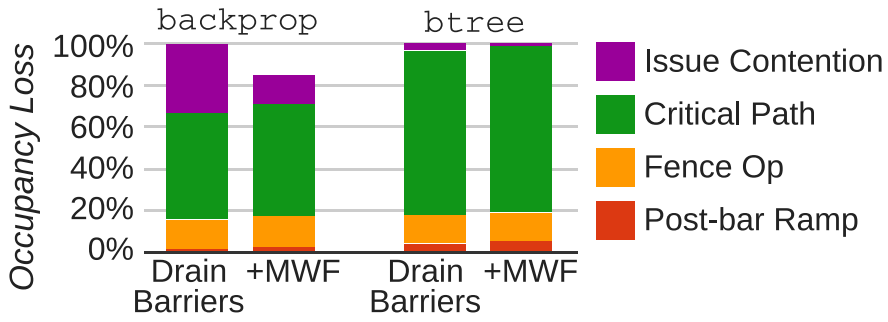


Figure 5.4: Example barrier occupancy loss breakdowns.

issue contention (17% of barrier occupancy loss) for a 1.015× speedup. Other causes of occupancy loss remain the same with MWF scheduling. By simulating PBS binaries with barriers removed as described in Section 5.2.2, we estimate that `backprop` could experience as much as 1.078× speedup if barrier occupancy loss could be eliminated. Further, many applications, such as `btree`, experience little issue contention, so MWF scheduling will not help.

Overall, eliminating barrier overhead will require that GPUs address barrier occupancy loss due to critical path execution, fence overhead, and re-ramping time. Cores will need to expose instructions that can execute while warps wait at a barrier to overlap activity and reclaim lost occupancy.

5.3 Using Fuzzy Barriers in GPUs

Prior work proposes fuzzy barriers [48, 49] as a solution to the load imbalance problem. Fuzzy barriers allow threads to hide barrier latency by executing independent ("fuzzy") instructions while waiting at a barrier. Fuzzy barriers are exposed at the architecture level, so programmers or compilers must specify which instructions are safe to execute at each barrier while threads wait. For GPUs, executing independent instructions during barriers is appealing, because GPUs have numerous threads that could execute instructions to fill lost occupancy slots.

Unfortunately, while fuzzy barriers present an opportunity to reduce barrier overhead,

programmers or compilers can find them difficult to optimize due to their complicated architectural interface. Fuzzy barriers were originally proposed for symmetric multiprocessor CPU systems when such systems offered sequential memory consistency and before formalized models of memory consistency. In these early systems, fuzzy barriers introduced nondeterministic memory behavior. Specifically, threads were allowed to execute fuzzy instructions, but may or may not have executed some fuzzy instructions before all other threads had executed the barrier entry instruction. Thus, programmers had to ensure that if a thread executed fuzzy instructions, they would not cause incorrect memory orderings with data updates synchronized by the barrier.

For GPUs, the fuzzy barrier programming interface is even more complicated. GPUs have relaxed consistency models, so programmers must already carefully add appropriate fences to ensure memory ordering and avoid race conditions. This memory model complexity is compounded by the nondeterministic ordering of fuzzy instructions. Further, GPUs have multiple memory spaces and scopes; programmers would need to reason about which memory spaces are synchronized by fences and barriers, and which cores have visibility to the synchronized data. To our knowledge, fuzzy barriers have not been implemented in GPUs, in part, because of their semantic complexity and the effort required to annotate each barrier with fuzzy instructions.

In this section, we describe our efforts to test fuzzy barriers in GPUs. To avoid exposing their programming complexity, we instead modify the compiler to identify fuzzy instructions and to reorder instructions into fuzzy barrier code regions. Even with aggressive compiler instruction reordering, fuzzy barriers are limited by their architectural interface, which requires the compiler to statically and conservatively specify fuzzy barrier regions. Our evaluation shows that compiler-specified fuzzy barriers can only hide 16% of lost occupancy on average for 1.015× speedup. Section 5.6 details how a dynamic solution can improve fuzzy barriers.

5.3.1 Implementing Fuzzy Barriers in a Compiler

When the compiler identifies fuzzy barrier regions, it must take care to meet their static constraints to ensure correct program execution. We describe fuzzy barriers in more detail to illuminate their static constraints here. We show that these constraints often limit the compiler from extending barrier regions past basic block boundaries.

At the architectural level, fuzzy barriers have entry and exit instructions, and compiled code must mark where threads enter and exit the barrier region. The barrier region is the set of instructions between barrier entry and exit, called "fuzzy" instructions, which a thread may execute either before or after other threads reach the barrier. To enforce barrier semantics, hardware must guarantee that threads block at the exit instruction until all other threads have entered the barrier.

Compiled code with fuzzy barriers must meet two constraints. First, if threads execute fuzzy instructions during the barrier region, they must result in correct barrier memory ordering. Namely, for each barrier region, all memory accesses before the barrier region must appear to complete before any memory accesses during or after the barrier region. Second, all threads must execute both barrier entry and exit instructions to ensure their forward progress. If some thread does not execute its barrier entry instruction, other threads will remain blocked at their exit instructions. Further, since hardware must track which threads have entered and exited each barrier region, all threads must execute each barrier exit instruction to ensure hardware maintains consistent state.

Given these constraints, the compiler must often make conservative decisions about which instructions to include in the barrier region. GPU code often contains memory ordering dependencies and nontrivial branching paths between barriers. Figure 5.5 demonstrates the compiler's limitations with a common code structure that blocks the expansion of the fuzzy barrier region. In this snippet, threads enter a loop and execute a barrier (instruction `b`) to ensure prior memory accesses have completed. After the barrier, threads can take one of two branch paths, either executing scratch ("`shared`") memory accesses in

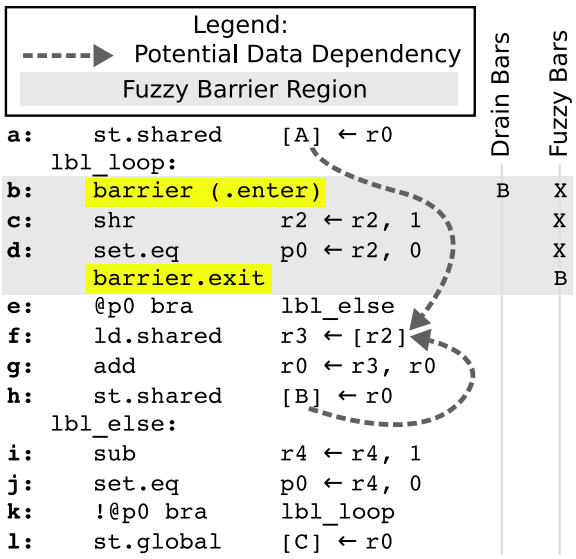


Figure 5.5: PTX assembly code snippet to demonstrate limitations of fuzzy barriers near memory and control instructions. The compiler splits the barrier instruction `b` into entry and exit instructions as highlighted. "`X`" denotes that an instruction can execute during a barrier, and "`B`" denotes that it must block until the barrier completes.

instructions `f`–`h`, or proceeding directly to the bottom of the loop at `lbl_else`. When some thread executes the load instruction, `f`, it might read data written by another thread's store instruction `h` in a prior loop iteration. Thus, unless the programmer or compiler could ensure all memory accesses are to unique memory addresses, the barrier must be executed in each loop iteration to ensure correct memory ordering.

In this example, the compiler can split the barrier instruction (`b`) to introduce a fuzzy barrier region (entry and exit instructions are highlighted), but the region remains short due to the compiler's inability to disambiguate branch paths and memory addresses. The compiler places the barrier entry instruction as early as possible at the head of the loop.

The compiler also inserts the barrier exit instruction as far after the entry as possible. In this example, it must end the barrier region before memory instructions that might require correct ordering, and even before control flow instructions that gate execution of such memory instructions. Specifically, some threads may branch past instructions `f`–`h`, so the compiler cannot move the barrier exit into this set of instructions in case some threads might not execute it. Although all threads execute the else-condition (instructions `i`–`k`), the

compiler cannot move the barrier exit into this set of instructions, because some threads may execute instructions `f–h`, which require the barrier to be complete to ensure prior memory accesses were correctly ordered.

The example in Figure 5.5 is a common code structure in GPU computing applications, and the compiler must often make conservative decisions about how to organize barrier regions. The combination of memory ordering and control flow constraints make it difficult for the compiler to extend barrier regions across basic block boundaries.

5.3.2 Compiler Instruction Reordering

Fuzzy barriers tend to be limited by the number of static instructions in their barrier regions. Compiler instruction reordering could seek out further instructions to execute during the barrier region, but we find that reordering provides very small benefit. Here, we show that compiler instruction reordering, on average, finds 3× more fuzzy barrier region instructions, but these instructions are often found near barrier regions that do not cause significant performance overhead. As noted in Section 5.2.1, GPU code often involves tight loops that contain control flow and synchronize data between threads. Such static constraints are major contributors to barrier overhead, and they often limit the compiler's ability to grow barrier regions. As we show in the evaluation section, even with aggressive compiler reordering, these static constraints limit fuzzy barriers to marginal performance gains of just 1.015×.

To establish a baseline for fuzzy barrier region size, we count the number of static instructions in fuzzy barrier regions for all applications listed in Table 5.1. Figure 5.6 plots the cumulative distributions of fuzzy barrier region size for roughly 2,400 static barriers in applications from the Lonestar, Pannotia, Parboil, and Rodinia suites. The plot shows results for compiler-specified fuzzy barriers without instruction reordering (blue line), as well as distributions for different reordering schemes that we describe below.

Figure 5.6 demonstrates the static limitations of fuzzy barriers when the compiler inserts

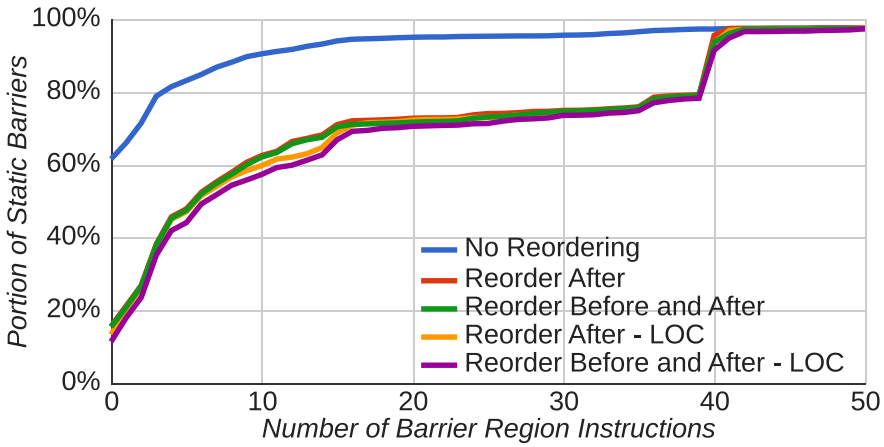


Figure 5.6: Cumulative distribution of number of barrier region instructions for 2,400 static barriers in current open-source GPU computing applications.

barrier regions without reordering instructions ("No Reordering" blue line). In most GPU code, memory instructions or blocking control flow immediately follow barrier instructions. The compiler can only expand barrier regions to contain fuzzy instructions for 38% of barriers. Further, in barrier regions for which the compiler finds fuzzy instructions, there are usually fewer than 10 instructions that can be executed between barrier entry and exit. On average, the compiler finds just 4.5 fuzzy instructions per static barrier.

We test two compiler reordering options that move instructions into the fuzzy barrier regions. In the first reordering option (called "Reorder After" in red), the compiler searches for independent arithmetic instructions after the barrier that it can pull up to the end of the barrier region. These instructions must not upset the ordering of memory accesses or register reads and writes. In the second option (called "Reorder Before and After" in green), the compiler also tracks chains of independent arithmetic instructions leading up to a barrier. Such instructions can be moved to the beginning of the barrier region without harming the correctness of the application, because they will not change memory access ordering around the barrier.

In the two baseline reordering schemes described above, the compiler obeys C++ evaluation order requirements, but we also test removing this restriction. C++ evaluation order requires that for each C++ source code statement, the instructions that perform that

statement must appear to be executed to show their side effects before any instructions that perform further source code statements. This restriction limits instructions from separate code lines from being reordered. The first two reordering schemes enforce this restriction in the compiler. However, some aggressive compiler optimizations often ignore evaluation order requirements. We test more aggressive instruction reorderings that remove this evaluation order requirement, lifting the ordered line-of-code restriction, so we name them "- LOC".

The compiler could perform more aggressive instruction reordering, but it would require program transformations that move independent instructions across basic block boundaries in the presence of memory instruction ordering dependencies. Such an approach would require the compiler to reason about the sets of threads that will execute different control flow paths through basic blocks. For example in Figure 5.5, instruction `i` could be reordered into the barrier region, but the compiler must ensure that all threads will execute it and that it does not harm program correctness. In contrast, instruction `j` has a register dependency on predicate register `p0`, which would cause incorrect execution if reordered into the barrier region. To reorder `j`, the compiler would need to bind its result to a different register and change the downstream dependencies.

Figure 5.6 shows that instruction reordering finds the most independent instructions after barriers. The "Reorder After" distribution indicates that pulling instructions into the barrier region from after the barrier finds fuzzy instructions for 45% more barriers than without instruction reordering. Further, for roughly 20% of barriers, instruction reordering finds more than 35 fuzzy instructions, so on average, barrier region size grows to 14.9 instructions.

The results also indicate that more aggressive instruction reordering techniques show marginal increase in barrier region size. Reordering instructions from before barriers into the barrier region ("Reorder Before and After") does not appreciably increase barrier region size. For about 5% of barriers, the compiler finds up to 10 more fuzzy instructions when

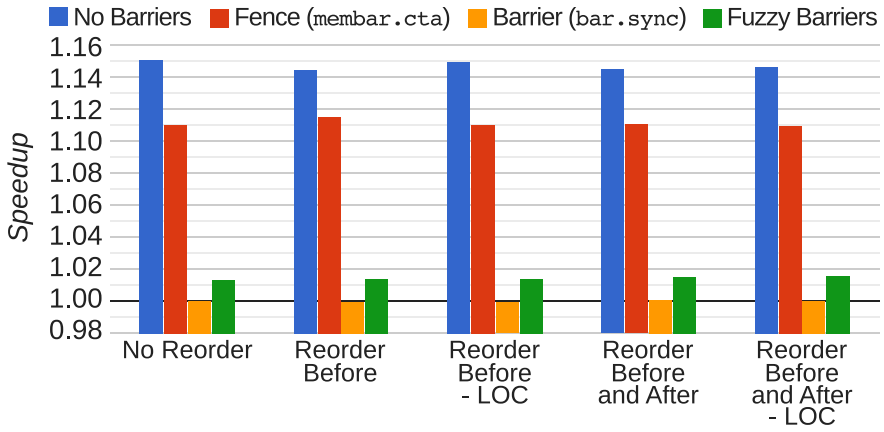


Figure 5.7: Geometric mean speedups when using fuzzy barriers and reordering normalized to drain barriers without instruction reordering. Plot includes speedups with barriers removed and when replaced with fences.

removing the C++ evaluation order restriction, and this gain pushes the average barrier region size to 15.4 instructions.

5.3.3 Fuzzy Barriers Show Small Performance Gain

Although we can implement fuzzy barriers in the compiler, and compiler instruction reordering can grow barrier region size significantly, fuzzy barriers show small performance benefit. Figure 5.7 shows the geometric mean speedups when using fuzzy barriers and the different compiler reordering schemes normalized to the baseline drain barriers implementation.

Overall, we find that fuzzy barriers provide a small 1.013–1.015× speedup over drain barriers. On deeper inspection, the static barriers that are most critical performance are those that are in tight loops and near control flow and memory dependencies. Such barriers tend to have very small barrier regions, typically less than 5 fuzzy instructions. Compiler reordering often finds more fuzzy instructions for barriers surrounded by numerous compute instructions. However, these barriers tend to cause low performance overhead, so increasing their barrier region size rarely improves performance.

The results here indicate that most static barriers are closely followed by control flow

Table 5.2: OpenCL 2 and CUDA architectural specifications for fences and barriers.

OpenCL Fence and Barrier Options			
Operations	**Orders**	**Memories**	**Scopes**
`atomic_work_item_fence`, `work_group_barrier`, `sub_group_barrier`	relaxed, acquire, release, acq_rel, seq_cst	local (scratch), global, image	work_item, sub_group work_group, device, all_svm_devices

CUDA PTX Fence and Barrier Instructions				
Instruction	**OpenCL Equivalent Op.**	**Order**	**Memory**	**Scope (PTX: 'level')**
`membar.cta`	`atomic_work_item_fence`	acq_rel	local (scratch) + global	work_group (`.cta`)
`membar.gl`	`atomic_work_item_fence`	acq_rel	local (scratch) + global	device (`.gl`)
`membar.sys`	`atomic_work_item_fence`	acq_rel	local (scratch) + global	all_svm_devices (`.sys`)
`bar.sync`	`work_group_barrier`	acq_rel	local (scratch) + global	work_group (`.cta`)

or memory instructions that limit static placement of fuzzy barriers. To get around these limitations would require a more advanced barrier technique that allows branch and memory instructions to execute while threads wait at barriers. This challenge motivates our study of microarchitectural techniques to improve fuzzy barriers.

5.4 Architectural Semantics

Although fuzzy barriers aim to address barrier lost occupancy, they expose a complicated architectural interface, and we are not aware of any current systems that support them. To avoid this architectural complexity, we analyze OpenCL 2 [69] and CUDA (PTX) [110, 111] architectural specifications to identify how hardware can legally reorder instructions near barriers without changing the architecture or existing applications. The key requirement is that while performing a barrier, threads must not cause side-effects that, to other threads, appear visibly reordered across the barrier.

5.4.1 Current GPU Fence Specifications

GPU barriers execute fences, which enforce barrier memory orderings. In relaxed consistency memory models, fences provide a means for programmers to enforce ordering between memory reads/writes from a single thread. A common use case is to place a fence

between writes from a single thread to ensure that other threads see those writes in their original program order (i.e., the fence is a release operation).

Table 5.2 summarizes current GPU fence and barrier semantics. OpenCL 2 offers programmers the full cross product of three fence options, which are relevant to our instruction reordering analysis. Fences can provide different memory ordering guarantees, ranging from release/acquire to sequential consistency. OpenCL defines multiple memory spaces to store data, so fences may enforce ordering on specific memories. Finally, fences can enforce memory access ordering at different scopes depending on the desired set of communicating threads. Broader scope fences (e.g., `device`) provide memory ordering guarantees to larger sets of threads, potentially increasing the synchronization latency. For completeness, Table 5.2 also lists the PTX instructions for fences (`membar`) and barriers (`bar.sync`), and their OpenCL equivalent specifications.

5.4.2 Current GPU Barrier Specifications

In GPU architectural specifications, barriers are defined in terms of execution progress of a set of threads, and they implicitly or explicitly specify fences. Unlike fences, which order memory accesses from a single thread, a barrier pauses the progress of a group of threads while their collective memory accesses are ordered to memory. Thus, barriers help many threads exchange data at the same time.

OpenCL Barriers: For an OpenCL workgroup barrier, all work-items in the workgroup must execute the `work_group_barrier` function before any work-items are allowed to continue execution beyond the barrier. Barriers enforce an entry fence that releases a thread's results, and an exit fence that acquires results from other threads. Further, all work-item entry-release fences must complete before any of the exit-acquire fences execute. This precision allows us to evaluate legal operation reorderings during barriers.

PTX Barriers: Though not as precise as OpenCL, CUDA also defines barriers in terms of execution progress. The barrier instruction (`bar.sync`) causes the executing thread to

wait until all threads in the threadblock (workgroup) arrive at the barrier before resuming execution. Barriers guarantee ordering and scope identical to the `membar.cta` fence, so threads in a threadblock can communicate through memory.

5.4.3 OpenCL 2 Instruction Ordering

Although architected fuzzy barriers may be too complicated in current GPUs, recent developments on C/C++ [14] and OpenCL 2 [69] programming models have formalized semantics, which permit precise definition of instruction reordering that hardware can perform near barriers. As described below, threads can execute fuzzy instructions that access thread-private data, such as registers and local memory. Further, accesses to shared data locations can be buffered until a barrier completes.

For C++ and derived languages, including OpenCL, Batty et al. [14] provide a formal definition of code side-effect visibility, which we use to identify correct operation reorderings around barriers. A code statement causes a side-effect when it modifies some hardware state. Plainly, the formal definition of side-effect visibility states that for operations on a single data location, a non-atomic read can only receive data from the last write operation that was ordered ahead of the read. The relation ordering these operations is called '*happens-before*'. *Happens-before* is the transitive closure of two other relations, *sequenced-before* and *synchronizes-with*, which dictate valid partial-orders of per work-item (thread) and memory operations, respectively.

According to the spec, OpenCL barriers are split so that each thread executes entry-release and exit-acquire fences, and each entry-release fence *synchronizes-with* all exit-acquire fences for participating threads. Figure 5.8 diagrams this ordering for operations executed near a barrier. Traversing through the barrier's operations, *happens-before* transitivity implies that instruction side-effects read/written by any operation, d_j, must appear to come after all side-effects read/written by operations at a_i or earlier (i.e., $\forall i, j, a_i \xrightarrow{hb} d_j$). Using this setup, we can precisely identify operations allowed to read/write side-effects

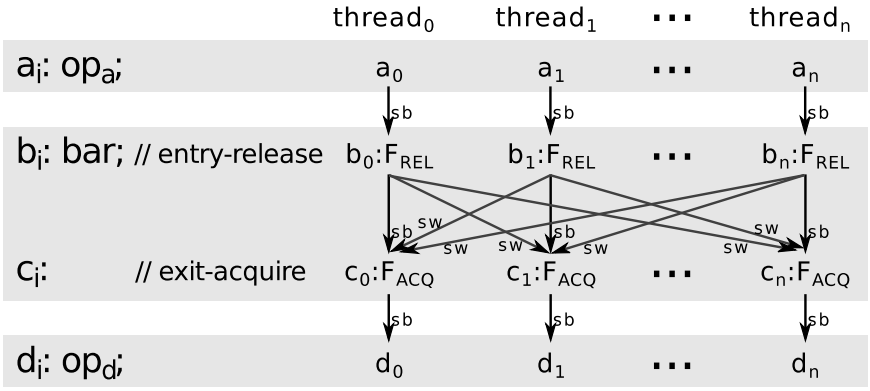


Figure 5.8: OpenCL *sequenced-before* (sb) and *synchronizes-with* (sw) relations around barriers.

while a thread waits at a barrier.

First, since side-effect visibility only applies to reads and writes on the same location, operations on disjoint locations could be reordered. However, we expect it would be exceptionally expensive to disambiguate memory operation locations while enforcing fence orderings across numerous threads during barriers. Thus, the following side-effect visibility assessment only considers opportunities to reorder operation classes rather than disjoint-data operations. Second, reordering d_0 operations ahead of any operations before $b_{i\neq 0}$ depends on the memory's visibility: private or shared among threads.

Private Data Locations: Separate threads might want to access data in their thread-private locations, such as registers or private memory. Since private data locations are not visible to other threads, operations that use registers or private-memory (e.g., most compute operations) could be evaluated while a thread is performing barrier fences without introducing visible side-effects to other threads. Further, each thread must appear to evaluate operations in program order (e.g., $a_0 \xrightarrow{sb} d_0$). Fortunately, existing register scoreboards and memory disambiguation already enforce this program order requirement.

Shared Data Locations: Separate threads might access shared, read-write data locations, such as local, global, or image memory spaces. To uphold the *happens-before* relation, $\forall i, a_i \xrightarrow{hb} d_0$, a side-effect written by d_0 must be suppressed to not become visible until after all operations prior to b_i are known to have read their data. Similarly, a side-effect

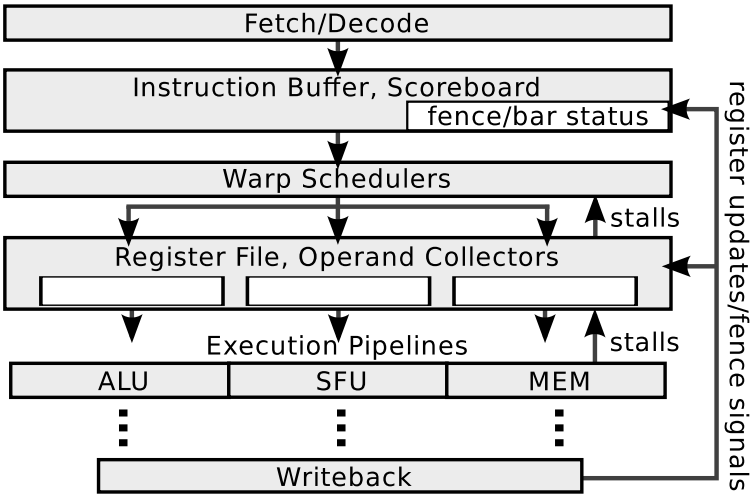


Figure 5.9: Generic GPU core microarchitecture.

read by d_0 must wait until after all operations prior to b_i have written their data. A microarchitecture could enforce this shared data operation ordering by buffering and maintaining order of post-barrier memory instructions as long as they read/write their side-effects after the barrier's synchronization is complete.

Read-Only Memories: This work focuses on ordering access to local (scratch) and global memory spaces, which allow side-effect reads and writes, but OpenCL and CUDA also have read-only memory spaces. Constant, texture, and parameter data locations are updated with blocking APIs to ensure ordering with GPU accesses. Given these constraints, GPU threads can read this data at any time, regardless of barrier activity.

