© 2017 Matthew David Sinclair

EFFICIENT COHERENCE AND CONSISTENCY FOR SPECIALIZED MEMORY HIERARCHIES

BY

MATTHEW DAVID SINCLAIR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Sarita V. Adve, Chair & Director of Research
Professor Vikram S. Adve
Professor Todd Austin, University of Michigan
Dr. Bradford Beckmann, AMD Research
Professor David Brooks, Harvard University
Professor Rob A. Rutenbar
Professor Marc Snir

Abstract

As the benefits from transistor scaling slow down, specialization is becoming increasingly important for a wide range of applications. Although traditional heterogeneous systems work well for streaming, data parallel applications, they are inefficient for emerging applications, like graph analytics workloads, with fine-grained synchronization, relaxed atomics, and more general sharing patterns. Heterogeneous systems are also difficult to program, which makes it harder for programmers to take advantage of the potential benefits of specialization.

This thesis redesigns the memory hierarchy of heterogeneous systems to make heterogeneous systems more efficient and easier to use. In particular, we focus on three key sources of inefficiency in the memory hierarchy of modern heterogeneous systems: (1) a unified global address space, (2) the cache coherence protocol, and (3) the memory consistency model.

A unified global address space makes it easier to write programs for heterogeneous systems. Although industry has recently begun to provide a unified global address space across CPUs and accelerators (primarily GPUs), there are many inefficiencies. For example, emerging applications with fine-grained synchronization need better support for coherence and consistency. We find that simple coherence and complex consistency are key sources of inefficiency. To resolve this problem, we adjust the division of complexity between the cache coherence protocol and memory consistency model: we introduce DeNovo for accelerators (DeNovoA), which extends DeNovo's hybrid, software-driven hardware coherence protocol to heterogeneous systems. Unlike current coherence protocols for heterogeneous systems, DeNovoA obtains ownership for written data, enables heterogeneous systems to use the simpler sequentially consistent for data-race-free (SC-for-DRF, or DRF) memory consistency model, and provides both efficiency and programmability. Across a wide variety of applications, DeNovoA with a DRF memory consistency model either outperforms or provides comparable efficiency to a the state-of-the-art approach.

Although DRF is easier to use and works well for most applications, there are some corner cases where its overheads are unnecessary and hurt performance. This led to the introduction of relaxed atomics in the memory consistency models for multi-core CPUs and heterogeneous systems. Although relaxed atomics can significantly improve performance, they are very difficult to use correctly. We address the impact of relaxed atomics on memory consistency models for heterogeneous systems by creating a new memory consistency model, *Data-Race-Free-Relaxed* or *DRFrlx*. DRFrlx extends the existing DRF memory consistency models to provide SC-centric semantics for all common uses of relaxed atomics in heterogeneous systems *while* retaining their efficiency benefits. Thus, DRFrlx makes it easier for programmers to safely use relaxed atomics.

Although current heterogeneous systems are adopting unified global address spaces, specialized memories such as scratchpads still exist in disjoint, private address spaces. This increases programming complexity and causes inefficiencies that negate some of the benefits of specialization. We introduce a new memory organization, *stash*, that mitigates the inefficiencies of specialized memories by integrating them into the coherent, globally visible address space. Stash makes it easier for programmers to use specialized memories and retains their efficiency benefits.

Finally, to better understand the tradeoffs and scalability of different coherence protocols and consistency models, we created a suite of synchronization microbenchmarks, HeteroSync. HeteroSync contains various fine-grained synchronization and relaxed atomics algorithms. Moreover, HeteroSync is highly configurable and provides a standard set of fine-grained synchronization microbenchmarks to compare the efficiency of different approaches.

In summary, this thesis questions the state-of-the-art approaches for designing memory hierarchies of heterogeneous systems, and shows that the current techniques provide neither efficiency nor programmability for emerging workloads. We demonstrate how DeNovoA with a DRFrlx memory consistency model improves efficiency and programmability for many heterogeneous applications and makes it easier for programmers to use heterogeneous systems.

To my Dad, who isn't around to see this day. I know you're looking down on me and smiling.

To my Mom, for always supporting me.

To Ann, for her constant encouragement.

Acknowledgments

I owe a debt of gratitude to a number of people who have helped me along the way. Most of all, I need to thank my advisor, Sarita Adve, for all of her help along the way. Working with Sarita has been a wonderful experience, and I am a better researcher and person for having worked with her. Her drive, patience, and commitment to excellence are inspiring, and things I strive to emulate moving forward.

Vikram Adve has also provided me with much sage advice along the way. No matter how busy he was, Vikram always had time to listen to my questions and offer constructive feedback. Similarly, despite their very busy schedules, Rob Rutenbar and Marc Snir always made me feel welcome and offered input from a different perspective that challenged me and made me think more critically about my work.