To conclude, the OpenCL specification permits microarchitectures that can perform the following operations while threads are waiting at a barrier. First, if a post-barrier operation accesses thread-private data locations, it can read/write side-effects in program order. Second, if the operation might access data visible to another thread, it can be buffered but must wait to read/write side-effects until the barrier appears complete.

5.5 Fence and Barrier Microarchitecture

This section describes straightforward GPU fence and barrier implementations that drain memory instructions before allowing warps to proceed. Microbenchmarking suggests that NVIDIA Maxwell GPUs block warps at issue to drain memory accesses and enforce ordering. As we describe these implementations, we refer to Figure 5.9, which depicts a GPU core microarchitecture loosely based on current GPUs, such as NVIDIA Maxwell [108] or AMD's Southern Islands [6].

5.5.1 Fence Microarchitecture

GPU microarchitectures aim for simple instruction handling and in-order issue to efficiently execute hundreds of concurrent threads with minimal buffering. Instructions, which might be executed by many concurrent warps, are fetched, decoded, and placed in instruction buffers for the scoreboard to inspect. In program order, the scoreboard checks whether a warp's current instruction satisfies register dependencies and other constraints. If so, warp schedulers arbitrate among the latest warp instructions to dispatch to execution pipelines.

Drain Fences: The scoreboard can enforce drain fences by blocking instructions from proceeding to warp schedulers while fence operations are in flight. To enforce the release portion of a fence, instructions behind the fence stall to wait for a response from all of a warp's prior memory instructions. The scoreboard inspects these responses to ensure that the memory accesses have reached the appropriate scope specified by the fence. Thus, a fence's wait time can be as long as a contended memory access out to GPU global cache or off-chip memory. To enforce the acquire portion of a fence (i.e., loads after the fence access current data versions), the scoreboard can continue to block instructions behind the fence to wait for caches to flush any stale data within the fence's scope.

Figure 5.9 shows the signals that are involved in this drain fence implementation. Specifically, just as register updates are passed from the register file to the scoreboard,

so too are signals indicating the completion of accesses and acquire flushes. When a fence operation is in progress, the scoreboard sets a single bit per warp to indicate that it must block further instructions from issuing and executing. As memory instructions and the fence operation complete, the signals indicate their completion. When all in-flight operations complete, the scoreboard can clear the in-flight fence operation bit to allow the warp to proceed.

5.5.2 Barrier Microarchitecture

Drain Barriers: Like drain fences, barriers can also be implemented by blocking a warp's instruction issue while waiting on barrier fence operations and other warps. When a warp reaches a barrier instruction, the scoreboard blocks the warp from issuing instructions and sets a bit in a barrier bit vector indicating that the warp has entered the barrier. The scoreboard detects, based on threadblock IDs and signals from the memory pipeline, when all participating warps have completed their entry-release fences, and at that point, warps are allowed to issue the barrier's exit-acquire fences in any order. The logic for starting the exit-acquire fence is as follows:

$$\texttt{start\_exit\_fence}(B_i) = \forall w_j \in B_i \left[\bigwedge \texttt{entry\_fence\_done}(w_j) \right]$$

Here, B_i is the ith threadblock on the core, and w_j is the jth warp in the threadblock. Such an implementation is a natural choice for GPU cores due to its simplicity.

NVIDIA Maxwell Barriers: We believe that NVIDIA Maxwell fences and barriers are implemented using draining. To analyze the Maxwell microarchitecture, we constructed a battery of microbenchmarks that execute tightly-controlled chains of various instructions and dependencies between barriers, and we ran PBS versions of these microbenchmarks on hardware to measure fence and barrier overheads. We find that Maxwell cores do not hide operation latencies across fence or barrier instructions, indicating that warp issue

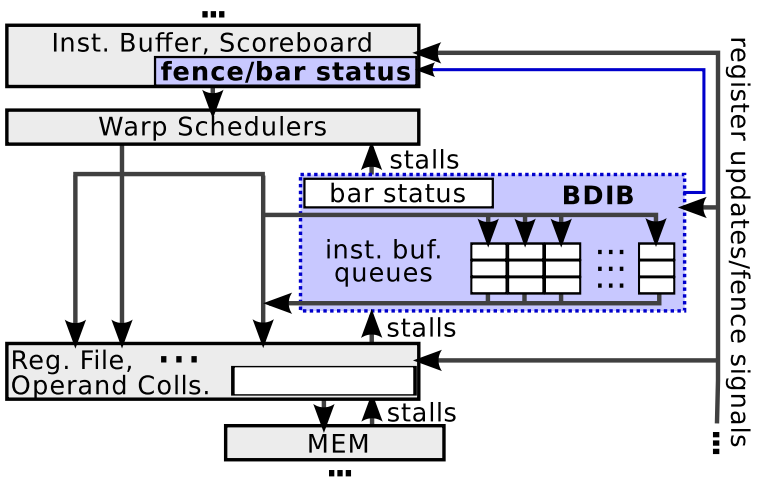


Figure 5.10: Highlighted in blue: TFB implementations modify scoreboard fence/barrier logic, and G-/F-TFB implementations add the barrier-displaced instruction buffer (BDIB) between warp schedulers and operand collectors.

halts at both. This microbenchmarking result is consistent with drain fence and barrier implementations.

5.6 Transparent Fuzzy Barriers

As described in Section 5.4, GPU architecture specifications allow threads to read/write side-effects in thread-private data locations during a barrier, while side-effect reads/writes on shared data must maintain correct ordering. To implement this transparent fuzzy barrier operation, a GPU microarchitecture must expose instructions from barrier-blocked threads to be issued and executed. GPU threading can offer numerous parallel instructions, but TFB implementations must preserve efficient GPU instruction handling.

Starting from a drain barrier microarchitecture, Figure 5.10 highlights the modified and added hardware to support TFB implementations described in the following subsections. Broadly, to expose barrier-parallel instructions while maintaining the GPU's in-order instruction issue, TFBs make small modification to the scoreboard's fence/barrier stalling logic to permit warps to issue post-barrier instructions under specific conditions. TFB implementations that allow shared memory instructions to issue send them to a new

component, the Barrier-displaced Instruction Buffer (BDIB), which buffers memory instructions and tracks barrier progress. While warps are performing barriers, the BDIB accepts shared-location memory instructions, unblocking the issue stage for the warp so that subsequent instructions might issue and execute. When a barrier completes, the BDIB releases that barrier's buffered instructions, correctly ordering them to memory. All other core microarchitecture can remain unchanged, including branch handling, register management, and warp scheduling.

The three TFB implementations below offer the core varying degrees of visibility to issue and execute instructions while threads are performing barriers. Table 5.3 provides a summary of microarchitectural activity allowed by the different implementations, and Figure 5.11 provides an assembly code snippet to demonstrate their varying capabilities.

5.6.1 Private-memory TFBs (P-TFBs)

To conservatively ensure that memory instructions do not write visible side-effects during barriers, threads could block from issuing instructions addressed to shared data locations. In the first TFB implementation, "*private-memory*" (P-TFBs), the scoreboard blocks memory instructions addressed to scratch and global memories while barriers are in flight. However, instructions that are addressed to private data locations, such as register-addressed compute operations, can be issued and executed as they will not cause visibly reordered side-effects.

P-TFBs require minor changes to scoreboard barrier handling to check which memories the next instruction accesses. If the instruction's opcode or control signals indicate it will

Table 5.3: Barrier implementation capabilities.

Instruction Type	Drain Bars	Fuzzy Bars	Priv. TFB	Global TFB	Full TFB
Private-addressed	block	exec.	exec.	exec.	exec.
Read-only-addressed	block	exec.	exec.	exec.	exec.
Control	block	–	exec.	exec.	exec.
Global-addressed	block	–	block	issue	issue
Scratch-addressed	block	–	block	block	issue
Fence, Barrier	block	–	exec.	issue	issue

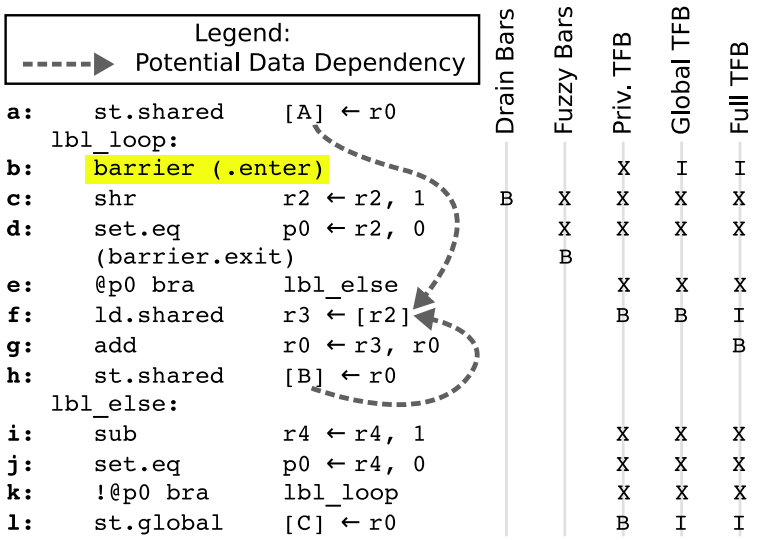


Figure 5.11: PTX assembly code snippet to demonstrate varying fuzzy barrier capabilities for memory and control instructions. "X" denotes that an instruction can execute during a barrier, "I" denotes that it can issue to the BDIB, and "B" denotes that it must block until the barrier completes.

access thread-private memory, such as registers, the scoreboard sends the instruction to warp schedulers as soon as registers are ready, regardless of whether the warp is performing a fence or barrier. For instance in Figure 5.11, P-TFBs allow warps to issue and execute instructions `c`, `d`, and `e` while the warp is performing the barrier. However, the scoreboard blocks instructions addressed to global or scratch memory (e.g., `f`) from proceeding to the warp schedulers until after the barrier.

P-TFBs tend to expose a few register- and private-addressed instructions for execution during barriers. Since GPU cores issue instructions in-order, P-TFBs can only issue up to the first shared-location memory instruction after a barrier. Barriers are used for inter-thread sharing, so shared-location memory instructions often closely follow barriers. The following TFB implementations aim to mitigate this limitation.

5.6.2 Global-memory TFBs (G-TFBs)

The scratch and global memory pipelines in existing GPUs have varying microarchitectural constraints. Scratch memory pipelines maintain minimal buffering and tight timings,

which only vary slightly for coalescing or cache port conflicts. In contrast, global memory hierarchies provide significant access buffering and reordering to hide long latency cache misses [63]. Thus, global memory pipelines may be more accepting of additional instruction buffering during in-flight barriers.

In the second TFB implementation, "*global-memory*" (G-TFBs), the scoreboard still blocks scratch memory instructions during barriers, but allows global memory instructions to issue and buffer for correct execution order. This way, the scoreboard and warp schedulers can view (in program order) further instructions that might be able to issue or execute.

To correctly manage global memory instruction ordering during a barrier, G-TFBs introduce the BDIB depicted in Figure 5.10. When a barrier instruction's entry-release fence issues, the BDIB snoops the fence and sets a bit indicating that the warp has entered a barrier. While a barrier is in flight, the BDIB queues all global memory instructions issued from warps that have entered the barrier. In Figure 5.11, during a G-TFB barrier, threads could execute through the taken-branch at `e` and execute instructions `i`–`k`. Threads can issue the global store at `l` to the BDIB to buffer, and may even be able to execute beyond `l`. Buffering `l` prevents its memory read from viewing potentially stale data before other threads finish the barrier. When the barrier completes, the BDIB detects warps finishing exit-acquire fences, and begins pushing barrier-displaced instructions directly into the memory pipeline.

The BDIB communicates to the scoreboard and warp schedulers with minimal extra signaling. While draining memory instructions after a barrier, the BDIB modifies existing stall signals to the warp schedulers to indicate that it is dispatching to the memory pipeline, so the warp schedulers must block. Similarly, when a warp's BDIB buffers fill, the BDIB must block the warp schedulers from issuing further global memory instructions to that warp. It indicates these conditions by modifying the existing signals back to the warp schedulers.

Table 5.4: TFB implementation hardware complexity per GPU core.

Pipe Stage	P-TFBs	G-/F-TFBs
Scoreboard	Logic	1-bit per warp for BDIB full Total: $\leqslant$8B
Fuzzy Instruction Buffering	None	BDIB: Instruction buffers (10–14B per instruction per warp) + barrier state Total: $<<$1kB for compact design

5.6.3 Full TFBs (F-TFBs)

Through testing, we find that the majority of applications with barriers use scratch memory intensively for inter-thread communication. For these applications, issuing global memory instructions during a barrier is still insufficient to improve performance. We further propose a TFB implementation, "*full fuzzy*" (F-TFBs), that can buffer all shared-location memory accesses during barrier execution.

F-TFBs differ minimally from G-TFBs. In addition to issuing private and global memory instructions during a barrier, the scoreboard allows scratch memory instructions to issue. During a F-TFBs barrier, the BDIBs intercepts and buffers both scratch and global memory instructions (e.g., both instructions `f` and `l` in Figure 5.11 if the BDIB has space). It sends the same stall signals to the scoreboard and warp schedulers as G-TFBs.

5.6.4 TFB Hardware Complexity

The above TFB implementations aim for modest logic and storage overheads. Table 5.4 summarizes our estimates of their hardware complexity. As the last subsections describe, the logic and signaling changes required to permit fuzzy instruction issue and execution are straightforward. Compared to existing scoreboard logic, which inspects opcodes and register conditions, TFBs introduce a small extra test whether to unblock particular instructions during a barrier.

The BDIB stores minimal state to communicate to the memory pipeline: instruction opcodes and control signals, register specifiers, and the warp's active thread mask. We estimate this storage at 10–14B per warp instruction buffer. For GPU cores that can maintain

up to 64 concurrent warps, this storage constitutes less than 900B to buffer one warp instruction per warp. Since applications often execute less than 3 memory instructions between barriers, the BDIB requires 1–2 buffers per warp, as we show in Section 5.7. Further, opcodes, control signals, and register specifiers are largely redundant across buffered instructions, so we expect that a storage-compact design could compress the BDIB by more than a factor of 2 to be ~1kB, or less than 1% the size of a GPU register file (e.g., 128–256kB).

In addition to their compact size, the BDIB is organized to be compatible with energy-efficient GPU cores. Specifically, it uses per-warp or per-block FIFO queues to facilitate efficient draining as barriers complete and offer limited GPU out-of-order issue. The queues avoid energy-expensive associative lookups common in out-of-order CPU instruction buffers. Though we evaluate the BDIB as a separate unit, it may be possible to achieve the same logical organization in existing instruction buffers. Prior studies [142, 78] suggest that NVIDIA Fermi cores contain two instruction fetch buffers per warp. If added logic could implement the TFB BDIB in this buffering, it would eliminate the BDIB's extra storage.

Finally, a configuration parameter of the BDIB is the depth of buffering that it permits per warp. Application data in Table 5.1 indicates that applications often execute less than 8 memory instructions on average between barriers. This observation suggests that unblocking just one or a small number of memory instructions per warp may be sufficient to expose significant barrier-parallel instructions. We test this BDIB depth parameter in the evaluation.

Exception Handling: If a GPU threadblock context needs to pause execution—for example, to handle an exception—the BDIB must manage its stored state to maintain correct thread progress. Current GPUs do not implement general exception support, but some GPUs do support particular exception handling capabilities. Although we anticipate further exception handling capabilities in GPUs, an actual exception handling implementation with TFBs is outside the scope of this work.

We expect that TFBs can be used in GPU cores that implement various exception handling techniques. BDIB state is not speculative, so the in-flight instructions it contains represent forward progress for threads. This state can be saved or cleared as necessary for different exception handling implementations. A GPUs large register capacities would require very heavy replication overheads to support precise exceptions, so such implementations are unlikely. However, with a restartable exceptions implementation, such as iGPU [93], if a thread is unaffected by an exception, its BDIB state can be saved in a manner similar to other volatile state. On the other hand, if a thread must be rolled back to a checkpoint restore point, the BDIB instruction entries for that thread could be selectively reviewed and cleared if they represent instructions since the checkpoint. After exception handling is complete, the BDIB can be restored either from a shadow buffer that replicates the state or by reloading from a specific memory location containing the saved state.

5.7 Evaluation

This section tests the TFB implementations, and shows that TFBs provide performance gains of 1.038–1.061×. On average, full transparent fuzzy barriers (F-TFBs) hide more than 50% of occupancy loss, and often completely eliminate performance overhead. Compared to drain barriers, TFBs expose more instructions during barriers to offer the warp schedulers the option to improve occupancy. TFBs show further improvement (1.075× speedup) when warp schedulers consider barrier progress like most-waiting first (MWF). Remaining barrier overhead factors are due to extreme load imbalance and memory ordering requirements, which would be difficult to recover.

5.7.1 Methodology

Simulated Processor: To test barrier implementations in an emerging memory hierarchy, we simulate a heterogeneous CPU-GPU processor with cache coherence across CPU L1+L2

Table 5.5: Heterogeneous processor parameters.

Component	Parameters
CPU Cores	(4×) 4-wide out-of-order, x86 cores, 3.5GHz
CPU Caches	Per-core 32kB L1I + 64kB L1D and exclusive, private 256kB L2 cache, 128B lines
GPU Cores	(16×) Maxwell-like SMMs, 700MHz, 2×2 GTO [123] warp schedulers, up to 32 threadblocks, 64 warps of 32 threads, 96kB scratch memory, 64k registers
GPU Caches	(2×) 24kB L1 per-core. GPU-shared, banked, non-inclusive L2 cache 2MB, 128B lines
Interconnects	GPU L1/L2: Dance-hall, All L2s/MCs: High-bandwidth, 12-port switch
Memory	(4×) shared GDDR5 channels, 179 GB/s peak

caches and the GPU's shared L2 cache. Figure 1.2 diagrams the processor, and Table 5.5 lists its parameters. CPU cores are comparable to current out-of-order superscalar cores, and have private L1s and an L2 cache. The GPU contains NVIDIA Maxwell-like SMMs, which can each manage up to 2,048 concurrent threads and issue up to two 32-wide SIMT instructions per cycle. GPU cores each have 96kB of scratch memory and 48kB data + instruction L1 cache, and they share a 2MB L2 cache. To focus tests toward GPU performance, we model GDDR5 memory comparable to current discrete GPUs.

Since CPU and GPU have coherent caches, this organization experiences some overheads of heterogeneous processor coherence traffic, so simulated cache latencies can be greater than NVIDIA GTX860M hardware. Microbenchmarking suggests that Maxwell cores implement more aggressive warp scheduling than our simulated GTO scheduler. The GTX860M tends to perform better when executing PBS binaries with barriers removed, indicating that barrier overheads are higher in hardware than in our simulation.

Simulator: To compare GPU barrier implementations, we use the gem5-gpu simulator [120]. gem5-gpu is a full-system heterogeneous processor simulator with flexible cache and memory hierarchies. gem5-gpu uses the out-of-order core model from gem5 [18] and gem5's Ruby to model coherent caches. We modified gem5-gpu's GPU model (GPGPU-Sim v3.2.2 [10]) to more accurately model the NVIDIA Maxwell cores, including compute and memory latencies and throughputs. We validated correctness of all fence and barrier implementations using consistency litmus tests, and validated that baseline drain barriers show strong performance correlation with Maxwell.

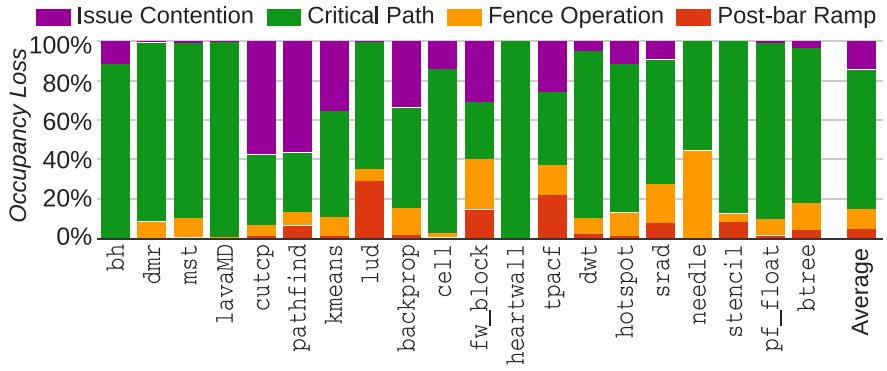


Figure 5.12: Breakdown of occupancy loss during drain barriers.

Applications: The evaluation below presents results for all LonestarGPU, Pannotia, Parboil, and Rodinia applications containing barriers that currently run correctly and to completion in gem5-gpu. To depict correlations among data, we order results from applications with least simulated slowdown due to barriers to most slowdown. Three applications contain data-dependent control flow that disallows measuring accurate barrier overheads in simulation (`bh`, `dmr`, and `mst`). Further, we separate "*barrier-insensitive*" applications that show less than 4% slowdown from barriers (usually due to their low barrier frequency or minimal load imbalance) from "*barrier-sensitive*" applications that show more than 7% slowdown.

5.7.2 Improving Occupancy

Figure 5.12 breaks down occupancy loss during drain barriers for all tested applications. The majority of occupancy loss (70% on average) is due to load imbalance, which causes long critical paths between the earliest and latest warps to arrive at each barrier. Issue contention can be a major source of occupancy loss in some applications, such as `cutcp`, `pathfind`, and `backprop`. Such applications are more likely to gain occupancy during barriers when using MWF warp scheduling. However, MWF scheduling cannot help applications with little issue contention, such as those that execute one threadblock per core (`heartwall`, `stencil`) or those with one warp per threadblock (`needle`). In these cases,

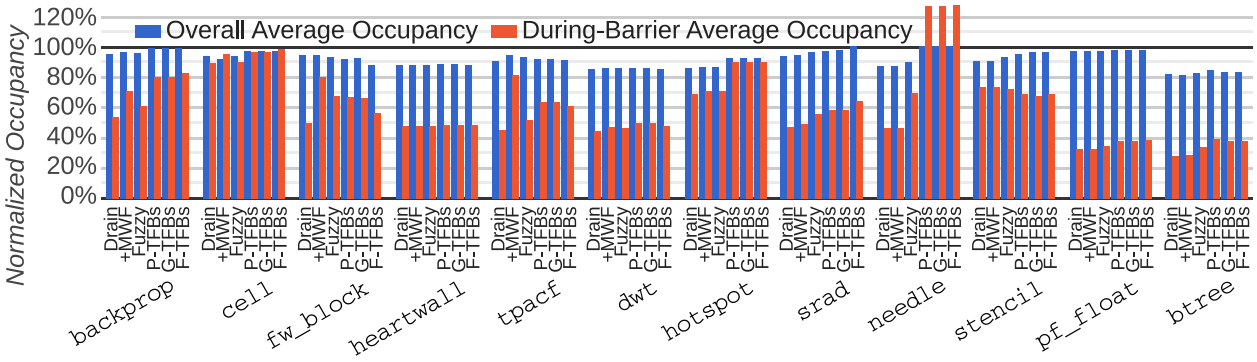


Figure 5.13: Overall average and during-barrier average warp instruction occupancy per threadblock of barrier-sensitive applications. Data are normalized to overall occupancy with barriers removed to show potential for occupancy gains.

threadblocks rarely or never compete with others for issue slots during barriers.

Though occupancy loss can indicate potential sources of performance loss, it is often difficult to reason about how occupancy loss affects overall performance. Threadblocks running on the same core may be able to pick up occupancy slots to hide occupancy lost by other threadblocks with barriers in flight. To observe a clearer tie to performance, we shift our attention from occupancy loss to occupancy.

Occupancy Characteristics: Figure 5.13 plots the per-threadblock warp instruction occupancy for barrier-sensitive applications and each system configuration, normalized to runs with barriers removed using the PBS method. Blue bars show the average occupancy through the whole application run time, while red bars show threadblock's average occupancy while executing barriers. During-barrier occupancy is often near 50%, consistent with estimated losses broken down in Figure 5.3. The figure shows the potential for improving occupancy; drain barriers (left-most bars in each group) often degrade overall occupancy by 4–17%. Further, since applications are ordered from least to greatest slowdown due to barriers, this data indicates the correlation between decreased overall occupancy and decreased performance.

Typically, MWF warp scheduling (second pair of bars) does not provide occupancy gains, but does provide marginal gains for a few applications. By prioritizing threadblocks while they execute barriers, during-barrier occupancy sometimes improves, as seen with

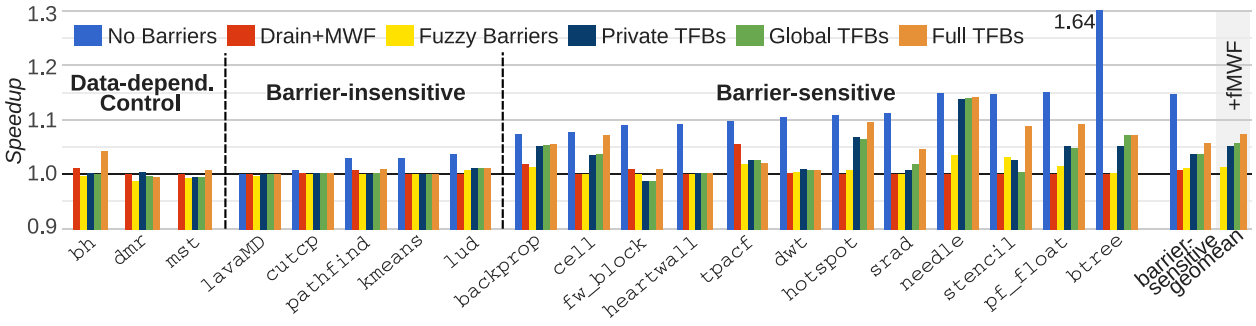


Figure 5.14: Application run times for MWF and fuzzy barrier implementations normalized to baseline drain barriers.

`backprop`, `cell`, `fw_block`, and `tpacf`. However, any time that a threadblock executing a barrier is prioritized over others using MWF, during-barrier occupancy gains come at the expense of another threadblock's occupancy. `cell` is an example in which MWF scheduling actually reduces overall occupancy as a result of this trade-off.

Starting with the third pair of bars in each group, fuzzy barriers and TFBs show better occupancy characteristics. In most cases, they expose more instructions that can be executed during barriers, which improves during-barrier occupancy. This improvement often results in increased overall occupancy, sometimes improving it to be comparable to the baseline with barriers removed (e.g., `backprop`, `cell`, `srad`, `needle`).

`stencil` is a notable exception that highlights a benefit of fuzzy barriers and TFBs. During-barrier occupancy declines by 5%, but overall occupancy increases by 7%. Fuzzy barriers allow warps to sustain more load imbalance through barriers; barrier latency increases, but warps outside barriers expose more instructions that can be issued to separate execution pipelines. In contrast, drain barriers cause warps to bunch up and compete for issue slots after barriers complete, resulting in the post-barrier ramp up contention shown in Figure 5.12.

5.7.3 Improving Performance

Figure 5.14 plots the speedup of MWF warp scheduling and fuzzy barrier implementations normalized to the drain barriers. Where we could verify correct code path execution, the

plot also includes the speedups from removing barriers; the 'No Barriers' bar estimates upper bounds on speedups if an implementation could eliminate barrier overhead (though removing barriers can result in incorrect program outputs).

First, as expected, MWF warp scheduling and fuzzy barriers provide small benefit for few applications. MWF provides a geometric mean speedup of just 1.010×, similar to results shown in prior work. Fuzzy barriers frequently cause warps to block at the first branch instruction after a barrier until all warps have reached the barrier. As a result, they hide little barrier overhead, and gain only 1.015× speedup.

TFB implementations, on the other hand, can often hide a significant portion of barrier overhead, resulting in speedups of up to 1.08–1.14×. As expected, F-TFBs hide the greatest portion of barrier latency. For barrier-sensitive applications, F-TFBs offer an average 1.061× speedup, and can even completely eliminate performance degradation from barriers.

Each TFB implementation exposes different sets of instructions that can be issued or executed during barriers. Since P-TFBs only issue instructions up to the first shared-location memory instruction following a barrier, they offer the smallest performance gains (geomean 1.038×). This result suggests that to hide more barrier latency may require unblocking memory instructions during barriers, as with G-TFBs and F-TFBs. Many applications access global memory between barriers, but global accesses tend to be a small portion of memory activity between barriers. G-TFBs see little benefit beyond P-TFBs. Instead, barriers frequently facilitate tight inter-thread communication through scratch memory. Thus, F-TFBs can substantially outperform P-TFBs and G-TFBs by unblocking scratch memory accesses, as in `cell`, `stencil`, and `pf_float`.

TFB limitations: Although TFBs provide performance gain in many cases, application characteristics can limit their ability to hide barrier latency. First, across all applications, an average dynamic barrier requires fewer than 200 fuzzy warp instructions to execute to completely hide occupancy loss. Unfortunately, a few poorly parallelized applications cause extreme load imbalance that would require significantly more fuzzy instructions.

For example, `heartwall` contains a region of code during which only two threads per threadblock calculate a wide sum while other threads wait at the next barrier. TFBs find 80–100 dynamic fuzzy instructions to execute at the barrier, but they fill less than 0.1% of occupancy slots in this imbalanced code region. These code regions would need to be modified to reduce load imbalance.

Second, intense memory accesses can limit the benefit of TFBs in a couple ways. Applications like `fw_block` and `srad` contain intense scratch memory access between barriers that often bottlenecks performance. TFBs do little to modulate these intense accesses, and sometimes cause minor performance degradation (e.g., $<$1% in `fw_block`). In other cases, when using the BDIB, the scoreboard may block warps from further instruction issue due to register dependencies on BDIB instructions. For some applications with low ILP after barriers (e.g., `dwt`), G-TFBs or F-TFBs find few fuzzy instructions beyond the first memory instruction.

Combining TFBs and warp scheduling: TFB performance can be further improved by adjusting warp schedulers to better utilize fuzzy instructions. When TFBs expose fuzzy instructions during a barrier, warp schedulers sometimes choose fuzzy instructions over critical path instructions from warps that are yet to reach the barrier. For applications with few threadblocks per core, threadblocks sometimes contend with themselves, prolonging barriers. To address this challenge, we adjust the MWF scheduler for TFBs: fuzzy MWF ("fMWF") prioritizes warps in threadblocks with the most warps at/past a barrier, but only prioritizes those warps yet to reach the barrier. Warps issuing fuzzy instructions while at a barrier keep the same priority as warps in threadblocks not executing barriers.

Overall, TFBs + fMWF provide an additional 1.014$\times$ performance lift, as highlighted at the right of Figure 5.14. F-TFB performance improves to an average 1.075$\times$ speedup for barrier-sensitive applications. Most applications see small gains, but applications with few threadblocks per core, like `tpacf` and `stencil`, show improvement of up to 1.10$\times$ by cutting down fuzzy instruction issue contention.

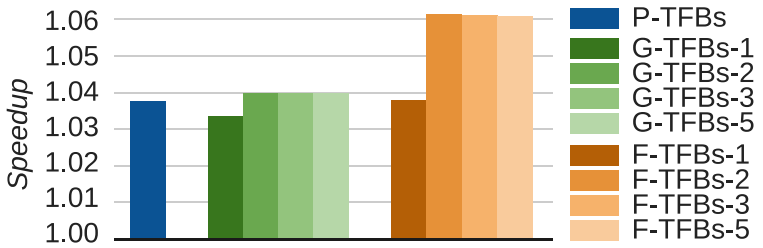


Figure 5.15: Geometric mean speedups normalized to drain barriers for varying BDIB buffering.

Configuring the BDIB: A key parameter of the barrier-displaced instruction buffer is the number of instructions it can buffer while barriers are in flight. Buffering more memory instructions potentially increases each warp's visibility of parallel instructions to issue or execute during barriers. Figure 5.15 plots the geometric mean performance of TFB implementations with 1, 2, 3, and 5 instruction buffers per warp. If the BDIB has few buffers, it may fill and cause warps with further memory instructions to block issue. Fortunately, filling occurs only infrequently for BDIBs with 1 buffer per warp. Across most applications, 2 BDIB buffers per warp are sufficient to keep a full BDIB from causing issue stalls, and performance remains stable beyond 2 buffers per warp. Figures 5.13 and 5.14 report results using 2 BDIB buffers per warp.

5.7.4 Barriers with Broader Scope Fences

Until now, our analysis of barriers has only considered barriers that enforce threadblock/-workgroup scope (`membar.cta` fences). However, future applications targeted at heterogeneous processors are increasingly likely to use barriers with broader fence scope, such as device (`membar.gl`) or all_svm_devices (`membar.sys`), to communicate results to other cores or devices. Broader scope barriers generally increase the critical path and fence operation sources of occupancy loss during barriers, because they must enforce memory access ordering out to the specified scope.

Up to this point, our results show that workgroup-scope barriers cause average slowdowns of 14%. Device-scope barriers cause slowdowns to increase to 21%. We collect these

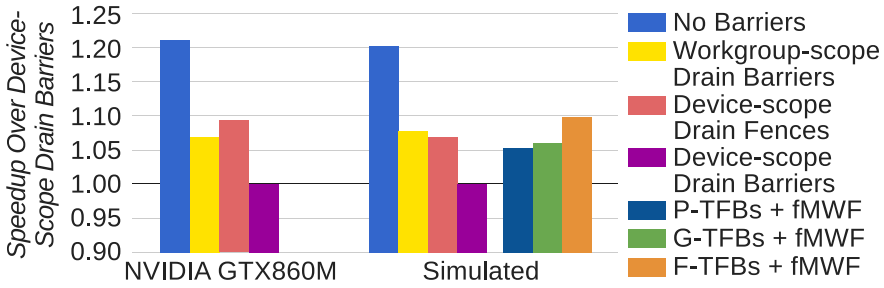


Figure 5.16: Geometric mean speedups for TFBs around device-scope barriers normalized to device-scope drain barriers.

data by executing all applications using PBS-generated binaries that include device-scope barriers. While device-scope barriers provide unnecessarily strong memory ordering guarantees for these applications, they help compare workgroup- and device-scope barrier overheads.

Figure 5.16 plots geometric mean speedups of NVIDIA GTX860M PBS binaries and the simulated TFB implementations in the presence of device-scope barriers, normalized to device-scope drain barriers. TFBs hide roughly half of barrier slowdown on average. P-TFBs and G-TFBs provide speedup of 1.05×. F-TFBs recover more load imbalance near device-scope barriers, providing 1.10× speedup.

TFBs recover more performance from device-scope barriers than workgroup-scope barriers. Device-scope fences can be up to 5× longer latency than workgroup-scope fences, and device-scope acquire fences can upset cache hit rates by clearing stale data from L1 caches. In some cases, TFBs hide elevated device-scope fence latency by bursting instructions from the BDIB, and restoring MLP lost to decreased cache hit rates.

5.7.5 Discussion

Overall, drain barriers upset progress divergence (load imbalance) between warps that can be desirable. As warps diverge, some expose more potential parallelism by running ahead to issue instructions to different execution pipelines (ALU, SFU, or memory) and improve occupancy. Unfortunately, drain barriers cause warps to bunch up, decreasing divergence

and occupancy. Warp schedulers could try to reduce divergence leading into drain barriers, but occupancy loss cannot be eliminated. Instead, TFBs expose more instructions that warp schedulers can optionally issue during barriers to maintain occupancy and an appropriate amount of warp progress divergence. Results above show that with reasonable logic or storage changes, TFBs can provide efficiency gains, even as emerging systems may incur greater synchronization overheads.

5.8 Related Work

Relevant prior work has investigated fences and barriers in both CPUs and GPUs, and TFBs target performance challenges similar to GPU warp scheduling, CPU out-of-order designs, and speculation techniques.

Prior work has proposed more aggressive CPU and GPU **fence implementations** than drain fences. Aggressive CPU fence implementations [20, 85, 86, 37] send fences along with memory accesses to load-store units and caches, which must enforce access ordering. These often require substantial access buffering or speculative state. Such designs have been overlooked for GPUs in favor of more flexible, but intricate consistency models that allow storage-efficient fence designs (e.g., Hechtman et al., QuickRelease [53]). For GPU computing, however, barriers are more frequent than fences, so fence optimizations are just likely to increase the importance of optimizing load imbalance near barriers, like TFBs.

Plenty of prior work proposes **hardware barrier implementations** for CPU multicore and multiprocessor systems. These works have focused on combining logic to test barrier completion [16, 132], and large-scale system interconnects [45, 15, 80, 35, 5, 33] and virtual networks [154, 128, 127] to distribute barrier signals. Our work looks at single-chip GPUs, where threads participating in barriers are executing on the same core and signals can be localized. Prior GPU studies have built upon GPU hardware barriers and compared performance to software implementations [29, 153, 140, 12], but have not evaluated changes

to barrier microarchitecture. For potential future work, barriers could be constructed from non-draining fences in an effort to hide fence latency. TFBs focus on core microarchitectures to expose and execute parallel work to hide barrier load imbalance.

Similar to **warp scheduling** techniques [43, 99], TFBs effectively shuffle instruction ordering across a set of active warps on a core. Most warp scheduling work has focused on cache locality [123, 64], which often encourages subsets of warps to run ahead of others. Other scheduling techniques, like Criticality Stacks [36] and Criticality-Aware Warp Scheduling [78], aim to prioritize critical threads to minimize occupancy loss near synchronization events. As noted above, TFBs may ease the tension between these classes of techniques, and compared to sophisticated warp scheduling, they are a low complexity option

Like many CPU core designs, TFBs in GPUs have a flavor of out-of-order issue, since they allow instructions to be reordered before entering execution pipelines. Since CPU cores must mine instruction-level parallelism from few threads, many aggressive **out-of-order and speculation** techniques have been tested to reduce synchronization overheads (e.g., [91, 136, 98, 22]. These techniques usually require complex pipeline designs or high-overhead checkpointing mechanisms to roll-back state in the event of misspeculation. Such designs are prohibitively expensive in GPUs due to their numerous threads, which can already provide significant parallel work to hide synchronization latency. GPU speculation techniques have been proposed to provide exception and virtualization support [87, 93], but have not been considered for reducing synchronization overheads. We recommend speculation as a potential area for future work.

Finally, we believe that this work provides the first formalized analysis of instruction ordering flexibility with architected barriers. Prior work has introduced barriers into the instruction set as architecturally visible [48, 40, 128, 155], but much of this work preceded formal models of relaxed memory consistency and architectural side-effect visibility. Other prior work discusses the complexity of implementing barriers in the context of a relaxed

consistency CMP [126, 82], but they do not consider fuzzy barriers.

5.9 Summary

Memory consistency in modern parallel processors like GPUs requires primitives such as fences and barriers. These synchronization primitives can cause significant overhead for existing applications, and we expect overhead to increase as heterogeneous memory system consistency and coherence models evolve. We show that load imbalance is the most significant cause of lost occupancy near barriers, and we propose Transparent Fuzzy Barriers (TFBs) that enable threads to execute independent instructions past a barrier while still adhering to architectural semantics. TFBs require no changes to applications and modest hardware changes to offer average speedup of 1.061× and up to 1.14×.

In addition to the TFB benefits for common existing GPU barrier use, this work shows that TFBs have benefits for future-looking applications. Chapter 4 showed that applications targeted at heterogeneous processors should use barriers to reduce synchronization granularity and synchronize data across GPU cores and even to CPU cores. These barriers will need to include broader scope fences. This chapter shows that TFBs provide even greater performance gains for device-scope fences used in these CPU-GPU coordinated work applications.

6 Q-CACHE: PRODUCER-CONSUMER COMMUNICATION IN CACHE

As we develop applications to execute producer and consumer tasks concurrently, we encounter core and cache utilization challenges that can limit expected performance and energy gains when running in heterogeneous processors. As described in Section 4.2, we run producers and consumers concurrently to overcome the cache spills caused by the coarse-grained nature of GPU application pipeline stages. Unfortunately, under concurrent producer-consumer activity, cache spills can still occur if consumers are unable to keep up with producers. Further, when consumer tasks pull producer results more quickly than producers generate them, they can cause energy inefficient execution while waiting for producer data to become available.

To address these challenges, producer and consumer tasks must match their cache access rates to keep intermediate data in cache and to limit consumer stalling. In this chapter, we investigate existing and emerging software transformations in search of general-purpose and efficient task management techniques to capture producer-consumer communication in cache. While software transformations can reduce cache spilling, they often place heavy burdens on the programmer to organize task communication and estimate dirty data footprint in cache.

In response to these complexities, we propose Q-cache, a hardware technique to support concurrent producer-consumer communication in cache by throttling producers or consumers using emerging core, task, and frequency management mechanisms. Compared to software techniques, applications supported by Q-cache have better performance and energy characteristics without the need for complicated software transformations or explicit buffered data management.

6.1 Introduction

Although GPU architectures have evolved more general-purpose producer-consumer capabilities, GPU computing applications are often structured as numerous software pipeline stages that implicitly synchronize data on kernel boundaries between stages. Unfortunately, kernel boundary synchronization is often cache-inefficient. When processing a data structure larger than cache capacity, producer kernels push their intermediate results out of cache to off-chip memory before they can be consumed by a following kernel. For open-source GPU computing workloads, producer-consumer cache spills cause an average of 15% and up to 34% excess off-chip memory accesses.

While software techniques can reduce cache spilling, they often place heavy burdens on the programmer. Techniques like kernel fusion and fission force programmers to explicitly map producer-to-consumer communication and ensure correct ordering. For emerging applications with irregular data structures or memory accesses, it is difficult or impossible to statically organize such mappings.

An emerging software technique, task queues (or "worklists"), offers programmers a general-purpose structure for mapping producer-to-consumer communication. As producers generate outputs, they signal that the outputs are ready by pushing a task into the queue. Any consumer can process any producer's task, eliminating the need for explicit producer-to-consumer mapping. With queues, producers and consumers can execute concurrently, potentially communicating in cache.

Though task queues simplify producer-consumer communication, they do not provide general-purpose support for managing that communication in cache. During a concurrent producer-consumer relationship, programmers must explicitly estimate and control the dirty data footprint to avoid spilling it from cache. Estimating cache footprint in software is often hard and necessarily conservative, resulting in cache spills or underutilized cache and compute resources.

To ease the programmer's burden for managing data in cache, we propose a novel

hardware technique, called Q-cache. Q-cache measures the cached data footprint and throttles producer or consumer tasks when buffered data might start spilling from cache. By improving cache footprint estimates, Q-cache provides average performance gains of 1.20× over the best software-only algorithm implementations. Further, Q-cache offers the best energy characteristics by reducing cache spills and trimming unnecessary core activity.

This chapter makes the following contributions:

- The first comparison of kernel fusion with emerging concurrent producer-consumer software queuing for a diverse set of workloads.
- Observes that software queue management in cache is often limited by necessarily conservative dirty data footprint estimates.
- Proposes and evaluates Q-cache, a hardware technique to dynamically track queued data in cache and throttle producer-consumer access rates to avoid spilling data from cache.

The chapter is organized as follows: Section 6.2 motivates the study of efficient producer-consumer communication. Section 6.3 describes the complexities of software queuing in cache, and Section 6.4 describes Q-cache, a hardware technique to simplify queuing in cache. Section 6.5 explains our testing methodology, and Section 6.6 evaluates the producer-consumer techniques. Section 6.8 discusses related work, and Section 6.9 concludes.

6.2 Background and Motivation

GPU computing applications use producer-consumer communication pervasively, and communication is diversifying in emerging applications. To avoid programming complexity, programmers often construct simple software pipelines that synchronize data on GPU kernel boundaries, a structure we call “kernel synchronization”. Kernel synchronization

often causes producer-consumer intermediate data to spill from cache, resulting in up to 34% extra off-chip memory accesses and eroding performance and energy efficiency.

For producers and consumers to pass data within cache live times, they must run in close temporal proximity. This section describes software techniques that aim to offer temporal proximity: kernel fusion and fission, and software task queues. Unfortunately, these techniques place heavy burdens on the programmer to organize producer-consumer mappings or explicitly manage cached data footprint. The growing parallelism and memory access diversity in emerging GPU computing workloads indicates a desire for more general-purpose and efficient support for producer-consumer communication within caches.

6.2.1 Existing Producer-Consumer Nature

Producer-Consumer Diversity is Growing: Producer-consumer communication is pervasive in GPU computing applications. We survey 58 GPU computing applications from the Lonestar [21], Pannotia [23], Parboil [138], and Rodinia [24] suites, and we find that 51 applications (88%) are structured as software pipelines with multiple producer-consumer relationships. Nearly all use kernel synchronization (50 of 51): producer and consumer tasks run in separate, serialized application stages.

Producer-consumer relationships span a broad spectra of thread handling and memory access characteristics. Most Rodinia and Parboil application kernels process regularly structured data arrays and use one GPU thread per input datum. Frequently, there are clear mappings from producer output data to the consumer threads that process them, and the GPU can often perfectly coalesce memory accesses. As we describe below, software transformations can help many of these regular producer-consumer structures keep data in cache.

On the other hand, emerging GPU computing workloads show increased need for more general-purpose and efficient producer-consumer communication. GPU computing APIs and hardware have become more general-purpose, encouraging applications to use more

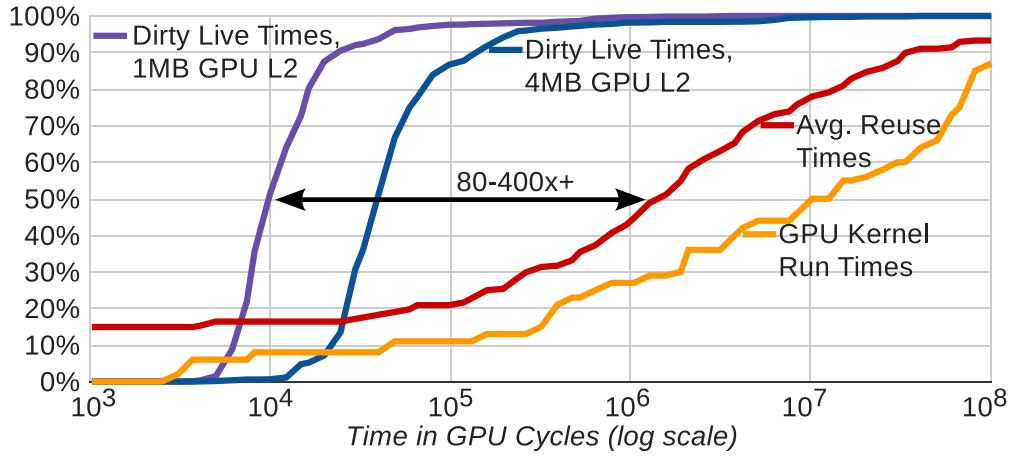


Figure 6.1: 46 GPU computing applications: Cumulative distributions of GPU kernel run times, data reuse times, and L2 cache dirty data live times in GPU cycles.

irregular data structures and memory accesses, like those in Lonestar and Pannotia suites. These applications often involve iterative data traversals that result in very deep software pipelines and repeated cache spills. The producer-consumer relationships are dynamic and data-dependent, making it difficult or impossible to explicitly manage cached data strictly in software.

Kernel Synchronization Causes Costly Spills: Kernel synchronization causes the most producer-consumer spills in GPU computing applications due to the long time between when data is produced and consumed. Kernel synchronization results in up to 34% extra off-chip memory accesses, reducing performance and energy efficiency. To characterize cache behavior caused by kernel synchronization, we collected and analyzed memory access traces from a simulated GPU running 46 applications from the four suites listed above. The simulated GPU contains 16 NVIDIA Maxwell-like cores, either a 1MB or 4MB L2 cache, and GDDR5 memory with peak off-chip bandwidth of 179GB/s. Over the 46 applications, Figure 6.1 plots cumulative distributions of average kernel run times, data reuse times (write to read), and L2 cache dirty data live times in GPU cycles.

The data show that kernel synchronization causes data reuse times to dominate data live times in cache. While a decent portion of kernels reuse data completely in cache (roughly

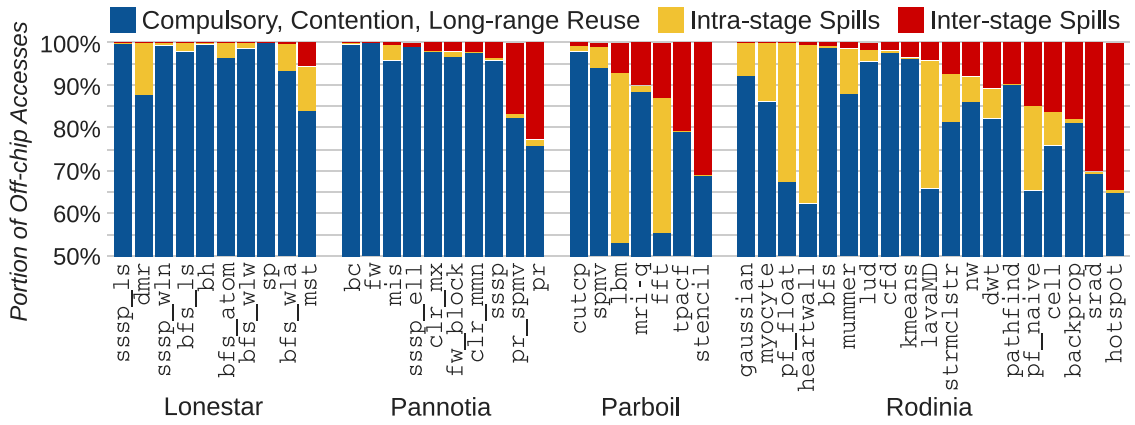


Figure 6.2: Off-chip memory accesses breaking out producer-to-consumer spilled accesses within pipeline stages and from one stage to the next; 1MB L2 cache.

18%), the data reuse time distribution in red correlates strongly with kernel run times in orange ($r = 0.96$), indicating that producer data is very often reused in the next kernel rather than within a kernel's run time. In stark contrast, dirty data cache live times are extremely short. Their distributions have medians 80–400$\times$ shorter than data reuse times: just 9,000 GPU cycles for 1MB L2 cache (purple), and 37,000 cycles for a 4MB L2 (blue).

As expected, the gap between live times and reuse times causes substantial producer-consumer cache spills. For the 46 applications, Figure 6.2 breaks down the portion of off-chip memory accesses resulting from cache spills. "Intra-stage Spills" are off-chip reads that occur after a prior cache line writeback during a single software pipeline stage, while "Inter-stage Spills" occur across subsequent pipeline stages. In total, spills cause an average of 15% of off-chip memory accesses, and kernel synchronization specifically causes up to 34%.