I also sincerely appreciate the input and feedback from the rest of my Ph.D. committee, Todd Austin, Brad Beckmann, and David Brooks, which significantly improved the quality of my work. Special thanks as well to Brad, Pablo Montesinos, Mike O'Connor, and Steve Keckler, among others, who exposed me to real world problems for heterogeneous systems.

Sarita's research group is full of wonderful, talented, and kind students. Working with them has been a wonderful experience for me and helped me grow in many different facets. I am especially thankful to Rakesh Komuravelli and Hyojin Sung for their collaboration and guidance during the stash project, Siva Hari for his kind, patient mentoring, and John Alsop for all of his help with the scopes, relaxed atomics, and HeteroSync projects. Additionally, I am thankful to my other collaborators and/or lab-mates: Khalique Ahmed, Lin Cheng, Byn Choi, Adel Ejjeh, Muhammad Huzaifa, Maria Kotsifakou, Abdulrahman Mahmoud, Sean Nicolay, Giordano Salvador, Rob Smolinski, Prakalp Srivastava, and Radha Venkatagiri.

The staff in the Computer Science department at Illinois is top notch, and they have also

helped me significantly along the way. Special thanks to Rhonda McElroy, both for her help in the department and with the Mavis Future Faculty Fellows program, and Colin Robertson, for all of his help with the fellowships and awards. Similarly, I am indebted to Dana Garard, Molly Flesner, Joe Jeffries, Michelle Osbourne, Meg Osfar, Tierra Reed, and Sherry Unkraut, not just for their help in completing various administrative tasks, but also for the interesting conversations and discussions. Finally, thanks to Julie Gustafson, Erin Henkelman, Mary Beth Kelley, Viveka Kudaligama, Kara MacGregor, and Kathy Ann Runck for making it easy for me to succeed.

Many, many thank yous to the TSG/Engineering IT staff for all of their patient help along the way, especially to Palmer Buss, Joel Franzen, Glen Rundblom, and Vincent Weathers. Without their help, none of the equipment I relied on daily would have worked properly.

My work has been supported by Intel through the Illinois/Intel Parallelism Center at Illinois, by the National Science Foundation under grants CCF 10-18796, CCF 13-02641, and CCF 16-19245 and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. My work was also supported by a Qualcomm Innovation Fellowship. Extra special thanks to Dr. Matsushita, Mrs. Poppelbaum, and Dr. and Mrs. Mavis for endowing the Saburo Muroga Fellowship, C.W. Poppelbaum Award, and Mavis Future Faculty Fellowship, respectively. I am honored to have been a recipient of these awards and fellowships.

Along the way I was lucky enough to meet a great group of people in Urbana-Champaign, who enriched my life significantly. A special thanks to Robert Deloatch and Ryan Musa for their friendship. Similarly, I am grateful to Henry Duwe, both for his friendship and for his sage advice.

I am also grateful to my family for providing support and encouragement along the way. I am very lucky to have grown up in a nurturing environment where learning was encouraged. Finally, thanks to my wife Ann. Without you, I would never have succeeded: whenever I was tired you were there for me. I proudly dedicate this thesis to you.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AoS	Array of Structures
BC	Betweenness Centrality
BP	Backprop
CU	Compute Unit
DeNovoA	DeNovo for Accelerators
DD	DeNovoA coherence, DRF consistency
DD0	DeNovoA coherence, DRF0 consistency
DD1	DeNovoA coherence, DRF1 consistency
DDR	DeNovoA coherence, DRFRlx consistency
DD+RO	DD with read-only optimization
DH	DeNovoA coherence, DRF consistency
DMA	Direct Memory Access
FAM	Fetch-and-Add Mutex
GD	GPU coherence, DRF consistency
GD0	GPU coherence, DRF0 consistency
GD1	GPU coherence, DRF1 consistency
GDR	GPU coherence, DRFrlx consistency
GH	GPU coherence, HRF consistency
GPGPU	General-purpose GPU
GPU	Graphics Processing Unit
GSI	GPU Stall Inspector
H	Histogram