Although the plot shows spills are a smaller portion of accesses in irregular applications, like Lonestar and Pannotia, we expect spills to become more important in future revisions of these applications. Currently, these applications are under-optimized and have significant cache contention from large working sets and poor memory access coalescing. Recent studies show optimization can greatly reduce these issues [152, 101].

Overall, the temporal distance between producers and consumers must be reduced

dramatically to capture data reuse in cache. Kernel synchronization must be replaced with finer-grained data signaling.

6.2.2 Software Producer-Consumer Techniques

A few software techniques can move producer and consumer tasks into closer temporal proximity, but their capabilities are limited. Here, we describe the general techniques, kernel fusion, fission, and concurrent producer-consumer execution using software queues. While each can improve producer-consumer data caching, they force programmers to wrestle with either mapping producer-to-consumer communication, or managing cached data footprint.

For each technique, Figure 6.3 diagrams examples of mapping producer and consumer threadblocks (OpenCL "workgroups") to GPU hardware contexts over application run time. The baseline is kernel synchronization (Figure 6.3a), in which the producer kernel runs to completion before the consumer, possibly spilling intermediate data from cache.

Kernel fusion hoists consumer tasks into close temporal proximity with their producers by statically joining the code [146, 145], but is limited to simple producer-consumer mappings and resource requirements. Figure 6.3b shows consumer code fused into the producer kernel so it runs immediately after the producer tasks in the same threadblock. This organization requires producer-to-consumer mappings to be onto, so that all consumer tasks run. Further, since GPUs do not guarantee thread ordering, the programmer must ensure that consumer tasks always run after the producers that generate their data. These restrictions basically limit fusion to be used for statically-determined, one-to-one producer-consumer mappings. Finally, kernel fusion does not explicitly control cached data, so data might still spill from caches.

Kernel fission is a transformation that divides GPU kernels into chunks and runs them in a time-multiplexed manner [151], but is limited similarly to kernel fusion. Figure 6.3c shows the baseline kernels divided into chunks and executed so that consumers run just

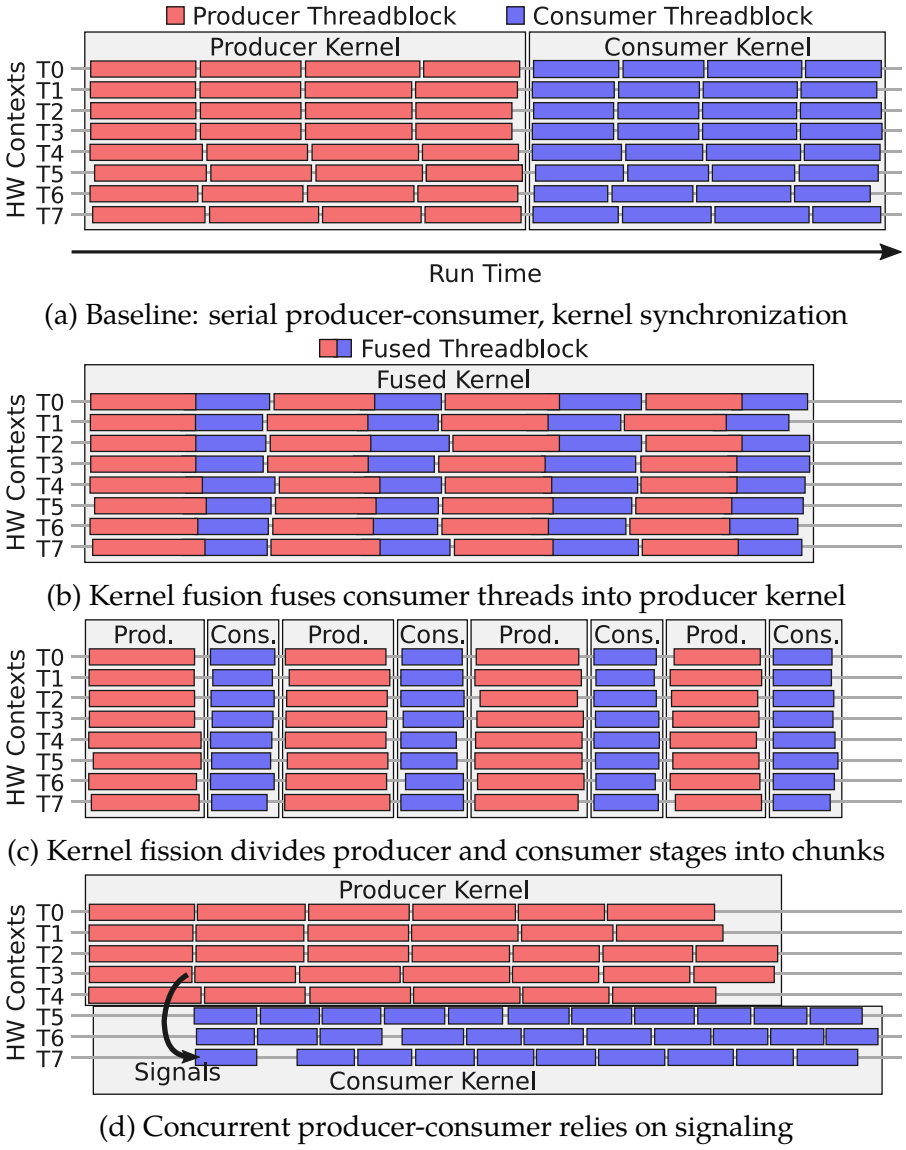


Figure 6.3: Software transformation producer, consumer threadblock mapping to GPU hardware contexts.

after their producers. Fission still uses kernel synchronization, but the granularity of these synchronizations is designed to keep data in cache. By serializing kernel chunks, producer-consumer mappings can involve many-to-one communication, which is more flexible than fusion. However, fission still forces the programmer to guarantee task ordering, and worse, to explicitly manage dirty data size of each producer kernel chunk. CUDA dynamic parallelism [103] is a more advanced variant of kernel fission, but comes with similar limitations [147].

Concurrent producer-consumer execution is a technique that launches producer and consumer tasks to execute concurrently, and consumer tasks wait for producers to signal when data is ready. Figure 6.3d shows an example of producer and consumer threadblocks executing concurrently on separate GPU hardware contexts using concurrent multikernel support [148].

Unlike kernel fusion and fission, producers must explicitly signal when their outputs are ready. An emerging technique in GPUs to support producer-consumer signaling is the software task queue (or "worklist"). Software queues do not require explicit producer-to-consumer mappings, because producers dynamically allocate queue space, and any consumer can pull and process any task in the queue. We describe queues in detail in the next section.

6.3 Queue Management in Cache

Concurrent producer-consumer execution with software queues offers a more general approach for signaling between producers and consumers compared to kernel fusion and fission. Unfortunately, as our results show, just running producers and consumers concurrently does not guarantee that consumers are able to keep up with producers, so intermediate data may still spill from cache. The major challenge with software queues is to make sure that producers do not run too far ahead and push intermediate data out of cache.

This section details the challenges and constraints of finite queuing in cache. As we demonstrate with example code, using software to manage the amount of cached data is hard for a number of reasons, and the resulting code is not general-purpose or efficient. We would instead prefer hardware to manage producer and consumer accesses to cache.

Baseline: Software Finite Queuing To concretely discuss queue management in cache, we describe a concurrent producer-consumer algorithm implementation that uses software

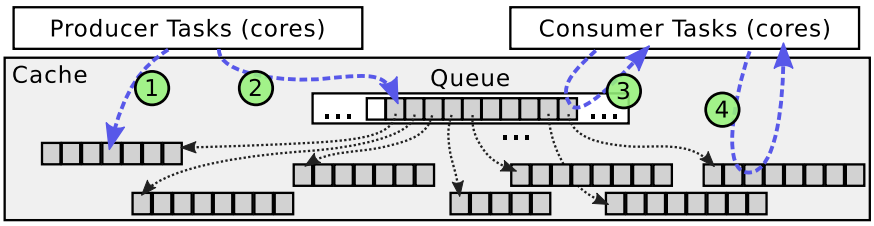


Figure 6.4: Logical design of queuing in cache.

finite queuing. Figure 6.4 shows the logical design of finite queuing in cache, and Figure 6.5 shows CUDA pseudo-code for the concurrent producer-consumer relationship. To start, the host thread executes the `main` function (lines 41–44), setting up data structures, and then launching the producer and consumer tasks to run concurrently.

Once running, computation proceeds as follows. Producer threadblocks generate intermediate data in chunks (1) and then push tasks pointing to the chunks into the queue (2). In lines 26–28, the producer threads synchronize to ensure their data chunk is visible to consumers before signaling the queue. Consumer threadblocks pull tasks as producers signal them (3), and then consume the task's referenced data (4).

Backpressure: In this structure, if producer tasks might generate data more quickly than consumers can pull it, some form of backpressure is required to keep producers from pushing data out of cache. For software finite queuing, backpressure involves explicit code that blocks producers from running when they estimate that the cache might be full. Each producer threadblock starts with the 0th thread trying to allocate cache space (line 22). If it finds enough space for the block's data, its threads proceed immediately to generate their results. On the other hand, if the 0th thread finds the queue full, it must pause the rest of the block's threads to wait for cache space. The function call, `q_wait_for_space` (line 8), estimates the amount of queued data in cache and does not return until enough space is available for the threadblock's data. Consumer threads must free queue space with the `q_dealloc` function (line 39), and wake up waiting producer blocks.

Unfortunately, estimating a queue's cache footprint is hard, so programmers will likely prefer static, conservative queue capacity estimates. For example, a memory- and compute-

```
1  void **queue; // Initialized to null pointers
2  int q_head = 0, q_next = 0, q_tail = 0;
3
4  // Allocate position in queue
5  __device__ int q_alloc() {
6      int index = atomicInc(&q_head);
7      // Wait for enough capacity in queue
8      q_wait_for_space(index);
9      return index; }
10
11 // Signal specified task is ready
12 __device__ void q_push(int index, void *ptr);
13
14 // Wait for the next queued task, then return it
15 __device__ void *q_pull();
16
17 // Deallocate a consumed element
18 __device__ void q_dealloc() { atomicInc(&q_tail); }
19
20 __global__ void producer(Data *input, Data *output){
21     int index;
22     if (threadIdx.x == 0) index = q_alloc();
23     // Wait for 0th thread to allocate cache space
24     __syncthreads();
25     process(input, output); (1)
26     __threadfence(); // Device-scope fence
27     // Wait for all block results to be written
28     __syncthreads();
29     if (threadIdx.x == 0)
30         q_push(index, &output[blockIdx.x]); (2) }
31
32 __global__ void consumer(Data *input) {
33     __shared__ void *chunk_ptr;
34     if (threadIdx.x == 0)
35         chunk_ptr = q_pull(); (3)
36     // Wait for 0th thread to share chunk_ptr
37     __syncthreads();
38     consume(input, chunk_ptr); (4)
39     q_dealloc(index); }
40
41 int main() {
42     // Set up, launch concurrent producer, consumer
43     producer<<<N, S1>>>(input, intermediate);
44     consumer<<<N, S2>>>(intermediate); }
```

Figure 6.5: Example CUDA code structure for concurrent producer-consumer activity. Kernels launch on separate streams, S1 and S2, to indicate parallelism. Highlighted code is extra required for software finite queuing.

efficient way to limit the queue's footprint is to count queued tasks and multiply by the average data accessed by each threadblock that generates a task. Equation 6.1 shows this footprint calculation, for which the programmer must estimate reads and writes per threadblock by statically counting them in the producer and consumer code.

$$\text{footprint} = (\text{q\_head} - \text{q\_tail})(\text{reads} + \text{writes per block}) \tag{6.1}$$

More accurate estimates would require dynamic counting of reads and writes per block, which would incur extra communication and run time overhead.

Due to the complexity of estimating dirty data footprints, software finite queuing fails to provide general-purpose producer-consumer communication in cache. Different applications and sometimes even different inputs require tuned footprint estimates, and calculations may need to be adjusted for hardware with different cache configurations. Further, many dynamic factors can affect cached data live times. Rather than placing these burdens on the programmer, we would prefer that hardware assist the software in making efficient use of caches for producer-consumer activity.

6.4 Q-cache Architecture

To address the cached data footprint challenges, we would like to imitate software finite queuing by instead using hardware to throttle producer and consumer cache activity. The key insight that we draw from software finite queuing is that producers and consumers should progress with roughly even rates; to avoid data races, consumers should not pull data from cache before than producers generate it and signal it is available. Finite queues limit producers from running too far ahead of consumers, requiring that when the queue is full, producers must generate results at the same rate that consumers pull.

To match producer and consumer cache access rates in hardware, we propose Q-cache, a novel technique that measures and throttles producer and consumer access to cached data. The design adds small cache logic and state that track the amount of cached data, and leverages existing coarse-grained core resource scaling to adjust producer and consumer progress rates. With Q-cache, programmers need not write the highlighted code in Figure 6.5, eliminating painful data footprint estimations. Here, we describe the microarchitecture and rate management policies, which are sufficient to adeptly capture a broad range of producer-consumer behaviors.

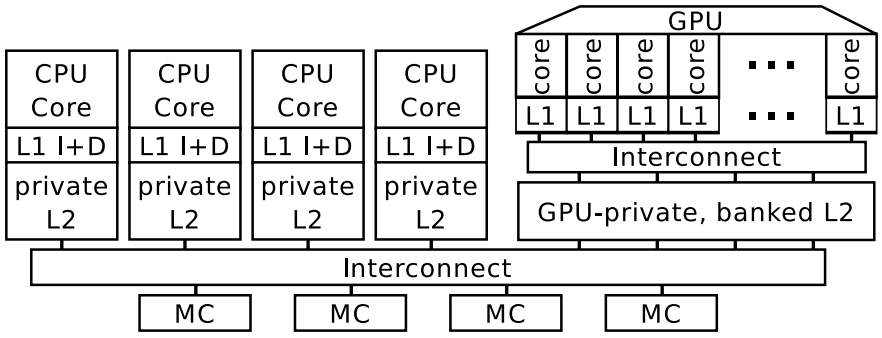


Figure 6.6: Heterogeneous CPU-GPU architecture.

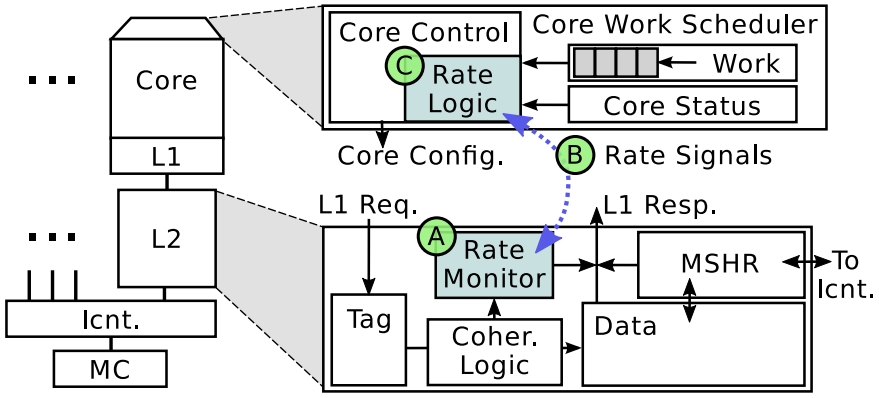


Figure 6.7: Logical microarchitecture of Q-cache: Rate monitor measures producer-consumer communication and signals core schedulers to throttle cache activity.

6.4.1 Overview of Q-cache

Our target system for Q-cache is a heterogeneous CPU-GPU processor as diagrammed in Figure 6.6. It contains 4 CPU cores and a 16 core GPU. The different cores can communicate through a cache coherent, unified memory space. This target represents a potential future heterogeneous processor.

Within this target processor, Q-cache adds minimal logic, storage, and messaging to manage producer-consumer communication, highlighted in blue in Figure 6.7. Specifically, to each of the L2 caches, Q-cache adds a rate monitor (A), which tracks producer-consumer behavior. When the rate monitor detects that a producer-consumer relationship requires throttling to keep data in cache, it sends a rate scaling request message to producer or consumer cores (B). Finally, core controllers can choose to throttle producer or consumer

activity based on the scaling request (C). Together, these components ensure that producer-consumer data rarely spills from cache.

6.4.2 Q-cache Rate Monitor

In the target system's L2 caches, we add a small bit of logic and state, called the rate monitor (A), to observe data production and consumption. The rate monitor is responsible for the following: detecting and tracking producer-consumer relationships, periodically checking dirty cache footprint and comparing rates, and signaling cores when rates should be adjusted.

Detecting producer-consumer relationships: When a group of producer or consumer tasks begins executing (e.g., GPU kernel), the work scheduler signals to enable the rate monitor. While enabled, the monitor observes coherence state transitions to detect when data is being produced or consumed from the cache. If the monitor detects both data production and consumption, it enters an "active" state, during which it measures producer-consumer activity. Otherwise, the monitor can disable itself if no relationship is detected. (e.g., after some number of cycles or amount of activity). Disabling the monitor ensures that it does not cause spurious compute rate adjustments.

Tracking communication: While in the "active" state, the rate monitor counts cache lines that are communicated between producers and consumers by observing coherence state transitions. Specifically, if a line's state transitions from invalid, shared, or exclusive to modified, the line is potentially an output from a producer task, so the monitor increments a produced data counter. Similarly, when a request hits a modified cache line, it is sent to the requester, and the monitor increments a consumed data counter. To completely track the producer-consumer dirty data footprint, the monitor also counts when modified cache lines are spilled from cache (i.e., writeback). As a fail-safe, if the monitor detects that results are spilling from cache too quickly to be consumed, it can simply disable itself to avoid disrupting the producer or consumer progress. Finally, the monitor tracks the number of

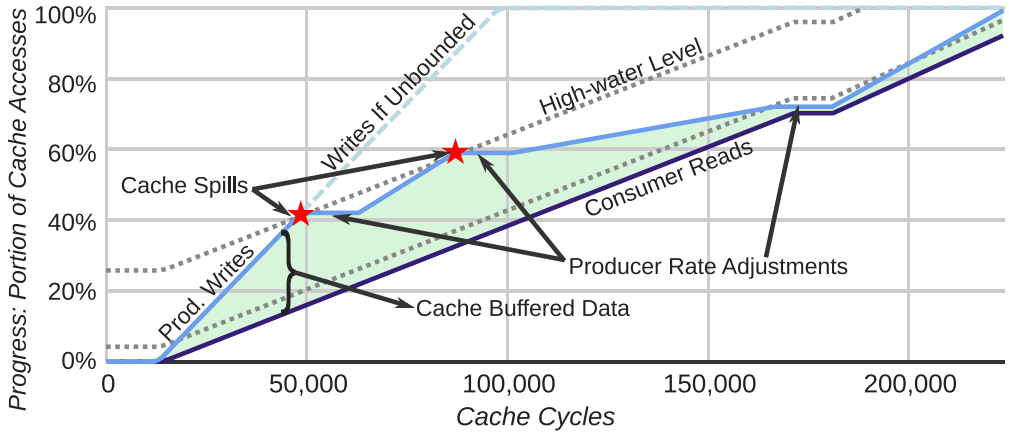


Figure 6.8: Example Q-cache operation: the rate monitor detects that the producer write rate is greater than the consumer read rate and adjust the producer rate twice. If it slows producers too far, it can detect when the consumers catch up and speed up the producer.

cycles since it saw the first producer and consumer activity, so it can estimate their access rates. It resets activity and cycle counts periodically and after rate change events to ensure they represent recent behavior.

6.4.3 Rate Change Policies

Using activity and cycle counts, the rate monitor periodically estimates whether the producer-consumer relationship is operating efficiently in cache. If not, it will send rate scaling requests to either the producer or consumer core schedulers. For efficient cache activity, the rate monitor has three goals. First, it should aim to keep producer and consumer rates matched to avoid spilling data from cache. Second, producers should run as quickly as possible to avoid causing consumers to block waiting for results. Finally, to optimize the available cache capacity, the rate monitor should try to keep the dirty data footprint small, but sufficiently large to keep producers active and avoid stalling consumers.

To demonstrate example Q-cache operation, Figure 6.8 shows the producer write and consumer read progress during the run time of a short kernel that employs Q-cache to throttle their rates. Table 6.1 lists the generic rate change policies employed by the monitor. To estimate the producer's rate (R_P), the monitor divides the produced cache lines by the

Table 6.1: General rate change policies.

P-C Rates	Queued Data (Q)	Spills	Rate Change Action
$R_P > R_C$	–	–	Speed consumer/Slow producer
$R_P = R_C$	> 0	Y	Speed consumer/Slow producer
$R_P = R_C$	> 0	N	Do nothing
$R_P \leqslant R_C$	$\rightarrow 0$	–	Speed producer/Slow consumer

number of cycles since the producer started generating results. Similarly, it estimates the consumer's rate (R_C). Finally, Q is the current dirty data footprint, calculated as the number of produced lines minus consumed and spilled lines.

The rate monitor must take care when estimating rates of bursty GPU producers or consumers. When GPU threadblocks start executing, they send numerous concurrent memory accesses to cache [54]. The rate calculation interval must be long enough to amortize bursts, but short enough to keep data in cache. Empirically, we find calculation intervals shorter than data live times can sufficiently smooth 16-core GPU bursts, and we test the effects of different memory access characteristics in Section 6.7.

If the monitor detects the producer running ahead of the consumer, it should send a request to scale rates. The monitor may detect that the producer's rate is greater than the consumer's, or if the rates are nearly the same, it might detect light data spilling from cache during periods when the queued data footprint is near the maximum cache capacity. Such spilling likely indicates that the producer is moving ahead of the consumer slowly. Given the aim of running producers as quickly as possible, the monitor prefers to request that consumers speed up before slowing down producers.

If the monitor detects a faster producer early during a producer-consumer relationship, it may want to delay sending rate scaling requests for a couple reasons. First, by allowing more dirty data to buffer, the monitor affords itself time to make more accurate decisions and if necessary, to scale back producer rates while minimally blocking the consumer. Second, by waiting for the first spills before sending a request, the monitor can estimate the maximum Q allowable during a producer-consumer relationship, and use it to avoid

future spills.

If the monitor detects similar producer and consumer rates, a reasonably large Q, and little spilling, it assumes that the rates are "locked" and does not make any requests. We find that the rate monitor commonly reaches this range with three or fewer rate changes, even under very asymmetric and dynamic rates.

Finally, the monitor detects if consumers are often stalled by observing the amount of queue data in the cache over time. If Q remains low (i.e., within some margin of zero), it is likely that the consumer's rate would exceed the producer's if the consumer were not waiting for queue signals. In this case, the monitor can request the producer speed up or consumer slow down.

6.4.4 Signals Between Cores and Rate Monitor

For rate management, Q-cache requires minimal messaging through existing cache hierarchy paths, depicted by (B) in Figure 6.7. As noted, the GPU sends messages to the rate monitor to delineate kernel beginning and end. These messages could be inferred from existing messages to manage GPU caches around kernel boundaries (e.g., for flushing stale data or TLBs).

Other messages aim to manage producer and consumer rates. The rate monitor sends rate asymmetry estimates to appropriate cores as a request for a rate change. These requests can be treated simply as hints that the compute core could aid efficient use of cache through a rate change. If a compute core accepts a rate change request, it responds with a message after reconfiguration indicating the change to the rate monitor, which resets its counter state.

Overall, these messages constitute a trivial portion of total memory traffic (0.02% in the worst case), and could even be eliminated during application phases that are known to not involve producer-consumer concurrency.

Table 6.2: Frequency scaling used to evaluate Q-cache.

Component	Parameters
(4) CPU core + L1 cache	800 – 3500 MHz with 6 P-states
(1) GPU + L1 caches	200 – 700 MHz with 5 P-states
Frequency change	15 – 45 µs delay, Park et al. [114]

6.4.5 Modulating Producer and Consumer Rates

Prior work and existing architectures have laid substantial groundwork for modulating compute rates for power, energy, and system-level quality of service purposes. To test Q-cache, we consider two orthogonal techniques to modulate producer and consumer rates as controlled by core work schedulers (C).

First, we enable dynamic voltage and frequency scaling [130] with parameters in Table 6.2. The processor contains independent clock domains for each CPU core + L1 cache and the GPU. On the CPU side, the operating system or hardware might adjust frequency based on core activity. As in state-of-the-art GPUs, we model a microcontroller that manages work distribution to cores and GPU-wide frequency [104, 113].

Unfortunately, frequency scaling only offers a small dynamic range on compute rates (roughly 6:1 for current CPUs and 3:1 for GPUs), but in practice, we find producer-consumer rate asymmetries upwards of 10:1. To offer greater dynamic range, we also model task preemption and core disable on the GPU. Specifically, the GPU can drain threadblocks from a core and then disable empty cores [141]. Core disable provides greater rate dynamic range (e.g., 16× range with 16 GPU cores), and task preemption frees cores to reduce static power consumption, or to be rescheduled for other purposes, such as adding producer or consumer threads. Although automatic CPU thread handling could be provided through various means [100, 26], we do not test it in this study. Instead, we manually scale CPU thread count to exercise a range of producer-consumer rate asymmetries.

Table 6.3: Heterogeneous processor specifications.

Component	Parameters
CPU Cores	(4×) 4-wide out-of-order, x86 cores, 3.5GHz
CPU Caches	Per-core 32kB/32kB L1I/D and exclusive, private 256kB L2 cache, 128B lines
GPU Cores	(16×) Maxwell-like SMMs (2×SM), 700MHz, 2×2 GTO [123] warp schedulers, up to 32 threadblocks, 64 warps of 32 threads, 96kB scratch memory, 64k registers
GPU Caches	Per-core 32kB L1, GPU-shared, banked, 2MB non-inclusive L2 cache, 128B lines, in-cache atomics (Kepler/Maxwell throughput)
Interconnect	GPU L1/L2: Dance-hall. All L2s/MCs: High-bandwidth, 12-port switch
Memory	(4×) GDDR5 channels, 179 GB/s peak

6.4.6 Overhead of Q-Cache

Q-cache requires minimal extra storage to track the production and consumption rates. The rate monitor maintains a bit-vector of active cores, its current state, and the five registers noted previously: (1) count of dirty lines produced, (2) count of dirty lines consumed, (3,4) one each for cycles since producers and consumers started accessing cache, and (5) current dirty data footprint. In total, Q-cache adds just 41B per L2 cache.

Logic and energy requirements of Q-cache are also minimal. The rate monitor observes current and next states of coherence transitions, so small logic additions check whether transitions should be recorded. Rate calculations are infrequent and latency-insensitive, so logic can be optimized for minimal area and energy.

6.5 Methodology

6.5.1 Target Heterogeneous Processor

Heterogeneous CPU-GPU processors support a wide range of potential producer-consumer activity. CPU or GPU cores can perform better under different application characteristics [54]. Both core types could run producers or consumers during different application pipeline stages, and emerging applications are likely to make use of both core types to fully utilize core resources. GPU-producer–CPU-consumer relationships are of particular interest, because wide-parallel producer stages often narrow the amount of work for

subsequent consumer stages.

We simulate multiple parallelization schemes, described below, within the heterogeneous CPU-GPU processor previously in Figure 6.6. Table 6.3 lists this processor's specifications. CPU and GPU cores are configured comparably to existing, state-of-the-art processors. They share unified virtual and physical address spaces, and the CPU caches and the GPU L2 cache are coherent to permit fine-grained communication.

Cache Hierarchy: Each CPU's cache hierarchy includes private, 32kB L1 instruction and data caches, and a private, 256kB L2 cache. Similar to NVIDIA Fermi caches, GPU L1 caches are write-through, disallow dirty cached data, and allow stale data versions. The GPU L2 cache and all CPU caches implement full MOESI directory coherence. All caches use pseudo-LRU replacement. The GPU L2 is banked to provide aggregate bandwidth similar to current GPUs. Each L2 cache has a Q-cache rate monitor, so the GPU L2's rate monitor aggregates activity from all banks.

Interconnects and Memory: We model two simple interconnects for inter-cache communication, and buffering and bandwidths are consistent with existing GPU interconnects and switches. Comparable to bandwidths expected from forthcoming memory technologies [116, 60], four GDDR5 memory controllers provide aggregate bandwidth of 179 GB/s to off-chip memory.

6.5.2 Tested Applications

To evaluate a broad range of application characteristics, we port a diverse set of applications from the NVIDIA code samples [107], Pannotia, and Rodinia suites to use parallel producer-consumer relationships. The applications listed in Table 6.4 each contain existing producer-consumer communication that causes intermediate data to spill from cache. Since the CPU and GPU share virtual and physical memory spaces and the target processor provides coherence, the different cores share memory allocations and GPU memory copies are removed. Aside from kernel fusion implementations, we parallelize each application to

Table 6.4: Applications modified to evaluate Q-cache.

Suite	Bench	Data Access	P-C Map	CPU Threads	R:W Ratio
Rodinia	`backprop`	Regular	Bijection	1	18:1
Pannotia	`color` [31]	Irregular	M:1	1–2	1.3–200:1
Rodinia	`kmeans`	Regular	1,M:1	1–2	1–15:1
NVIDIA	`reduce`	Regular	Bijection	1–4	2–64:1
Rodinia	`strmclstr`	Irregular	1,M:1	1–2	5.6–16.8:1

run a GPU producer kernel concurrently with CPU consumer threads.

These ported applications have a range of characteristics that can be loosely classified into three groups. First, `backprop` and `reduce` contain regular producer-consumer communication in the form of reductions. Reductions can be performed on flat memory arrays in a regular manner, and the GPU producer can perfectly coalesce memory accesses. These applications permit a static bijection from producer to consumer tasks, making it simple to use kernel fusion.

Second, `kmeans` and `strmclstr` contain conditional communication between producers and consumers. `Kmeans` consumers perform a reduction-like operation, while `strmclstr` consumers independently process each producer output. Memory access is mostly regular, and most communication can be statically mapped from producer to consumer, but both applications contain data-dependent behavior that requires many-to-one producer-to-consumer mapping and further processing. Conditional and sparse irregular memory accesses make it tricky to use kernel fusion effectively.

Finally, `color` contains fully dynamic, many-to-one producer-consumer communication common in many Lonestar and Pannotia applications. `Color` producers conditionally process edges of an input graph, while consumers conditionally process vertices based on edge updates. The irregularity of the graph structure causes GPU producer memory accesses to be poorly coalesced, and the data-dependent behavior does not permit a static producer-to-consumer mapping.

6.5.3 Producer-consumer Parallelization

Parallelization techniques: Section 6.2.2 described software structures that can optimize producer-consumer behaviors, and we test variants of them as described here. Our baseline is the **serial producer-consumer** organization, in which the GPU producer kernel runs to completion before the CPU consumer. Next, we test **kernel fusion** implementations of all applications except `color`, which does not permit a reasonable means to track the dynamic producer-to-consumer signaling at a fine grain.

Further, we test four different implementations of concurrent producer-consumer execution. First, for all applications except `color`, we test a version that uses **fine-grained signaling**; each producer thread signals its output to a consumer thread.

Fine-grained signaling uses excessive extra communication to synchronize, so the other implementations use coarse-grained producer-consumer signaling. Specifically, like the code sample in Figure 6.5, all producer threads in a threadblock generate their outputs and the 0th thread signals that their results are ready. Each application uses 256–512 threads per threadblock.

Software unbounded queuing does not bound the dirty data footprint to fit in cache, and results show this implementation generally does not reduce cache spills. However, we also use this implementation to test **Q-cache**, which actively manages producer-consumer communication in cache. **Software finite queuing** bounds the size of the dirty data footprint using estimations described in Section 6.3. As footprint estimations can be inaccurate, we tested a range of configurations and present results for those with best performance.

Producer-consumer asymmetry: Different producer and consumer tasks make progress with widely varying rates, which can make it difficult for programmers to predict their effects on cached dirty data. Most producer tasks in our tests are memory bound, either on off-chip memory access or when accessing GPU L2 atomic operation units. On the other hand, CPU consumer tasks are sometimes latency bound when accessing intermediate data, and other times, compute bound while processing that intermediate data. For indicators of

producer-consumer rate characteristics, Table 6.4 provides two bits of information. First, all producer kernels start running on all 16 GPU cores, but we manually test multiple different CPU consumer thread counts to balance their progress with GPU producers. Second, when memory or cache bandwidth bound, the GPU producer's cache read:write ratio ("R:W Ratio") indicates the portion of activity that results in producer-intermediate data in cache.

6.5.4 Simulation and Energy Modeling

For simulation, we again use gem5-gpu [120]. To estimate power and energy consumed during application run time, we construct a model based on the integrated power models in gem5, called McPAT [83, 18], and GPGPU-Sim, called GPUWattch [81]. By combining results from both simulator power models, we calculate system-wide energy, which sums the heterogeneous processor chip (CPU and GPU cores, caches, and IO) and off-chip memory energy numbers discussed below. For estimating static energy, we assume that cores can be clock and power gated when no work is scheduled to them.

6.6 Evaluation

This section compares the five producer-consumer parallelism techniques: fine-grained signaling, kernel fusion, software unbounded queuing, software finite queuing, and unbounded queuing supported by Q-cache. Overall, the results show that Q-cache supported applications have the best performance and energy characteristics with the least programmer effort. Techniques to keep producers-to-consumer data in cache—kernel fusion, software finite queuing, and software queuing supported by Q-cache—significantly reduce cache spills and improve performance, indicating they are critical for energy-efficiency. Q-cache supported applications usually perform the best, up to a 1.89× speedup over the best software techniques. Q-cache also provides significant energy reduction—up to 26%—by eagerly throttling producers or consumers, and disabling inactive cores. To

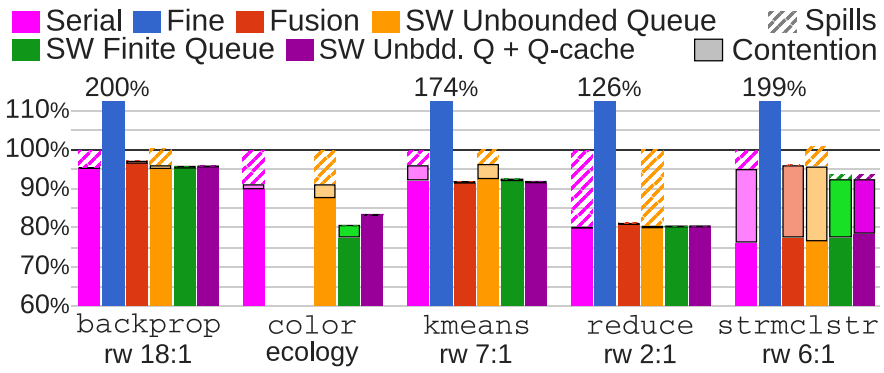


Figure 6.9: Off-chip memory accesses normalized to the baseline serial producer-consumer execution, and broken down by cause. Lower is better.

achieve similar results, software techniques would require access to the dynamic dirty cache footprint, and mechanisms to dynamically enable and disable producer or consumer tasks.

6.6.1 Reducing Cache Spills

We begin our analysis by looking at off-chip memory accesses to observe producer-consumer spills. For representative inputs to each application, Figure 6.9 breaks down off-chip accesses normalized to the baseline serialized producer-consumer case. The striped portion of each bar shows excess memory accesses caused by producer-consumer spills, which are 4–20% of accesses for the baseline versions. The boxed portion of each bar show extra memory accesses caused by cache capacity contention, which can be up to 19% of accesses for the baseline versions.

As expected, techniques that do not manage producer-consumer communication in cache suffer extra memory accesses. Fine-grained producer-consumer signaling (blue) suffers 1.26–2× extra memory accesses for synchronization, consistent with prior findings [71, 25]. Though the plot is cut, spills account for a similar portion of accesses as the baselines. The software queuing techniques avoid this overhead using coarser-grained synchronization, which reduce synchronization accesses to at most 1.6%. However, in software unbounded queuing (yellow), producers often run ahead of consumers quickly,

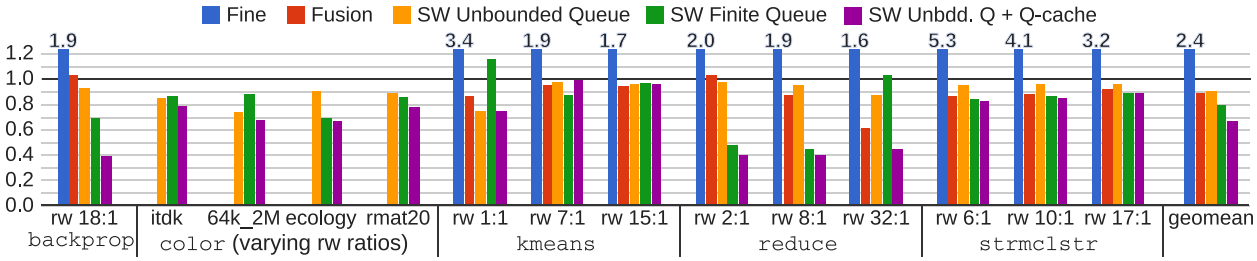


Figure 6.10: Run time of producer-consumer parallelism techniques for a range of applications and input sets normalized to the baseline serial producer-consumer version. Lower is better.

so spill counts are similar to the baseline serial version.

As desired, techniques that aim to manage cache capacity generally eliminate most producer-consumer spills. Kernel fusion (red) generally uses atomics to accumulate results, so data that spilled in baseline application versions is instead combined using in-cache atomic units. Software finite queuing (green) and unbounded queuing with Q-cache (purple) use the same underlying queue structure, so they are similarly able to reduce spills by back-pressuring producers when the queue fills cache. Further, throttling producer and consumer access rates can offer the added benefit of reduced cache contention in the cases of `color`, `kmeans`, and `strmclstr`. Both techniques limit producer working sets to better fit in cache, improving overall data reuse.

6.6.2 Improving Performance

The improved cache behaviors reduce memory access times and improve performance. Figure 6.10 plots the run time of the different techniques normalized to the baseline serial producer-consumer version of each application (omitted). Here, we show input sets that demonstrate the breadth of application character, including a range of cache read:write ratios (labeled) and varying memory access (ir)regularity. Overall, the results show the importance of managing the cached dirty data footprint, and that Q-cache tends to perform the best.

Fine-grained signaling: Fine-grained signaling suffers from excessive communication

for producer-consumer data. Blue bars show that fine-grained signaling incurs more than 2× performance overhead compared to serial implementation. Consumer tasks stall waiting on signal variables, which are often pushed out of cache, causing the consumer's off-chip memory accesses to compete with the producer's. These results are consistent across applications, suggesting that programmers should consider coarser-grained signaling for communication, as we do with software queuing.

Kernel fusion: Kernel fusion (red) can provide reasonable performance benefit by mitigating cache spills. The `backprop` and `reduce` applications fuse consumer tasks, which require heavy use of atomics to accumulate results. Contention for the in-cache atomic units can actually cause slow-downs, especially for low read:write ratios and consumer output data that is localized to a small set of cache lines. Programmers could further optimize fusion implementations by accumulating results hierarchically. However, this organization comes with even more programming complexity than implementing fusion, and it may not provide further performance benefit. Kernel fusion also underutilizes compute and cache resources compared to the other tested implementations, leaving potential performance gains on the table.

Software unbounded queues: Unbounded software queues can provide some performance benefit; performance results (yellow) indicate it can compete with kernel fusion by exposing producer and consumer concurrency. In cases that kernel fusion encounters heavy atomics contention, software unbounded queuing tends to perform better; CPU consumer threads accumulate producer output with a higher throughput than in-cache atomic units. However, unbounded queuing does not manage producer-consumer data in caches, which can result in cache spills. For most applications and input sets, consumer tasks cannot keep up with producers, so cache spills require the consumer to access data off-chip. These extra off-chip accesses limit performance gains of unbounded queuing for all applications.

Software finite queues: When managing queue footprint to fit in cache, performance

can improve substantially. By making better use of compute and cache resources, software finite queuing usually performs better than both kernel fusion and unbounded queuing, resulting in an average speedup of 1.13× over both. However, results also show the shortcomings of estimating cache footprint in software. In cases with high read:write ratios, like `backprop` and `reduce`, software must use conservative cache footprint estimations; producer threadblocks have very large data footprints, so few threadblocks are allowed to run concurrently. In these situations, small effects exacerbate already short cache live times: replacement policy decisions, cache banking and associativity, and memory latencies. The conservative cache footprints cause excessive producer throttling, and performance begins to decline.

Software queues managed by Q-cache: The final set of bars (purple) show that software queuing supported by Q-cache generally performs the best of the techniques, providing an average 1.20× speedup over software finite queuing. For inputs with low read:write ratios where software finite queuing performs well, Q-cache consistently outperforms by reducing synchronization overhead. When queues appear full to the software, software finite queuing require producer tasks to stall and wait for consumer tasks to clear queue entries. Q-cache does not require this blocking and producer wake-up activity, offering small average gains of 1.08× as in `reduce` and `strmclstr`. For higher read:write ratios, Q-cache more accurately estimates the cache's dirty data footprint, so it is more aggressive about allowing many producer tasks to run concurrently. As a result, Q-cache improves performance by up to 1.89× over software finite queuing, which conservatively estimates dirty footprint.

Current results also show cases in which Q-cache conservatively throttles producers, suggesting there is still room to improve tuning. For the `kmeans` rw 7:1 input, Q-cache throttles producers slightly due to early erratic cache spills in some kernels. After this early rate adjustment, producer and consumer rates are very similar such that Q-cache detects matched rates over a long period, and does not try to increase rates. We expect these cases

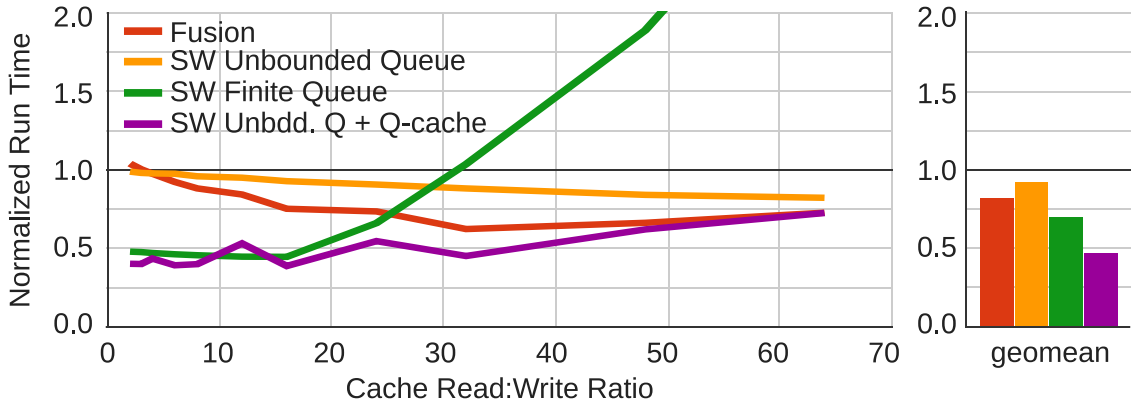


Figure 6.11: `reduce` run times for a range of read:write ratios, each normalized to the back-to-back producer-consumer version. Geometric mean is across all tested applications and inputs. Lower is better.

could be improved by tuning Q-cache's parameters, or extending its management policies.

6.6.3 Q-cache is Adaptable

Q-cache is highly adaptable to many application characteristics, and we demonstrate this adaptability on different and dynamically changing read-write ratios. Later, Section 6.7 analyzes Q-cache's robustness under a broader range of application characteristics.

Read:write ratios: Using the `reduce` application, we sweep the range of read:write ratios to compare performance trends of the parallelism schemes. To increase read:write ratios, we increase the portion of the reduction-sum performed by the GPU producers, reducing the number of partial sums they send to consumers. Figure 6.11 plots the run times normalized to the serial `reduce` version. The plot shows that while kernel fusion and software finite queuing perform well for certain portions of the read:write ratio range, Q-cache adapts to different read:write ratios and performs best in most cases. Across the different read:write ratios, Q-cache shows a geometric mean speedup of 1.48× over software finite queuing.

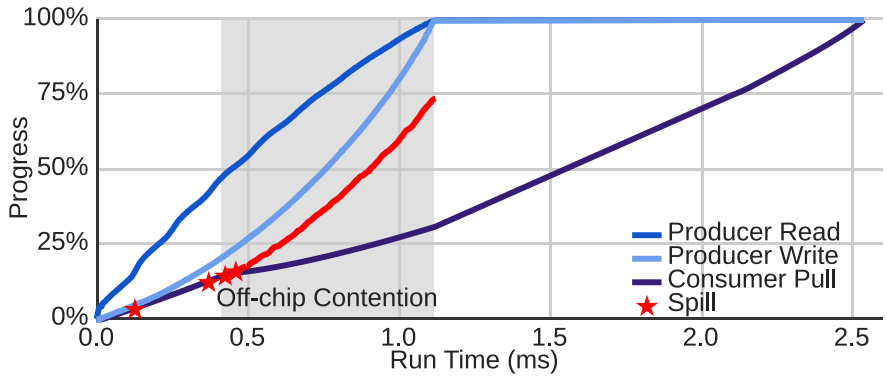
Dynamic producer-consumer behavior Though the `reduce` application has a constant cache read:write ratio during its producer-consumer parallel portion, emerging applications, such as graph analytics, are likely to have more dynamic producer-consumer

behaviors. Next, we show how Q-cache dynamically adapts to producer-consumer behavior changes during run time. Figure 6.12 plots the progress of producer and consumer activity for a kernel in the `color` application using three different parallelization schemes. The vertical axis of each plot is the portion of activity completed through kernel run time for producer reads and writes (blue lines), and consumer data pulls (purple) from the producer writes. We also plot when the cache spills dirty data (red).

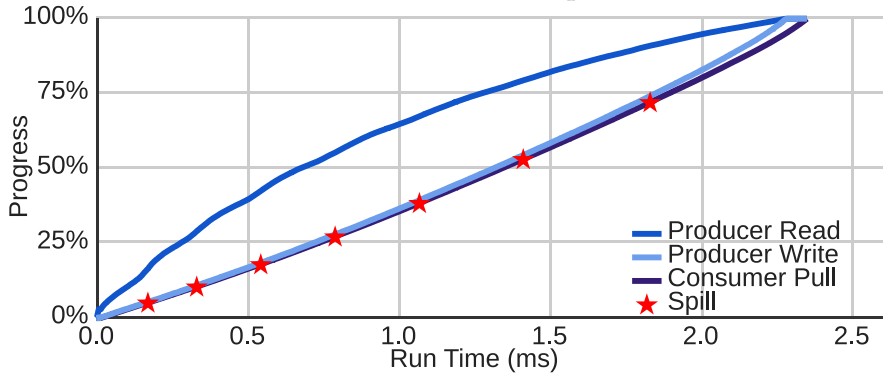
Figure 6.12a shows progress when running software unbounded queues. Consumers fall behind producers, which generate results at an accelerating rate (producer write curve steepens). When consumers (purple) fall too far behind around $0.4ms$, data starts spilling off-chip, so consumers slow as they contend with producers for off-chip memory accesses. They speed up again after the producer finishes at around $1.1ms$, but miss the opportunity to capture a significant portion of data from cache (a 20% run time overhead).

By bounding the size of intermediate cached data, finite queuing throttles producers, as shown in Figure 6.12b. Unfortunately, in this case, software must conservatively estimate cache footprint for times of heavy cache spilling. The conservative estimate limits the number of parallel producers throughout the kernel. The itdk input graph's structure causes producer read:write ratios to fluctuate from 20:1 early in the kernel down to about 1.3:1 late in the kernel. For best performance (and fewest cache spills), software finite queuing must estimate that available cache footprint is just 14% of the maximum late in the kernel, so performance improves by just $1.08\times$ over the serial version.

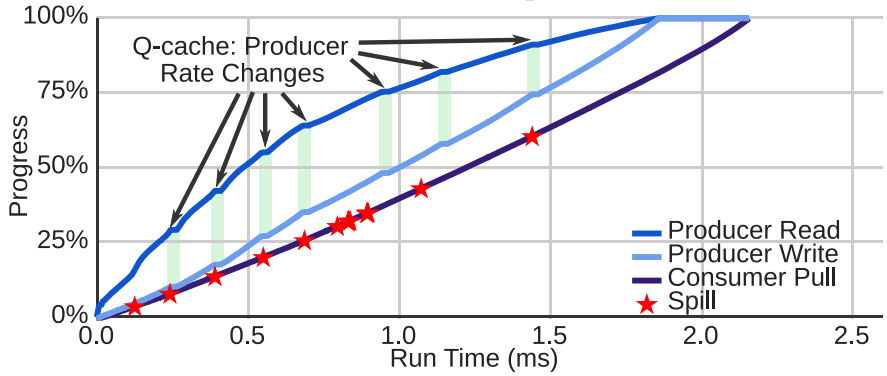
Figure 6.12c shows progress when using unbounded queuing supported by Q-cache, which makes accurate and timely producer rate adjustments. Specifically, as Q-cache detects that cached data may start spilling (e.g., $0.25ms$), it estimates producer and consumer data rates and requests that producers throttle their memory accesses. The plot marks where Q-cache makes 7 rate adjustments, 5 of which are triggered by cache spills when the footprint grows too large. This behavior signals that Q-cache is maximizing the available cache dirty data capacity throughout the execution. This allows producers to make more aggressive



(a) Software unbounded queue



(b) Software finite queue



(c) Software unbounded queue with Q-cache

Figure 6.12: `color` application processing the itdk input graph: producer and consumer progress over run time of a kernel.

progress than finite queuing, resulting in an additional 1.09× speedup.

6.6.4 Improving Energy Efficiency

Three notable factors affect how different application versions consume system energy: off-chip memory accesses, run time changes, and synchronization and stalling overheads.

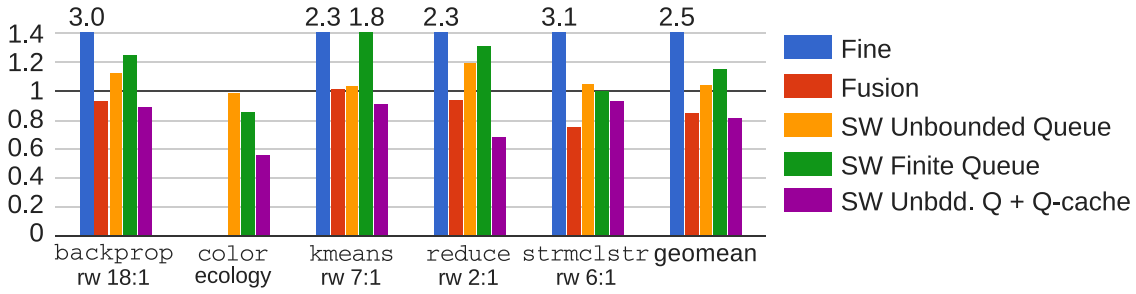


Figure 6.13: Application system-wide energy normalized to serial version. Geometric mean is across all tested applications and inputs. Lower is better.

As we describe these factors, we refer to Figure 6.13, which includes energy estimates for different application versions normalized to the serial version. Results show that to obtain the best performance and energy characteristics, applications must manage cache capacity and balance the number of producer and consumer tasks to avoid stalled and underutilized cores.

First, compared to the serial versions, techniques that cut down cache spills also reduce the off-chip memory access dynamic energy. Although reduced spilling often improves application run time, these off-chip access energy gains usually only account for 1–5% reduced system-wide energy. The largest energy benefit with these techniques comes from reduced application run time. Improved run time reduces an application's static energy by 1–12% by allowing cores to be reallocated or return to low-power states after completion. These two factors account for the majority of energy reductions in kernel fusion versions (average 15%).

Compared to serial versions, however, each of the concurrent producer-consumer techniques can experience energy overheads from synchronization and stalling. With fine-grained signaling, excessive run time and synchronization significantly increase energy across all applications. Kernel fusion often uses GPU atomics to accumulate results, and during times of high atomic unit contention, GPU threads must stall to wait on atomic memory accesses. The use of atomics can slightly reduce dynamic energy compared to executing the operations on CPU or GPU cores, but the GPU cores stall to wait for atomics,

adding run time and static energy. This atomics behavior is most prevalent in the `reduce` application, which experiences high atomic unit contention and little performance or energy benefit. Software unbounded queuing also adds synchronization compared to the serial versions, and since it often fails to improve data caching, it causes an average increase of 4% energy.

The key to achieving energy efficient producer-consumer activity is to balance the compute resources used for producer and consumer tasks. The value of this balance can be seen when comparing software finite queuing and unbounded queuing supported by Q-cache. To achieve peak performance, software finite queuing often requires overallocating the number of producer or consumer tasks, but these tasks can often end up stalling to wait for access to the queue. The required number of producer and consumer tasks is usually both input and system design dependent, making it difficult for the programmer to select a reasonable balance. The extra synchronization and stalling causes an average increase in energy of 15% compared to serial versions.

In contrast, by dynamically disabling or frequency scaling cores allocated to producer or consumer tasks, Q-cache matches their rates to reduce the excess energy from stalled tasks. Q-cache shows an average 18% energy reduction over serial versions and 33% over software finite queuing. Software techniques would need to be able to similarly scale tasks and perhaps core frequency in order to achieve similar energy benefit.

6.6.5 Discussion

Overall, we conclude that the keys to efficient producer-consumer communication in cache are (1) matching producer and consumer data access rates to cache, and (2) accurate dirty data footprint estimates to push the limits on producer rates. Software finite queuing uses fine-grained task management to match producer-consumer rates, but is limited by coarse-grained estimations of dirty data footprint. In contrast, Q-cache tracks cache footprint in a fine-grained manner, and shows that coarse-grained task management is sufficient to

utilize cache capacity efficiently.

For software finite queuing to work as efficiently as Q-cache would require substantial changes to existing GPU programming APIs and hardware. To achieve similar performance as Q-cache, finite queuing will require means to more accurately estimate the dirty data footprint in cache. GPU computing APIs could expose Q-cache-like hardware counters to the software, though it would be complicated for programmers or runtimes to leverage these counters in a manner similar to Q-cache.

To more efficiently manage compute resources, GPU computing APIs would need to expose finer-grained hardware controls. For example, programmers may want to run GPU kernels on a subset of cores or dynamically enable/disable cores to control cache access rates. Further, to provide similar rate management granularity as Q-cache, APIs might need to expose core frequency controls. These functionalities are not currently available, but could allow programmers to optimize software finite queuing for compute and cache efficiency.

6.7 Q-cache Sensitivity to Application Behaviors

Although Q-cache shows desirable performance and energy characteristics for our tested applications, when deployed in a real processor's cache, it would potentially observe many other cache access behaviors. Thus, we must verify that Q-cache can adapt to a broad range of application behaviors and still improve communication. To provide robust producer-consumer communication in cache, Q-cache must accurately estimate access rates and throttle producer-consumer communication in the presence of hardware and software effects that can perturb communication rates.

This section breaks down and tests the different factors that could affect Q-cache's ability to improve producer-consumer caching. We start by analyzing sources of cache access rate fluctuations based on their potential magnitude. First, Section 6.7.1 tests microarchitectural

factors that can perturb memory access latencies by 10s–1,000s of cycles. Section 6.7.2 analyzes rate fluctuations caused by task handling and synchronization structures that can cause bursty memory accesses over 1,000s to 10,000s of cycles. Section 6.7.3 analyzes application-level behaviors that can cause rates to fluctuate over numerous task epochs during pipeline stages. Overall, we show that Q-cache smooths short-term rate fluctuations, while still tracking longer-term rate changes to avoid growing the cache's dirty data footprint beyond available capacity.

We then discuss two factors that can complicate Q-cache's job of accurately tracking cached data footprint. First, Section 6.7.4 describes the causes and effects of false data sharing on Q-cache's rate estimations and shows that Q-cache's throttling can compensate for the resulting rate fluctuations. Second, producers sometimes generate data that will not be used by consumer tasks, so some data will spill from cache. Section 6.7.5 describes the causes of this unconsumed data and shows that Q-cache gracefully allows data to spill for applications that have a low proportion of communication that can be classified as producer-consumer.

To perform these tests, we develop a microbenchmark, called "`prodcons`", with adjustable producer-consumer behavior parameterized to span the desired communication characteristics. The microbenchmark can execute producers and consumers in serialized stages as a baseline, but also parallelizes producer and consumer stages using four of the schemes above: fine-grained signaling, software unbounded queues, finite queues, and queues managed by Q-cache. To parameterize `prodcons`' application characteristics requires variable producer-to-consumer communication mappings. While concurrent producer-consumer versions use queued data synchronization that permit dynamic producer-consumer mappings, to develop a kernel fusion version would require a fused kernel that can variably map producer-to-consumer communication depending on `prodcons`' input parameters. Given this complexity and that kernel fusion cannot fully utilize both CPU and GPU cores, we have not invested the effort to create a kernel fusion version for

comparison.

Using `prodcons`, we show that when Q-cache supports concurrent producer-consumer applications, it is robust to different microarchitectural and application behaviors. Overall, Q-cache matches or outperforms software finite queuing in all but a few instances that we expect could be captured with further Q-cache policy tuning. To underscore Q-cache's robustness to different synchronization granularity, we show that it also greatly improves performance when applications use fine-grained signaling. Finally, the results here offer insight into appropriate cache capacity required for Q-cache to be effective. Caches sized comparably to existing hardware are large enough to provide sufficient data live time for Q-cache to make accurate rate estimations, even under very bursty cache access.

6.7.1 Per-Memory-Access Rate Fluctuations

As architects develop a processor's design, they make many decisions about microarchitectural features that can perturb memory access latencies. Extra latencies of 10s–1,000s of cycles can occur regularly or intermittently depending on microarchitectural factors, like arbitration for buffering, or address translation. We must ensure that Q-cache can accurately track producer-consumer communication rates even in the presence of these latency perturbations.

Q-cache can easily handle small fluctuations in timings between memory accesses, but must be configured to estimate rates appropriately when memory access activity causes long delays, such as TLB misses. First, by observing numerous cache accesses before calculating rates, rate estimations can amortize error introduced by small latency perturbations that result from microarchitectural effects. Small latency sources include CPU pipeline squashes from branch misprediction or memory order violations, GPU warp scheduling, or DRAM row buffer misses. Most microarchitectural latency perturbations are less than 10s of cycles, so when amortized over hundreds of memory accesses, they rarely contribute more than 1% error into rate estimations.

To verify that Q-cache robustly handles these small perturbations, we test running the `prodcons` microbenchmark in simulation. We configure gem5-gpu to introduce random delays of up to 30 cycles into cache and interconnect buffers through simulation run time. In common cases, the GPU accesses memory close to practical bandwidth limits, and random memory access delays have almost no affect on rate estimations. Although rate estimations fluctuate up to 2.8% in our tests, the average difference is just 0.21%. Despite small changes in rate estimates, Q-cache triggers the same rate scaling in all tests, because task and frequency scaling are significantly coarser than rate estimations. Performance remains the same.

Longer latency perturbations in the range of 100s–1,000s of cycles can introduce significant rate estimation differences under certain circumstances, but when configured appropriately, Q-cache still behaves well. Specifically, TLB misses from either the CPU or the GPU can trigger long latency page table walks or minor page fault handling. Page table walks can have latencies of up to 600 cache cycles, while minor page fault handling can take up to 3,500 cache cycles. If producer or consumer stages have limited TLP and MLP, such as a single-threaded low-ILP CPU consumer, spurious TLB misses could cause rate estimation error up to roughly 25% due to a lack of parallel activity to amortize them.

In practice, we expect that Q-cache will be able to gracefully handle TLB misses. First, in our test applications, which tend to have high TLP, we find TLB misses to be either very infrequent or regular, resulting in generally stable rate estimations. In these circumstances, Q-cache performs well. Further, we expect future processor designs will aim to mitigate the effects of these long latency activities. Recent studies [121, 117] and IOMMU drivers for AMD processors show that address translation latency in heterogeneous processors can be significantly reduced using caching and parallelization techniques. These advances should reduce the potential rate estimation error caused by TLB misses. Finally, in the worst case, Q-cache could remain disabled when software launches producer or consumer stages with limited TLP.

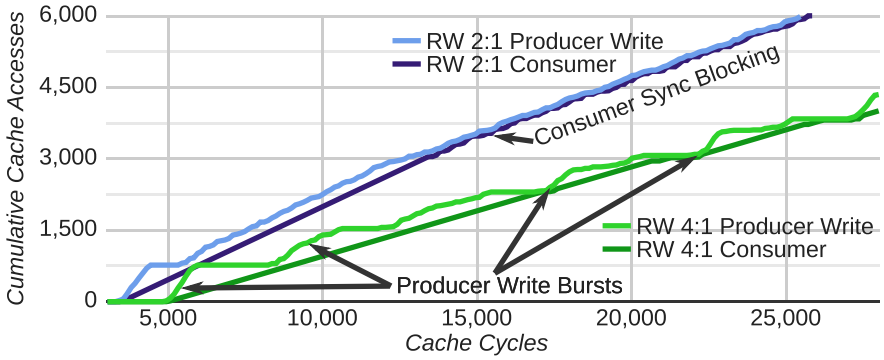


Figure 6.14: `prodcons` application: Example effects of varying task granularity on GPU-producer cache writes and CPU-consumer reads. GPU bursts can be hundreds of cache accesses and spread arbitrarily through time. Coarse-grained synchronization can cause the consumer to block for hundreds of cycles.

6.7.2 Task-level Fluctuations

Chapter 2 described that GPU threadblocks can cause large bursts of memory accesses. These bursts are an example of how the collective memory accesses of numerous concurrent tasks can cause significant cache access rate fluctuations, especially when tasks progress evenly. Here, we analyze such fluctuations by testing the effects of GPU-producer burst writes and CPU-consumer synchronization granularity. Q-cache must observe enough cache accesses so that when estimating producer and consumer rates, it amortizes bursts and synchronization blocking to avoid oscillating rate adjustments. We find that a 2MB L2 cache is sufficient to adequately smooth rate estimates.

Many GPU applications contain bursty memory access behaviors. To show examples of these common task-level cache access characteristics, Figure 6.14 plots cumulative GPU producer writes and CPU consumer reads for the `prodcons` microbenchmark configured with two different producer read-write ratios, 2:1 and 4:1. The plot shows bursty producer write behavior that can cause oscillating instantaneous cache access rates. Coarse-grained synchronization can also cause consumer tasks to wait for extended periods between when producer threadblocks generate data and signal that it is available.

GPU Threadblock Bursts: GPU access bursts can occur during a couple phases of a kernel. Often, the GPU's first threadblocks in a kernel will write their outputs at roughly

the same time, as can be seen between 3,500 and 5,500 caches cycles in the plot. These early cache access bursts are usually most pronounced, but beyond the first burst, two behaviors can arise. First, with the RW 2:1 input, task run times are short, averaging 3,100 cache cycles. Their collective cache accesses cause enough contention and delays that tasks evenly distribute their accesses through time. Despite small perturbations, the GPU reaches a steady data write rate, and Q-cache can easily estimate this rate. Steady cache access behavior is common in applications with low read-to-write ratios or irregular task progress.

Second, longer task run times can cause consistent GPU write bursts through a kernel. In the `prodcons` RW 4:1 input, tasks average 5,000 cycles run time. Though these tasks also experience contention that distributes their cache accesses through run time, their write time distribution remains small relative to task run times. Tasks also have a narrow distribution of run times, so their writes appear as bursts throughout the kernel. Figure 6.14 shows these bursts of roughly 768 cache accesses every 5,000 cycles.

To ensure that Q-cache observes enough bursts to amortize the instantaneous rates, it can wait until it observes dirty data spills before sending rate change requests. The instantaneous write rate of `prodcons` RW 4:1 bursts can exceed the sustained write rate of RW 2:1 by more than 5×. Fortunately, bursts of any size or periodicity can be amortized by observing just 3–5 bursts, reducing rate estimate error to within 10–25%. Q-cache can wait up to 100,000s of cache cycles—the dirty data live time in cache—to ensure accurate rate estimates. Section 6.6.3 shows that Q-cache performs well over the range of read-write ratios.

Besides read-write ratios the compute intensity of GPU tasks can also change their write burstiness. Tasks with higher compute intensity run for longer periods before they write their results, causing bursts when they progress evenly. Using the `prodcons` application, we sweep a range of compute intensity from 0.5 FLOPs/B to 20 FLOPs/B. To increase compute intensity, we parameterize `prodcons`' GPU producer threadblocks to perform

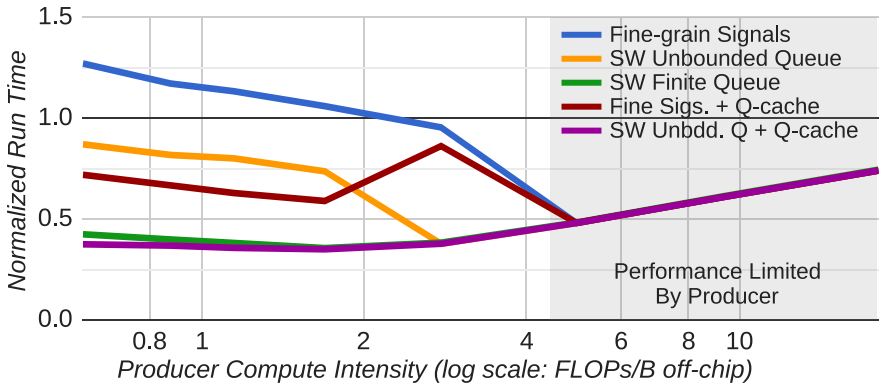


Figure 6.15: `prodcons` run times for a range of producer compute intensities, each normalized to the serial producer-consumer version. Lower is better.

arbitrary operations on input data in a manner similar to small matrix convolutions.

Figure 6.15 plots the run times normalized to the serial `prodcons` version for varying compute intensity. These results show that Q-cache adapts well to varying compute intensity. For both fine-grained synchronization and software unbounded queues, Q-cache support provides up to 2× performance gains when consumers are unable to keep up with producers (i.e., compute intensity below 3–5 FLOPs/B). Software finite buffering performs comparably to unbounded queuing supported by Q-cache. For higher compute intensities, consumers are able to keep up with producers (shaded portion of the plot), so all concurrent producer-consumer techniques can capture communication in cache and improve performance over the serial versions.

When using fine-grained signaling, Q-cache stumbles with compute intensity near 3 FLOPs/B due to the size of bursts. Low compute intensities cause producer writes to be smoothed, allowing Q-cache to accurately estimate producer rates. Medium compute intensities, on the other hand, cause GPU write bursts with nearly twice as many cache accesses as the coarse-grained synchronization of software queuing due to the extra signal variables, and these bursts are roughly half the dirty cache footprint. Q-cache is only able to observe and smooth about 2 bursts before it needs to throttle the producer rate. Eventually, it accurately throttles the producer rate, but not before some data is spilled from cache. A larger cache would allow Q-cache more time to accurately estimate producer rates.

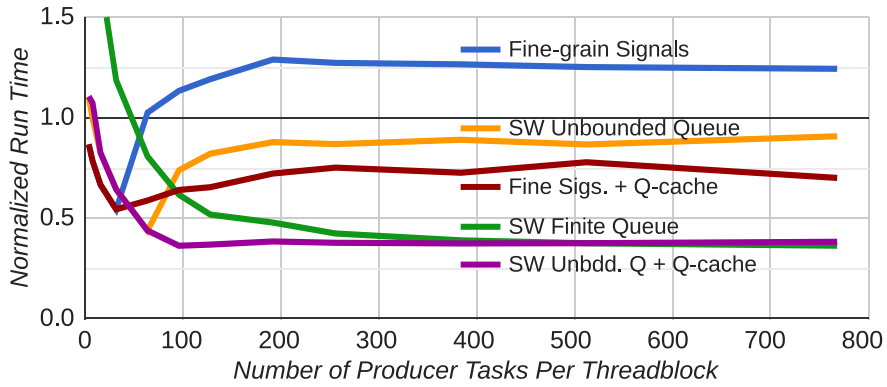


Figure 6.16: `prodcons` run times for a range of producer threadblock sizes normalized to the serial versions. The number of threads per GPU producer threadblock dictates number of participating threads in each software queue signal. Lower is better.

Synchronization Granularity: During concurrent producer-consumer activity in some of our test applications, consumers can also experience coarse-grained blocking while waiting for GPU threadblock signals. With fine-grained signaling between producer and consumer, the producer usually generates a small output per signal variable (e.g., 4–8B). With threadblock signaling in software queues, on the other hand, a signal variable can guard chunks of data up to 8kB or larger, depending on task activity and threadblock size. As a result of large data chunks per signal, consumer tasks can wait up to roughly 1,000 cache cycles between data chunks. Figure 6.14 shows examples of this synchronization blocking, with consumers waiting up to 300 cache cycles for some chunks of data produced by 256-thread GPU threadblocks.

To test varying synchronization granularities, we parameterize the `prodcons` microbenchmark to change the number of threads per GPU threadblock. Fine-grained signaling uses a signal variable per thread, so changing the number of threads per threadblock does not change the synchronization granularity, but can change the total number of concurrent tasks when the number of threads per threadblock is low. On the other hand, since software queue versions of `prodcons` signal results on a per-threadblock basis, this parameter changes the amount of data that the producer generates per synchronization variable.

Regardless of synchronization granularities in these tests, results indicate that Q-cache

provides robust performance. Figure 6.16 shows normalized run times of the different `prodcons` parallelization schemes for a range of threadblock sizes. With threadblocks smaller than 32 threads, fewer threads can coalesce their memory requests, increasing the number of GPU cache accesses and cache contention. In this setting, consumers can keep up with producers, so performance is the same with and without Q-cache. For software unbounded queues and larger synchronization granularity, Q-cache easily throttles producer and consumer rates to perform as well or better than software finite queuing. For fine-grained signaling, Q-cache provides similar improvements of roughly 2×.

Due to cache contention with few threads per threadblock, software finite queuing experiences very high queue management overheads. Finite queuing requires producer threadblocks to stall while allocating space in the queue before generating outputs. With few threads per threadblock causing elevated cache contention, queue allocation latency can be very high, causing the performance of software finite queuing to degrade more quickly as threadblock size decreases. Fortunately, most applications can be structured to use more than about 128 threads per threadblock, but finite queuing sees significant overheads up to 256 threads per threadblock.

Task-level take-aways: The challenges described here indicate that Q-cache needs enough cache capacity to observe more than 3–5 GPU memory access bursts before it can accurately estimate producer rates. Our tests use a 2MB GPU L2 cache, which is large enough to observe large and infrequent write bursts from 16 aggressive GPU cores. Caches larger than 2MB would provide Q-cache more time to calculate more accurate rate estimates, and it is reasonable to expect emerging heterogeneous processors will have enough cache capacity for Q-cache to be effective.

6.7.3 Application-level Fluctuations

At the application-level, data structures and the computation associated with each data element can change throughout producer and consumer kernels. These fluctuations can

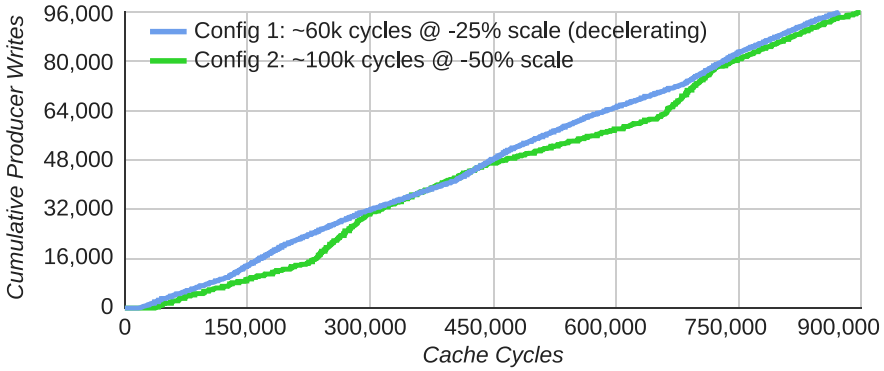


Figure 6.17: `prodcons` cumulative producer writes over time for two application-level rate scaling configurations (approximate cycles between scalings and magnitude of each change).

result in longer and larger-scale shifts in producer and consumer rates. The `color` application is an example (e.g., Figure 6.12a) in which producer write rate smoothly increases by more than 5.5× through kernel run time.

We test how well Q-cache can track rate fluctuations of varying magnitude and frequency. We structure `prodcons` to change producer write rates by modulating the compute intensity at varying time scales. We change producer rates between roughly every 10,000 and 1,000,000 cache cycles, and increase or decrease compute intensity in steps of 25% or 50% around the base compute intensities of 1 and 3 FLOPs/B. For low compute intensities, producers have higher cache access rates and are mostly memory bound. On the other hand, by fluctuating around 3 FLOPs/B, different producer phases can be memory or compute bound.

Figure 6.17 shows examples of fluctuating producer writes over time. For these examples, Figure 6.18 plots the actual producer write rates, which show phase-like behavior with steps between rates. The rate accelerations tested with `prodcons` have magnitudes more than three times that of tested applications, and we expect they exceed common application behaviors.

Overall, we find that Q-cache robustly adjusts to application-level rate fluctuations. Figure 6.19 shows the run time of parallel application versions normalized to the serial

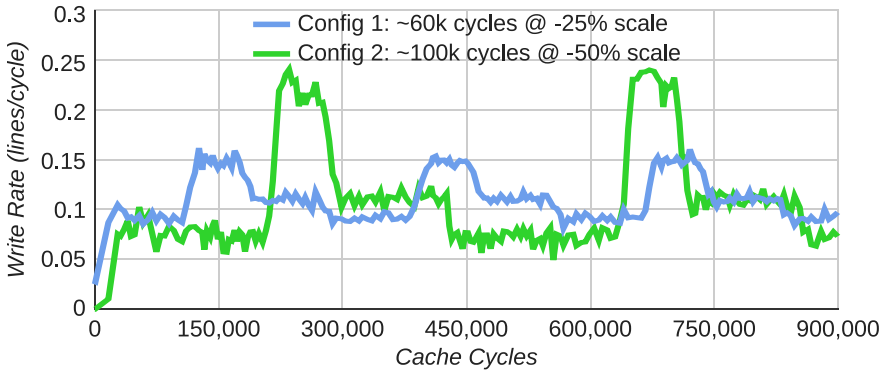


Figure 6.18: `prodcons` producer write rates over time for two application-level rate scaling configurations.

version for different producer write rate scaling. We plot results from scaling producer rates roughly every 60,000 cycles, because results are roughly the same regardless of the rate change frequency. Q-cache improves run times of both fine-grained signaling and software queuing in the presence of these rate changes, and software queuing supported by Q-cache outperforms software finite queuing.

Although relative performance of the different `prodcons` versions remains stable, Q-cache can behave differently depending on the frequency of producer rate changes. When rate changes more quickly than the time to adjust producer or consumer rates (i.e., roughly 50,000 cache cycles), Q-cache tends to throttle producers to a particular rate that works throughout run time. During times of high producer rate, dirty cache footprint increases, but not enough to spill data off-chip. On the other hand, when rate changes are less frequent, Q-cache can oscillate producer throttling. When data will start spilling, it trims producer rate until it detects that the consumer's rate exceeds the producer's, at which point, it can increase the producer's rate again.

6.7.4 False Sharing

In some applications, producer-consumer communication involves moving data from the producer to the consumer's cache, though the consumer may not use all of it. Since Q-cache tracks data movement at the cache-line granularity and assumes that dirty lines will be

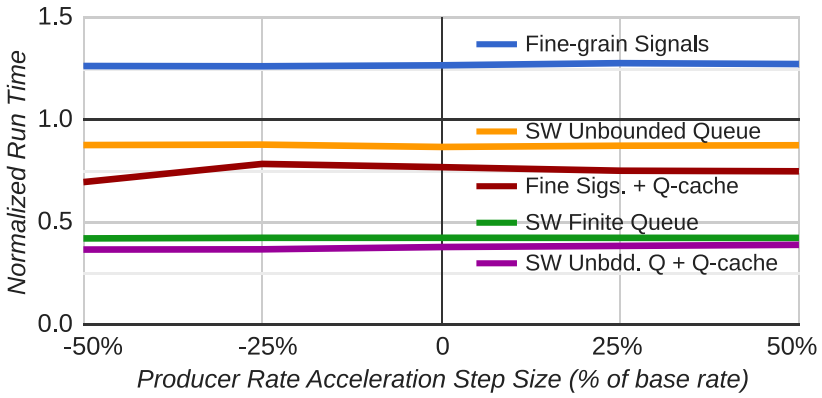


Figure 6.19: `prodcons` run times for a range of application-level producer rate scaling factors and roughly 60,000 cache cycles between scalings, normalized to the serial version. Rate scale factors less than zero have decelerating producer writes. Lower is better.

shared, false sharing occurs within cache lines from which consumer tasks do not use all of the producer data. When data is falsely shared, it appears to Q-cache as though the consumer briefly accelerates its consumption rates, because it does not spend the time to process all data in each cache line. False sharing could cause Q-cache to over-estimate the consumer's rate and incorrectly throttle producer or consumer as a result. We find false sharing of up to 22% of producer output in `strmclstr` and `color` applications, and here, we use the `prodcons` microbenchmark to isolate the effects of false sharing on Q-cache.

By observing memory access address traces, we find that false sharing in both `strmclstr` and `color` is fairly uniform across separate cache lines, so we aim to mimic this behavior. To make `prodcons` cause false sharing, we parameterize the consumer tasks to selectively ignore portions of each cache line that they pull from the consumer. Before the producer-consumer relationship begins, each consumer task calculates a cache line offset value that will be used to decide which portion of data in each cache line that the task might ignore. As consumer tasks pull producer results, they calculate both a cache line identifier and the cache line offset of their data element reads. For a randomized subset of cache lines indicated by their identifiers, a consumer task ignores data in the line with offsets greater than the pre-calculated offset such that the total falsely shared data is equal to the `prodcons` false sharing input parameter.

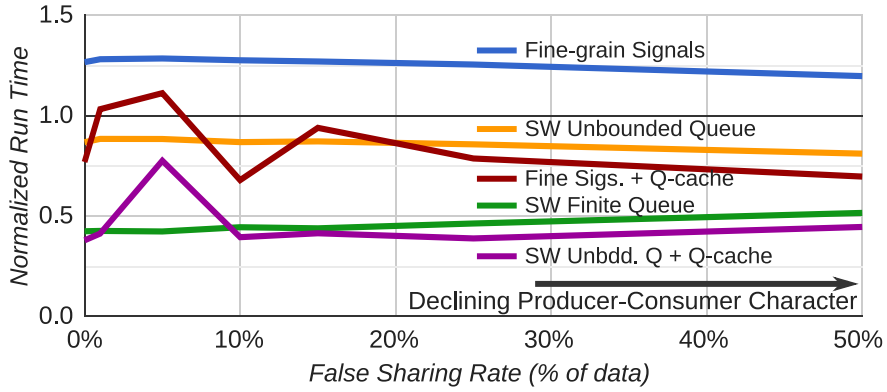


Figure 6.20: `prodcons` run times for a range of false data sharing rates normalized to the serialized producer-consumer version. As false sharing rate increases, the direct producer-consumer character of the application declines.

Generally, false sharing rarely upsets the operation of Q-cache when it can be difficult to correct for the apparent consumer rate changes. Figure 6.20 plots the run times of the different parallelization schemes normalized to the serial versions for a range of false sharing. Note that when more than 50% of data is false shared to the consumer, the communication character is no longer a strong producer-consumer relationship. Such relationships are unlikely to be good targets for concurrent producer-consumer activity. In most cases, Q-cache correctly tracks and throttles producer and consumer rates, and improves performance consistently with the case that all data is consumed (i.e., 0% false sharing).

Q-cache can experience trouble when false sharing causes the appearance that consumer rates oscillate (e.g., fine signals at 1%, 5%, and 15% false sharing, and software queuing at 5% false sharing). In these cases, Q-cache can overestimate the consumer rates and request that producers speed up, causing them to push intermediate data out of cache. This spilling further complicates Q-cache's rate estimation, because it needs to later slow the producer down again. Q-cache might be able to avoid this problem by adding a damping factor when it detects rate oscillations. Despite the extra cache spills in these tricky cases, Q-cache still improves the performance of both fine-grained signaling and software unbounded queuing.

6.7.5 Unconsumed Data Will Spill

Unlike false sharing, which only occurs in cache lines from which the consumer reads some data, some applications also produce outputs in cache lines that the consumer will not read immediately or at all. Consumers might read these "unconsumed" cache lines later, but before then, they will spill off-chip. By default, Q-cache expects the consumer to pull these unconsumed lines, so their spills can adversely affect Q-cache's access rate estimates. We observe that in `strmclstr` and `color`, consumers can pull data from as few as 84% of cache lines written by the producer, leaving the rest to spill from cache. Here, we use `prodcons` to isolate the effects as unconsumed data increases.

Similarly to false sharing in `strmclstr` and `color`, we find that unconsumed data cache lines tend to be roughly uniformly distributed through the address ranges of producer-generated data. To model unconsumed data with the `prodcons` microbenchmark, we parameterize the way producer tasks write their results to the output data structure. Specifically, the data structure is allocated to be twice as large as necessary for producer-generated data. All producers write their results at least once to the consumer-expected portion of the output data structure. Randomly, producers write their results to an adjacent chunk of the data structure when they aim to create data that will not be consumed. Consumer tasks only pull the data from the expected portion of the structure, and they do not consume any of the randomly generated results in adjacent locations. These adjacent data eventually spill from cache without getting consumed.

Figure 6.21 shows the run times of the five parallelism schemes normalized to the serial version of the application. As with false sharing, when more than 50% of data is not consumed from a kernel, the communication character is no longer a strong producer-consumer relationship. With both software unbounded queues and fine-grained signals, Q-cache is able to correct for cache spilling up to roughly 20–30% unconsumed data (i.e., 70–80% of data is consumed). Until this point, software unbounded queuing supported by Q-cache performs as well or better than software finite queuing.

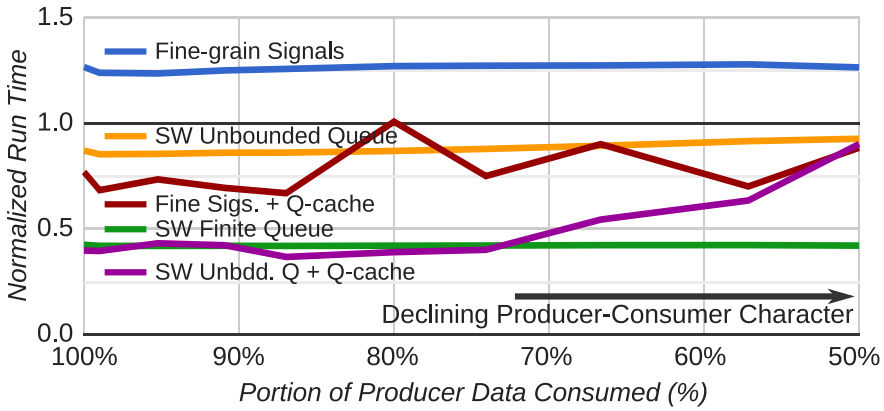


Figure 6.21: `prodcons` run times for varying proportion of producer data that is consumed. Q-cache gracefully degrades performance as less data is consumed. Lower is better.

Although Q-cache shows robust benefits for low unconsumed data rates, as the proportion of unconsumed data increases, it becomes difficult for Q-cache to discern whether data is spilling because it will not be consumed or because consumers have fallen too far behind. When Q-cache detects that roughly as much data is spilled as consumed, it gives up trying to throttle producers. It returns producers to their maximum rates, and disables itself, allowing data to spill similarly to software unbounded queuing. Thus, with unbounded queuing, Q-cache performance begins to degrade around 75% of data consumed, because it can eventually stop throttling producers. More advanced techniques might be able to disambiguate spilled and unconsumed data, but as-is, Q-cache gracefully degrades performance without causing slowdown.

6.7.6 Sensitivity Summary

Overall, results in this section show that Q-cache can provide robust support for producer-consumer communication in cache. It improves performance of producer-consumer communication in the face of microarchitectural, task-level, and application-level perturbations. It also improves performance when producer-consumer communication involves false data sharing or data that is not immediately consumed. Further, across each of these communication rate factors, we show that Q-cache adapts well to different producer-consumer

signaling structures; in addition to supporting software queues, our results show that Q-cache improves performance of fine-grained signaling by roughly 2×.

6.8 Related Work

To the best of our knowledge, this work is the first to (1) propose cache access rate monitoring as a means of comparing producer and consumer progress rates, and (2) propose the use of parallelism management techniques to throttle producer-consumer cache access rates.

Managing fine-grained worklists: Most relevant to our tests, Orr et al. [113], Kim and Batten [71], and Chen and Shen [25] investigate techniques to optimize fine-grained worklists for irregular applications. These works focus on efficient task handling, but still time-multiplex producers and consumers similar to kernel synchronization or fusion/fission. In contrast, this work avoids the complexity of managing fine-grained tasks by using concurrent producer-consumer execution and focusing on efficiently managing task data in cache.

Observing Coherence Transitions: We believe Q-cache to be the first proposal that observes coherence transitions to track and use producer-consumer cache access rates. However, many prior works have also observed coherence events for cache and coherence optimizations. Some use coherence in multiprocessors to predict sharing patterns [96] and switching to low-power cache configurations [70], and to assist pair-wise sharing [66], dynamic self-invalidation [76], migratory sharing [34, 137]. Of particular note, Cheng et al. [28] and Kayi et al. [67] optimize coherence protocols for producer-consumer sharing, which reduces coherence traffic and memory latency, but does not address asymmetric producer-consumer cache access rates.

Core and Memory-Level Parallelism Management: Prior works have considered parallelism management techniques for cache [92, 77] and off-chip memory access quality of service (QoS) [131], and power or energy management [115], but not for efficient producer-

to-consumer communication in cache. Q-cache employs task preemption [141], inactive core disable, and frequency scaling [42] for rate modulation. However, other prior techniques could be employed. Ebrahimi et al. [38] use techniques to limit outstanding memory accesses from load-store units in chip multiprocessors. Similarly, Kayiran et al. [68] reduce outstanding memory accesses by limiting the number of active GPU warps per core.

6.9 Summary

Existing GPU computing applications contain producer-consumer relationships that cause significant cache spilling of intermediate data. When ported to heterogeneous processors, there are opportunities to coordinate work between CPU and GPU cores by concurrently executing producers and consumers. We make the fundamental observation that efficient producer-consumer communication in cache requires that producers and consumers run in close temporal proximity and match their data access rates. Although software transformations can reduce cache spills between producer and consumer, they often place heavy burdens on the programmer to organize task communication and estimate dirty data footprint in cache.

To address these challenges, we propose Q-cache, a novel technique to monitor cache access rates and throttle producer and consumer progress to ensure that intermediate data is passed through cache. Compared to software-only techniques for managing producer-consumer communication, Q-cache improves performance by 1.20× on average and reduces application energy by up to 26%. We conclude that Q-cache provides a general-purpose and efficient solution for managing producer-consumer communication in cache.

7 SUMMARY AND FUTURE DIRECTIONS

This chapter summarizes this thesis and describes a few avenues for future work.

7.1 Summary

Over the last decade, the graphics processor (GPU) has become a prominent compute accelerator by offering fast, energy-efficient processing for data-parallel applications. In systems like desktops and mobile devices, heterogeneous processors have integrated GPUs onto the same chips as general-purpose cores (CPUs), and emerging heterogeneous processors offer shared virtual memory and cache coherence across CPU and GPU cores. These new capabilities encourage programmers to coordinate work across CPU and GPU cores to effectively utilize the cores and cache in heterogeneous processors.

This thesis aims to enable programmers to write applications that deftly and efficiently coordinate computation across CPU and GPU cores in heterogeneous processors. We begin this work by characterizing the microarchitectural behaviors of CPU and GPU cores, and the software pipeline structures of GPU computing applications. Our results show that GPU cores tend to perform as well or better than CPU cores on applications with very wide data-level parallelism (DLP). However, CPU cores can perform better on low-DLP computation, especially when the work has high instruction-level parallelism (ILP). Many software structures contain varying DLP from one pipeline stage to the next. These results indicate that for heterogeneous processors, programmers may adapt code to concurrently execute application stages with high DLP on GPU cores, and stages with low DLP or high ILP on CPU cores. We document our efforts to develop optimized heterogeneous processor applications and the simulation infrastructure to test possible hardware configurations with high fidelity.

Then, we propose and evaluate techniques that improve the performance, programmability, and energy efficiency of synchronization and coordinated computation in GPUs and

heterogeneous processors. Specifically, we identify two processor design challenges that limit the performance and energy efficiency of these coordinated work applications.

First, GPU barrier synchronization provides a useful mechanism for programmers to reduce synchronization granularity from numerous GPU threads. However, GPU barriers can still cause high overhead; threads must wait at barriers for lagging threads. Further, to communicate from GPU to CPU cores requires longer latency memory ordering guarantees than commonly used in discrete GPU applications. To reduce barrier overhead, we propose a hardware technique, the Transparent Fuzzy Barrier, which dynamically finds and executes instructions while threads wait at barriers to reduce barrier overhead by more than 50%. Transparent Fuzzy Barriers require small changes to logic or added storage, and they can be implemented within the existing architectural specification of barriers, easing programmer and compiler burdens to hide barrier overhead.

Second, GPU applications often use very coarse-grained producer-consumer communication, which reduces performance due to excessive cache spills and energy-expensive off-chip memory accesses. Existing software transformations can reduce cache spilling, but they tend to be complicated, requiring the programmer to reason about producer-to-consumer mappings and cached data footprint. To ease the cache management burden, we propose a novel hardware technique, called Q-cache, to support concurrent producer-consumer activity. Q-cache measures the cached data footprint and throttles producer or consumer tasks when buffered data might start spilling from cache. Q-cache eliminates cache spills and reduces contention to significantly improve application performance and energy.

More broadly, our work demonstrates that as systems incorporate more tightly-coupled heterogeneous compute and memory resources, programmers will have opportunities to adapt applications to execute different portions on the most effective resources. In this setting, systems need to offer high-performance and energy-efficient communication and synchronization to coordinate different portions of computation. Additionally, to

ensure programmers can productively utilize the heterogeneous resources, system programming models should offer simple mechanisms to map portions of computation to the compute resources and to coordinate their data communication. Both of our hardware proposals—Transparent Fuzzy Barriers and Q-cache—assist programmers in creating high-performance and energy-efficient applications for heterogeneous processors.

7.2 Future Directions

In Chapters 2 and 3, we discussed many potential future directions for emerging systems and applications for heterogeneous processors. Here, we discuss the lessons we learned while studying Transparent Fuzzy Barriers and Q-cache, and how these lessons guide us toward future work.

Efficient bulk synchronization in GPUs: The results of our study of GPU hardware barriers in Chapter 5 indicate that excessive load imbalance causes the majority of remaining barrier overhead when using Transparent Fuzzy Barriers. If systems integrate more features with latency greater than current off-chip memory access—such as non-volatile memories—the overheads of load imbalance will increase. Programmers and system designers will have more incentive to reduce load imbalance.

System designers can select from many different techniques in an effort to manage load imbalance in GPUs, so future work should investigate the most effective techniques. First, although we tested a warp scheduling technique with TFBs, there is more opportunity to manage thread progress divergence within threadblocks. Warp scheduling techniques that perform well on codes without barriers often prioritize a subset of threads to run ahead of others, but this progress divergence can exacerbate load imbalance in codes with barriers. Future techniques should try to manage progress divergence and to identify and prioritize threads that will run longer than others between barriers.

Second, increased or better-managed GPU parallelism has potential to better hide load

imbalance. NVIDIA Maxwell GPUs add more threadblock contexts and warp schedulers per core. This extra parallelism has potential to hide portions of run time during which few threads are active near barriers by executing instructions from other threadblocks. Our results show that warp schedulers must be careful not to increase contention with lagging threads approaching a barrier, but more parallelism would give flexibility to cover more barrier overheads.

Finally, we expect that performance can be improved around barriers using more aggressive instruction reordering techniques. Register dependencies on memory accesses are the second largest cause of barrier overheads when using TFBs. Out-of-order techniques similar to the TFBs barrier-displaced instruction buffer could allow even more aggressive instruction buffering and reordering. The BDIB allows GPU threads to have in-order issue, but out-of-order dispatch and execution. This concept might be useful for other GPU instruction handling capabilities, such as speculative instruction execution. Further, we expect there is more opportunity to reorder instructions in the compiler using more aggressive techniques than we tested in this thesis. We recommend that future work explore these possibilities.

Efficient producer-consumer communication: The analysis and results in Chapter 6 show that caches play an important role in high-performance and efficient producer-consumer communication. Unfortunately, most existing software is structured using kernel synchronization, which causes significant cache spilling for back-to-back producer-consumer kernels. We expect that adjustments to software systems or runtimes, hardware-software interfaces, and emerging memory consistency and coherence models could also support more efficient producer-consumer communication.

First, for programmers, writing code to execute producer and consumer tasks concurrently and correctly can be difficult. If software techniques introduced higher-level constructs to write code with less explicit thread handling, compilers or runtimes could help manage caches. Specifically, existing software systems, such as Torch and TensorFlow,

structure code using high-level computing functions, which explicitly map producer-to-consumer data flow that can be analyzed using algebraic manipulations in the compiler or runtime. Runtimes for these software systems already map computation to the heterogeneous cores of large-scale systems, but they do not currently support concurrent producer-consumer relationships as a way to manage communication in caches. We expect it would be possible to extend these software systems to automatically and concurrently execute producer and consumer tasks and to signal data readiness between them.

Second, higher-level software abstractions might improve use of cache in the context of regular producer-consumer communication, but it is often difficult for these software systems to interpret and manipulate conditional or irregular communication, such as in graph analytics. In the case of more irregular communication, it might be useful for hardware to expose a more explicit interface for controlling task communication rates. For instance, hardware could provide an instruction to signal when a concurrent producer-consumer relationship is starting as a way to enable a Q-cache-like controller in cache. Further, more advanced techniques might allow software to access estimated cache footprint and for software to explicitly throttle threads when cached data might start spilling. While these techniques will expose significant complexity to software, software libraries or runtime techniques could assist the programmer in on-line estimation and management of cached data footprint.

Third, although we implement Q-cache in the context of a cache coherent heterogeneous processor with scoped memory consistency, there is opportunity to use this cache management technique in the context of other systems. We recommend that future work explore the possibility of implementing Q-cache in GPUs that offer concurrent multikernel execution but lack all of the machinery for full cache coherence. Further, we expect that Q-cache can be implemented within other heterogeneous processors with emerging memory consistency and cache coherence models.

Finally, we test particular mechanisms that allow Q-cache to throttle communication

rates, but other techniques could be employed to throttle producer and consumer communication. Depending on system constraints, future work could consider techniques that modulate memory access rates completely in hardware and do not require core or thread context adjustments. Software intervention techniques could allow threads to explicitly yield their contexts when cached data might start spilling. We recommend that future work explore further possibilities for managing communication rates.

A COMMUNICATION CHARACTERISTICS OF ALL SUITES

This appendix provides a complete set of algorithm-level characteristics for each application from the five benchmark suites that we study in this work. We list the applications and classifications of their characteristics in Table A.1, and we describe the characteristics below.

A.1 GPU Computing Algorithm Structures

Table A.1 records which applications have algorithms that are structured as software pipelines with at least one producer-consumer relationship between stages ("Pipe P-C Comm."). As we describe in Chapter 3, these producer-consumer relationships have a common structure in GPU computing applications. The producer stage processes all elements in a large data structure and generates some intermediate results that a later stage will consume. The intermediate results can update the existing data structure, but more frequently, the producer stage writes results to another data structure that is input to the consumer. The majority of applications in all open-source GPU computing suites contain producer-consumer communication.

Of the applications with coarse-grained producer-consumer communication between pipeline stages, all but two applications could be parallelized to execute producer and consumer stages concurrently. We classify applications as producer-consumer parallelizable ("Pipe Paral.-able") if a programmer could make minimal modification to pipeline stages such that producer stage tasks could signal when data is ready to consumer tasks that can immediate run, possibly while other producer tasks proceed.

We classify software pipelines as producer-consumer parallelizable when they meet two conditions. First, the producer-consumer communication must be unidirectional to avoid potential deadlocks. Fortunately, the GPU computing software pipelines we study all break down the computation so that producer and consumer stages avoid bi-directional communication. Second, to be parallelizable, producer-consumer communication cannot

Table A.1: Complete software pipeline characteristics for Lonestar, Pannotia, Parboil, Polybench, and Rodinia applications in gem5-gpu repositories (14 December 2016).

		Algorithm Organization			Implementation Details					
Suite	Application	Pipe P-C Comm.	Pipe Paral. -able	Vary-DLP P-C	Reg. P-C Comm.	Irreg. P-C Comm.	Topo.-driven	Data-driven	Fine Sync: Atomic	SW Work Queue
Lonestar	bfs_atomic	Y	Y	Y	Y	Y	Y		Y	
	bfs_ls	Y	Y		Y		Y		Y	
	bfs_wla	Y	Y	Y	Y	Y	Y			
	bfs_wlc	Y	Y	Y	Y	Y		Y	Y	Y
	bfs_wlc_gb	Y		Y	Y	Y		Y	Y	Y
	bfs_wlw	Y	Y	Y	Y	Y		Y	Y	Y
	bh	Y	Y	Y	Y	Y	Y	Y	Y	Y
	dmr	Y	Y	Y	Y	Y		Y	Y	Y
	mst	Y	Y	Y	Y	Y	Y	Y	Y	
	pta	Y	Y	Y	Y	Y	Y	Y	Y	Y
	sp	Y	Y	Y	Y	Y	Y	Y	Y	Y
	sssp_ls	Y	Y	Y	Y	Y	Y		Y	
	sssp_wlc	Y	Y	Y	Y	Y		Y	Y	Y
	sssp_wln	Y	Y	Y	Y	Y		Y	Y	Y
Pannotia	bc	Y	Y	Y	Y	Y	Y		Y	
	color_mx	Y	Y	Y	Y	Y	Y			
	color_mmx	Y	Y	Y	Y	Y	Y			
	fw	Y	Y		Y	Y	Y			
	fw_block	Y	Y	Y	Y	Y	Y			
	mis	Y	Y	Y	Y	Y	Y			
	pr	Y	Y	Y	Y	Y	Y		Y	
	pr_spmv	Y	Y	Y	Y	Y	Y			
	sssp	Y	Y	Y	Y	Y	Y			
	sssp_ell	Y	Y	Y	Y	Y	Y			
Parboil	bfs	Y	Y	Y	Y	Y	Y		Y	Y
	cutcp	Y	Y	Y	Y		Y			
	fft	Y	Y	Y	Y	Y	Y			
	histo	Y	Y	Y	Y		Y		Y	
	lbm	Y	Y		Y		Y			
	mm						Y			
	mri-grid	Y	Y	Y	Y	Y	Y		Y	
	mri-q	Y	Y	Y	Y		Y			
	sad	Y	Y	Y	Y		Y			
	spmv						Y			
	stencil	Y	Y		Y		Y			
	tpacf						Y			

		Algorithm Organization			Implementation Details					
Suite	Application	Pipe P-C Comm.	Pipe Paral. -able	Vary-DLP P-C	Reg. P-C Comm.	Irreg. P-C Comm.	Topo.-driven	Data-driven	Fine Sync: Atomic	SW Work Queue
Polybench	2dconv						Y			
	2mm	Y	Y	Y	Y		Y			
	3dconv						Y			
	3mm	Y	Y	Y	Y		Y			
	atax	Y	Y	Y	Y		Y			
	bicg	Y	Y	Y			Y			
	corr	Y	Y	Y	Y		Y			
	covar	Y	Y	Y	Y		Y			
	fdtd-2d	Y	Y	Y	Y		Y			
	gemm	Y	Y		Y		Y			
	gesummv						Y			
	gramschm	Y	Y	Y	Y		Y			
	mvt	Y	Y		Y		Y			
	syr2k						Y			
	syrk						Y			
Rodinia	backprop	Y	Y	Y	Y		Y			
	bfs	Y	Y	Y	Y	Y	Y			
	btree						Y			
	cell	Y	Y		Y		Y			
	cfd	Y	Y	Y	Y		Y			
	dwt	Y	Y	Y	Y		Y			
	gaussian	Y	Y	Y	Y		Y			
	heartwall	Y		Y	Y		Y			
	hotspot	Y	Y		Y		Y			
	kmeans	Y	Y	Y	Y		Y			
	lavaMD						Y			
	leukocyte	Y	Y	Y	Y		Y			
	lud	Y	Y	Y	Y	Y	Y			
	mummer	Y	Y		Y	Y	Y			
	myocyte	Y	Y		Y		Y			
	nn						Y			
	nw	Y	Y	Y	Y	Y	Y			
	pf_float	Y	Y	Y	Y	Y	Y			
	pf_naive	Y	Y	Y	Y		Y			
	pathfind	Y	Y		Y		Y			
	srad	Y	Y		Y		Y			
	strmclstr	Y	Y	Y	Y	Y	Y			
Totals										
Lonestar	14	14	13	13	14	13	8	10	13	9
Pannotia	10	10	10	9	10	10	10	0	2	0
Parboil	12	9	9	7	9	3	12	0	3	1
Polybench	15	10	10	8	9	0	15	0	0	0
Rodinia	22	19	18	13	19	6	22	0	0	0
Sums	73	62	60	50	61	32	67	10	18	10
		85%	82%	68%	84%	44%	92%	14%	25%	14%

include all-producers-to-all-consumers data passing. If all consumers required data from all producers, all consumers would need to wait until all producers finish executing. Again, fortunately, applications we study have mostly one-to-one or localized many-to-one producer-to-consumer communication.

We do not count two applications as producer-consumer parallelizable: Lonestar `bfs_wlc_gb` and Rodinia `heartwall`. These applications use large fused kernels to perform all work on the GPU, and they complete all synchronization on the GPU using hardware threadblock barriers or global barriers that cause all GPU threads to proceed together through stages. This structure witnesses producer-consumer communication and can reduce overheads from launching a kernel for each processing stage. However, a programmer would need to unfuse the kernels and localize producer-consumer communication to run producer and consumer stages concurrently.

Finally, as described in Chapter 4.1, applications that are good candidates to target at heterogeneous processors have varying data-level parallelism (DLP) from one pipeline stage to the next. We classify an application as having varying DLP when it contains successive pipeline stages with data footprints or task counts that vary by more than 50%. Data footprint and task counts are proxies for the number of data elements processed by each stage. Chapter 4.1 describes the algorithm structures and classes of applications that commonly contain varying DLP pipeline stages, and we denote the particular applications in Table A.1 ("Vary-DLP P-C").

A.2 Regular and Irregular Algorithms

Burtscher et al. define regular and irregular applications [21]. A computation is considered to be regular if the data can be regularly structured, as in arrays or matrices, and tasks process the data in regular or statically predictable ways. In irregular computations, on the other hand, tasks operate on irregular data structures, such as graphs or priority queues,

and tasks may need to be spawned dynamically based on the data.

Each pipeline stage of an application can be a regular or irregular computation, so an application can be classified as containing both regular and irregular algorithms. As we describe in Chapter 4.1, irregular applications often contain the varying DLP characteristics that may be a good fit for heterogeneous processors. However, when adapting these applications for heterogeneous processors, programmers will need to consider the synchronization and communication structures, which can be affected by the regularity of the computations.

We classify the regularity of communication between pipeline stages in Table A.1. Similar to the definition of regular computations, regular producer-consumer communication ("Reg. P-C Comm.") passes producer intermediate data to consumers using regular structures and with regular or statically predictable mapping between producer and consumer tasks. On the other hand, irregular producer-consumer communication ("Irreg. P-C Comm.") either passes data using irregular structures, or with irregular or dynamically-determined producer-to-consumer mappings.

Most applications in the Parboil, Polybench, and Rodinia benchmark suites contain regular data structures and computation, and as a result, the communication between pipeline stages is mostly regular. On the other hand, the Lonestar and Pannotia benchmark suites contain numerous irregular pipeline stages, so data is often passed from producer to consumer using irregular structures.

A.3 Topology-driven and Data-driven Algorithms

Nasre et al. describe topology-driven ("Topo.-driven") and data-driven ("Data-driven") task handling in emerging irregular applications [101]. In each stage of computation, topology-driven applications spawn a task for every data element in a structure, such as each node or edge in a graph. Tasks first check whether there is work to do at their

respective element. If so, the task completes the work, but if not, the task exits immediately, completing no useful work. When small subsets of the data must be processed in a stage, few tasks do useful work, so a topology-driven approach can be very wasteful of compute resources.

An alternative approach is called "data-driven" irregular computation [101], because it accounts for whether a data element must be processed in the next computational stage. When a task updates a data element during one stage, it can often check whether its activity will trigger further work on that data in the following stage. If so, it can signal this requirement by creating a work item (request) in a queue. The following stage of computation can just spawn tasks for each work item in the queue rather than for all data elements.

Although Lonestar and Pannotia suites both contain applications that operate on irregular data structures, only applications in the Lonestar suite are optimized to process data using data-driven algorithm structures. The Lonestar applications witness significant performance improvement—sometimes more than 25×—from data-driven algorithm design. Our testing suggests that most of the Pannotia applications could benefit from data-driven transformations, but our work only applies this technique in practice for the `color` application, which shows up to 35× performance improvement depending on the character of the input graph.

A.4 Existing Application Synchronization

Finally, some applications in open-source suites already contain some synchronized data communication using structures we describe in Chapter 4.2. Some applications use fine-grained data updates with GPU atomic operations ("Fine Sync: Atomic" in Table A.1) to perform coordinated data updates across many threads in a lock-free manner. In a few cases, like Parboil `histo` and `mri-grid`, use of atomic operations can signal more complicated

inter-task communication than the common producer-consumer communication in most applications. These atomics introduce extra communication ordering requirements that can complicate the programmer's job when trying to parallelize producer-consumer pipeline stages.

More commonly, though, atomics are used to manage software queues to pass data from producers to consumers ("SW Work Queue" in Table A.1). Producer stage tasks atomically push work items into software queues—often using atomics—to designate the work that consumer tasks must complete at a later time. Most of the Lonestar applications have data-driven irregular algorithm implementations that reduce the amount of work that successive stages of the application must complete. These implementations use software queues with atomics to track the work items. In Chapter 6, we describe how software queues can reduce the programmer's burdens for mapping producer-to-consumer communication.

BIBLIOGRAPHY

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *ArXiv e-prints*, March 2016.

[2] Neha Agarwal, David W. Nellans, Mike O'Connor, Stephen W. Keckler, and Thomas F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2015, pp. 354–365.

[3] Neha Agarwal, David W. Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler, "Page Placement Strategies for GPUs within Heterogeneous Memory Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2015, pp. 607–618.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan, "Data Center TCP (DCTCP)," in *Proceedings of the ACM SIGCOMM Conference*, August 2010, pp. 63–74.

[5] George Almási, Charles Archer, José G. Castaños, John Gunnels, C. Chris Erway, Philip Heidelberger, Xavier Martorell, José E. Moreira, Kurt Pinnow, Joe Ratterman, Burkhard Steinmacher-burow, William D. Gropp, and Brian Toonen, "The Design and Implementation of Message Passing Services for the BlueGene/L Supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 393–406, March 2005.

[6] AMD, "Southern Islands Series Instruction Set Architecture," http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf, December 2012, accessed: 2015-07-30.

[7] Jayvant Anantpur and R. Govindarajan, "PRO: Progress Aware GPU Warp Scheduling Algorithm," in *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)*, May 2015, pp. 979–988.

[8] ARM, "CoreLink CCI-400 Cache Coherent Interconnect," 2012.

[9] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 416–427.

[10] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[11] Ioana Baldini, Stephen J. Fink, and Erik Altman, "Predicting GPU Performance from CPU Runs Using Machine Learning," in *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 254–261.

[12] Ethel Bardsley and Alastair F. Donaldson, "Warps and Atomics: Beyond Barrier Synchronization in the Verification of GPU Kernels," in *NASA Formal Methods*. Springer, 2014, pp. 230–245.

[13] Luiz A. Barroso, Kourosh Gharachorloo, and Edouard Bugnion, "Memory System Characterization of Commercial Workloads," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1998, pp. 3–14.

[14] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber, "Mathematizing C++ Concurrency," in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, January 2011, pp. 55–66.

[15] Bob Beck, Bob Kasten, and Shreekant Thakkar, "VLSI Assist for a Multiprocessor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1987, pp. 10–20.

[16] Carl J. Beckmann and Constantine D. Polychronopoulos, "Fast Barrier Synchronization Hardware," in *Proceedings of the International Conference on High Performance Networking and Computing (SC)*, November 1990, pp. 180–189.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, August 2011.

[19] Emily Blem, Hadi Esmaeilzadeh, Renee St. Amant, Karu Sankaralingam, and Doug Burger, "Multicore Model from Abstract Single Core Inputs," *IEEE Computer Architecture Letters*, vol. 12, no. 2, pp. 59–62, July 2013.

[20] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch, "InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 233–244.

[21] Martin Burtscher, Rupesh Nasre, and Keshav Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, November 2012, pp. 141–151.

[22] Trevor E. Carlson, Wim Heirman, Osman Allam, Stefanos Kaxiras, and Lieven Eeckhout, "The Load Slice Core Microarchitecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2015, pp. 272–284.

[23] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, September 2013, pp. 185–195.

[24] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[25] Guoyang Chen and Xipeng Shen, "Free Launch: Optimizing GPU Dynamic Kernel Launches Through Thread Reuse," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2015, pp. 407–419.

[26] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi, "Hardware-modulated Parallelism in Chip Multiprocessors," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 54–63, November 2005.

[27] Xi E. Chen and Tor M. Aamodt, "Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 3, pp. 10:1–10:28, October 2011.

[28] Liqun Cheng, J.B. Carter, and Donglai Dai, "An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2007, pp. 328–339.

[29] Wu chun Feng and Shucai Xiao, "To GPU Synchronize or not GPU Synchronize?" in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, June 2010, pp. 3801–3804.

[30] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd M. Austin, "MEVBench: A Mobile Computer Vision Benchmarking Suite," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2011, pp. 91–102.

[31] Jonathan Cohen and Patrice Castonguay, "Efficient Graph Matching and Coloring on the GPU," in *GPU Technology Conference*, May 2012.

[32] Ronan Collobert, Laurens van der Maaten, and Armand Joulin, "Torchnet: An Open-Source Platform for (Deep) Learning Research," in *International Conference on Machine Learning (ICML)*, 2016.

[33] P. Coteus, H. R. Bickford, T. M. Cipolla, P. G. Crumley, A. Gara, S. A. Hall, G. V. Kopcsay, A. P. Lanzetta, L. S. Mok, R. Rand, R. Swetz, T. Takken, P. La Rocca, C. Marroquin, P. R. Germann, and M. J. Jeanson, "Packaging the BlueGene/L Supercomputer," *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 213–248, March 2005.

[34] Alan L. Cox and Robert J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 1993, pp. 98–108.

[35] "Cray Research, Inc., CRAY T3D System Architecture Overview," September 1993.

[36] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout, "Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2013, pp. 511–522.

[37] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas, "WeeFence: Toward Making Fences Free in TSO," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 213–224.

[38] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010, pp. 335–346.

[39] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A. Joao, Onur Mutlu, and Yale N. Patt, "Parallel Application Memory Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2011, pp. 362–373.

[40] Alexandre E. Eichenberger and Santosh G. Abraham, "Modeling Load Imbalance and Fuzzy Barriers for Scalable Shared-memory Multiprocessors," in *Proceedings of the Hawaii International Conference of System Sciences (HICSS)*, January 1995, pp. 262–271.

[41] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Proceedings of the International Conference on Parallel Processing (ICPP)*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 216–225.

[42] Rong Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Ziliang Zong, "Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, October 2013, pp. 826–833.

[43] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2011, pp. 235–246.

[44] Michael I. Gordon, William Thies, and Saman Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006, pp. 151–162.

[45] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, "The NYU Ultracomputer - Designing a MIMD, Shared-memory Parallel Machine," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, April 1982, pp. 27–42.

[46] Chris Gregg and Kim Hazelwood, "Where is the data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2011, pp. 134–144.

[47] Kshitij Gupta, Jeff A. Stuart, and John D. Owens, "A Study of Persistent Threads Style GPU Programming for GPGPU Workloads," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–14.

[48] Rajiv Gupta, "The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1989, pp. 54–63.

[49] Rajiv Gupta and Michael Epstein, "High Speed Synchronization of Processors Using Fuzzy Barriers," *International Journal of Parallel Programming*, no. 1, pp. 53–73, February 1990.

[50] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *IEEE International Workshop on Workload Characterization*, December 2001, pp. 3–14.

[51] A. Gutierrez, R.G. Dreslinski, T.F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2011, pp. 81–90.

[52] Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Computer Architecture Letters*, vol. 8, no. 1, pp. 25–28, 2009.

[53] Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood, "QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs," in *Proceedings*

of the International Symposium on High-Performance Computer Architecture (HPCA), February 2014, pp. 189–200.

[54] Joel Hestness, Stephen W. Keckler, and David A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, October 2014, pp. 150–160.

[55] Joel Hestness, Stephen W. Keckler, and David A. Wood, "GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, October 2015, pp. 87–97.

[56] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood, "Heterogeneous-race-free Memory Models," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 427–440.

[57] "HSA Foundation Presented Deeper Detail on HSA and HSAIL," HotChips, August 2013.

[58] Intel, "OpenCL Programmability on 4th Generation Intel Core Processors," http://software.intel.com/sites/billboard/article/opencl-programmability-4th-generation-intel-core-processors, June 2013.

[59] Intel, "The Compute Architecture of Intel Processor Graphics Gen8," https://software.intel.com/en-us/file/compute-architecture-of-intel-processor-graphics-gen8pdf, September 2014, accessed: 2015-04-19.

[60] JEDEC, "High Bandwidth Memory (HBM) DRAM," JEDEC, Tech. Rep. JESD235A, Nov 2015.

[61] Min Kyu Jeong, Chander Sudanthi, Nigel Paver, and Mattan Erez, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *Proceedings of the Design Automation Conference (DAC)*, June 2012.

[62] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *Proceedings of the International Conference on Supercomputing (ICS)*. New York, NY, USA: ACM, 2012, pp. 15–24.

[63] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 272–283.

[64] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU

Performance," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2013, pp. 395–406.

[65] Sven Karlsson and Mats Brorsson, "A Comparative Characterization of Communication Patterns in Applications Using MPI and Shared Memory on an IBM SP2," in *Network-Based Parallel Computing Communication, Architecture, and Applications*, ser. Lecture Notes in Computer Science, Dhabaleswar K. Panda and Craig B. Stunkel, Eds. Springer Berlin Heidelberg, 1998, vol. 1362, pp. 189–201.

[66] Stefanos Kaxiras and James R. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, January 1999, pp. 161–170.

[67] Abdullah Kayi, Olivier Serres, and Tarek El-Ghazawi, "Adaptive Cache Coherence Mechanisms with Producer-Consumer Sharing Optimization for Chip Multiprocessors," *IEEE Transactions on Computers (preprints)*, 2013.

[68] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2014.

[69] Khronos Group, "The OpenCL Specification," https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf, October 2014, accessed: 2015-04-20.

[70] Hyunhee Kim and Jihong Kim, "A Leakage-aware L2 Cache Management Technique for Producer-consumer Sharing in Low-power Chip Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 71, no. 12, pp. 1545–1557, December 2011.

[71] Ji Kim and Christopher Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2014.

[72] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 65–76.

[73] Lindsay Kleeman and Antonio Cantoni, "The Analysis and Performance of Batching Arbiters," in *Proceedings of the 1986 ACM SIGMETRICS Joint International Conference on Computer Performance Modelling, Measurement and Evaluation*. New York, NY, USA: ACM, 1986, pp. 35–43.

[74] Géraud Krawezik, "Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors," in *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*. New York, NY, USA: ACM, 2003, pp. 118–127.

[75] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula, "Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2015, pp. 733–745.

[76] Alvin R. Lebeck and David A. Wood, "Dynamic Self-invalidation: Reducing Coherence Overhead in Shared-memory Multiprocessors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 48–59.

[77] Jaekyu Lee and Hyesoon Kim, "TAP: A TLP-aware Cache Management Policy for a CPU-GPU Heterogeneous Architecture," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2012, pp. 1–12.

[78] Shin-Ying Lee and Carole-Jean Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. New York, NY, USA: ACM, 2014, pp. 175–186.

[79] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey, "Debunking the 100X GPU vs. CPU myth: an Evaluation of Throughput Computing on CPU and GPU," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2010, pp. 451–460.

[80] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak, "The Network Architecture of the Connection Machine CM-5 (Extended Abstract)," in *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1992, pp. 272–285.

[81] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 2013, pp. 487–498.

[82] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis, "Comparing Memory Systems for Chip Multiprocessors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2007, pp. 358–368.

[83] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2009, pp. 469–480.

[84] Chunhua Liao, Yonghong Yan, Bronis R. de Supinski, Daniel J. Quinlan, and Barbara Chapman, "Early Experiences with the OpenMP Accelerator Model," in *Proceedings of the International Workshop on OpenMP (IWOMP)*, Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, September 2013, pp. 84–98.

[85] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta, "Efficient Sequential Consistency Using Conditional Fences," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2010, pp. 295–306.

[86] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta, "Address-aware Fences," in *Proceedings of the International Conference on Supercomputing (ICS)*, June 2013, pp. 313–324.

[87] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot, "Value Prediction and Speculative Execution on GPU," *International Journal of Parallel Programming*, vol. 39, no. 5, pp. 533–552, 2011.

[88] Yuxi Liu, Zhibin Yu, Lieven Eeckhout, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Chengzhong Xu, and Vijay Janapa Reddi, "Barrier-Aware Warp Scheduling for Throughput Processors," in *Proceedings of the International Conference on Supercomputing (ICS)*. New York, NY, USA: ACM, 2016.

[89] Gabriel H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 453–464.

[90] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers Via Sensible Co-locations," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2011, pp. 248–259.

[91] José F. Martínez and Josep Torrellas, "Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2002, pp. 18–29.

[92] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai, "Managing Shared Last-level Cache in a Heterogeneous Multicore Processor," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2013, pp. 225–234.

[93] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam, "iGPU: Exception Support and Speculative Execution on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 72–83.

[94] Mitesh R. Meswani, Laura Carrington, Didem Unat, Allan Snavely, Scott B. Baden, and Stephen Poole, "Modeling and Predicting Performance of High Performance Computing Applications on Hardware Accelerators," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 27, pp. 89–108, 2012.

[95] Paulius Micikevicius, "GPU Performance Analysis and Optimization," in *GPU Technology Conference*, 2012.

[96] Shubhendu S. Mukherjee and Mark D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1998, pp. 179–190.

[97] Naveen Muralimanohar and Rajeev Balasubramonian, "CACTI 6.0: A Tool to Understand Large Caches," 2007.

[98] Vijay Nagarajan and Rajiv Gupta, "Speculative Optimizations for Parallel Programs on Multicores," in *Proceedings of the International Conference on Languages and Compilers for Parallel Computing (LCPC)*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 323–337.

[99] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2011, pp. 308–317.

[100] Girija J. Narlikar and Guy E. Blelloch, "Pthreads for Dynamic and Irregular Parallelism," in *Proceedings of the International Conference on High Performance Networking and Computing (SC)*, November 1998.

[101] Rupesh Nasre, Martin Burtscher, and Keshav Pingali, "Data-Driven Versus Topology-driven Irregular Computations on GPUs," in *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)*, May 2013, pp. 463–474.

[102] NVIDIA, *GPU Gems 3*. NVIDIA Corporation, 2007.

[103] NVIDIA, "Dynamic Parallelism in CUDA," http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf, 2012.

[104] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[105] NVIDIA, "NVIDIA Brings Kepler, World's Most Advanced Graphics Architecture, to Mobile Devices," http://blogs.nvidia.com/blog/2013/07/24/kepler-to-mobile/, July 2013.

[106] NVIDIA, "Unified Memory in CUDA 6," http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, November 2013, accessed: 2015-04-03.

[107] NVIDIA, "CUDA C/C++ SDK Code Samples," http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2014.

[108] NVIDIA, "NVIDIA GeForce GTX 980," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, 2014, accessed: 2015-07-30.

[109] NVIDIA, "Summit and Sierra Supercomputers: An Inside Look at the U.S. Department of Energy's New Pre-Exascale Systems," November 2014.

[110] NVIDIA Corporation, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, March 2015, accessed: 2015-07-20.

[111] NVIDIA Corporation, "Parallel Thread Execution ISA," http://docs.nvidia.com/cuda/pdf/ptx_isa_4.2.pdf, March 2015, accessed: 2015-07-20.

[112] OpenACC-Standard.org, "The OpenACC Application Programming Interface, Version 2.5," http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, October 2015, accessed: 2016-08-03.

[113] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood, "Fine-grain Task Aggregation and Coordination on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 181–192.

[114] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, M. Pedram, and Naehyuck Chang, "Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 5, pp. 695–708, May 2013.

[115] Indrani Paul, Wei Huang, Manish Arora, and Sudhakar Yalamanchili, "Harmonia: Balancing Compute and Memory Power in High-performance GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 54–65.

[116] J. Thomas Pawlowski, "Hybrid Memory Cube (HMC)," in *Hot Chips*, Aug 2011.

[117] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014, pp. 743–758.

[118] Louis-Noel Pouchet, "PolyBench/GPU," http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/, 2012.

[119] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2013, pp. 457–467.

[120] Jason Power, Joel Hestness, Marc Orr, Mark Hill, and David Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *IEEE Computer Architecture Letters*, vol. 13, no. 1, January 2014.

[121] Jason Power, Mark D. Hill, and David A. Wood, "Supporting x86-64 Address Translation for 100s of GPU Lanes," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 568–578.

[122] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens, "Memory Access Scheduling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 128–138.

[123] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2012, pp. 72–83.

[124] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel, "PTask: Operating System Abstractions To Manage GPUs as Compute Devices," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, October 2011, pp. 233–248.

[125] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2008, pp. 73–82.

[126] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2006, pp. 235–246.

[127] John Sartori and Rakesh Kumar, "Low-overhead, High-speed Multi-core Barrier Synchronization," in *High Performance Embedded Architectures and Compilers*. Springer, 2010, pp. 18–34.

[128] Steven L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996, pp. 26–36.

[129] John Sell and Patrick O'Connor, "The Xbox One System on a Chip and Kinect Sensor," *IEEE Micro*, vol. 34, no. 2, pp. 44–53, March 2014.

[130] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2002, pp. 29–40.

[131] Ankit Sethia and Scott Mahlke, "Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec 2014, pp. 647–658.

[132] Shisheng Shang and Kai Hwang, "Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 591–605, June 1995.

[133] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve, "Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2015, pp. 647–659.

[134] Inderpreet Singh, Arrvindh Shriraman, Wilson W.L. Fung, Mike O'Connor, and Tor M. Aamodt, "Cache Coherence for GPU Architectures," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2013, pp. 578–590.

[135] Karandeep Singh, John Paul Walters, Joel Hestness, Jinwoo Suh, Craig M. Rogers, and Stephen P. Crago, "FFTW and Complex Ambiguity Function Performance on the Maestro Processor," in *Aerospace Conference*, IEEE. IEEE, 2011, pp. 1–8.

[136] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton, "Continual Flow Pipelines," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2004, pp. 107–119. [Online]. Available: http://doi.acm.org/10.1145/1024393.1024407

[137] Per Stenström, Mats Brorsson, and Lars Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, May 1993, pp. 109–118.

[138] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[139] Magnus Strengert, Martin Kraus, and Thomas Ertl, "Pyramid Methods in GPU-based Image Processing," in *Proceedings of Vision, Modeling, and Visualization*, November 2006, pp. 169–176.

[140] Jeff A. Stuart and John D. Owens, "Efficient Synchronization Primitives for GPUs," *Computing Research Repository (CoRR)*, vol. abs/1110.4623, 2011.

[141] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero, "Enabling Preemptive Multiprogramming on GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2014, pp. 193–204.

[142] Real World Tech, "Inside Fermi: NVIDIA's HPC Push," http://www.realworldtech.com/fermi/5/, September 2009, accessed: 2016-08-01.

[143] William Thies, Michal Karczmarek, and Saman P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the International Conference on Compiler Construction (CC)*, April 2002, pp. 179–196.

[144] Mary K. Vernon and Udi Manber, "Distributed Round-robin and First-come First-serve Protocols and Their Applications to Multiprocessor Bus Arbitration," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 269–279.

[145] Mohamed Wahib and Naoya Maruyama, "Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. New York, NY, USA: ACM, 2015, pp. 259–270.

[146] Guibin Wang, YiSong Lin, and Wei Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *International Conference on Green Computing and Communications*, December 2010, pp. 344–350.

[147] Jin Wang and Sudhakar Yalamanchili, "Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2014.

[148] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting Concurrent Kernel Execution on Graphic Processing Units," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, July 2011, pp. 24–32.

[149] Bruce Wile, "Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems," September 2014.

[150] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1995, pp. 24–36.

[151] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar, "Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission," in *International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, May 2012.

[152] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens, "Performance Characterization of High-Level Programming Models for GPU Graph Analytics," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Oct 2015, pp. 66–75.

[153] Shucai Xiao and Wu chun Feng, "Inter-block GPU Communication via Fast Barrier Synchronization," in *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[154] Hong Xu, Philip K. McKinley, and Lionel M. Ni, "Efficient Implementation of Barrier Synchronization in Wormhole-Routed Hypercube Multicomputers," in *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, June 1992, pp. 118–125.

[155] Yonghong Yan, Sanjay Chatterjee, Daniel A. Orozco, Elkin Garcia, Zoran Budimlić, Jun Shirako, Robert S. Pavel, Guang R. Gao, and Vivek Sarkar, "Hardware and Software Tradeoffs for Task Synchronization on Manycore Architectures," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, September 2011, pp. 112–123.

[156] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2010, pp. 86–97.

[157] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim, "Profiling Network Performance for Multi-tier Data Center Applications," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NDSI)*, March 2011, pp. 57–70.