HG	Histogram Global
HG-2K	Histogram Global with 2K bins
HG-NO	Histogram Global-Non-Ordering
hLRC	heterogeneous Lazy Release Consistency
HSA	Heterogeneous System Architecture
hVISC	heterogeneous Virtual Instruction Set Computing
HW	Hardware
Lava	LavaMD
LFTRB	Lock-Free Tree Barrier
LFTRBEX	Lock-Free Tree Barrier with local data exchange
LLC	Last Level Cache
LUD	LU Decomposition
ML	Multiple Locks
NN	Nearest Neighbor
NW	Needlman-Wunsch
PF	Pathfinder
PR	PageRank
RC	Reference Counter
RO	Read-Only
RTLB	Reverse Translation Lookaside Buffer
SC	Sequential Consistency
SC for DRF	Sequentially consistent for data-race-free
DRF	SC for DRF
SC for DRF0	Sequentially consistent for data-race-free-0
DRF0	SC for DRF0
SC for DRF1	Sequentially consistent for data-race-free-1
DRF1	SC for DRF1
SC for DRFrlx	Sequentially consistent for data-race-free-relaxed
DRFrlx	Data-race-free-relaxed

SC for HRF	Sequentially consistent for heterogeneous-race-free
HRF	SC for HRF
SEQ	Seqlocks
SLM	Sleep Mutex
SoC	System-on-a-Chip
SPC	Split Counter
SPM	Spin Mutex
SPMBO	Spin Mutex with Backoff
SS	Spin Semaphore
SSBO	Spin Semaphore with Backoff
ST	Stencil
SyncPrims	Synchronization Primitives
TB	Thread Block
TRB	Tree Barrier
TRBEX	Tree Barrier with local data exchange
TLB	Translation Lookaside Buffer
UTS	Unbalanced Tree Search

Chapter 1

Introduction

1.1 Motivation

For many years hardware designers used the increasing number of transistors provided by Moore's Law to create complex single core processors that were both faster **and** smaller than their predecessors. However, diminishing returns from optimizing single-threaded performance and the end of Dennard's scaling meant that designers could no longer design energy efficient single core processors that ran faster without consequence [11]. In response, hardware manufacturers switched from creating very fast, single core CPUs to multi-core CPUs where each core runs at a slower frequency. Although this transition allowed the hardware manufacturers to continue doubling the number of transistors on chip every two years according to the "multi-core" Moore's Law [75], subsequent work showed that multi-core scaling is limited by power constraints [58].

Heterogeneous systems with specialized compute units have emerged as one potential solution because specialization offers a natural path to energy efficiency. However, current data movement mechanisms in heterogeneous systems are very inefficient, especially for emerging applications such as graph analytics workloads and applications that use specialized memories. Additionally, heterogeneous systems are difficult to program, which makes it hard to take advantage of the benefits specialization provides. Since the memory hierarchy is expected to become a dominant consumer of overall energy [56, 86], efficient data movement is essential for efficient heterogeneous systems.

In this thesis, we redesign the memory hierarchy of heterogeneous systems. Unlike current systems, we provide performance- and energy-efficiency and programmability for emerging workloads. In particular, we focus on three key sources of inefficiency: (1) loosely coupled memory hierarchies, (2) the cache coherence protocol, and (3) the memory consistency model.

Traditional heterogeneous systems had loosely coupled memory hierarchies and required pro-

grammers to explicitly copy data between different accelerators via main memory to keep data coherent. In an effort to make heterogeneous systems more programmable, industry has recently transitioned to more tightly coupled heterogeneous systems with a unified global address space and coherent caches across CPUs and accelerators (primarily GPUs) [18, 78, 79, 135]. A global address space allows data to be transparently moved between accelerators in hardware and provides high performance for the simple, streaming, data parallel applications that heterogeneous systems traditionally run. Since these applications have little or no sharing or data reuse, heterogeneous systems use simple, software-driven coherence protocols that assume data-race-freedom, regular data accesses, and mostly coarse-grained synchronization. These protocols invalidate the entire cache at acquires and flush (writethrough) all dirty data before the next release [71]. Since synchronization (implemented with *atomics*) is infrequent, synchronization accesses bypass the private caches and are executed at the next shared level of the hierarchy.

Thus, unlike conventional multi-core CPU coherence protocols, conventional heterogeneous coherence protocols are very simple, without need for writer-initiated invalidations, ownership requests, downgrade requests, protocol state bits, or directories (Section 2.3). Further, although consistency models for heterogeneous systems have been slow to be clearly defined [13, 104, 143, 144], heterogeneous coherence protocol implementations were amenable to the familiar sequentially consistent for data-race-free (SC-for-DRF, or DRF) consistency model, such as the sequentially consistent for data-race-free-0 (SC-for-DRF0, or DRF0) model widely adopted for multi-cores today [4]. DRF0 allows programmers to reason with the familiar SC model as long as there are no data races.

Although simple heterogeneous coherence protocols work well for traditional heterogeneous applications, emerging applications with more general sharing patterns and fine-grained synchronization [38, 42, 77, 87, 121, 146] suffer because heterogeneous systems are not designed with these traits in mind. For these applications, full cache invalidates, dirty data flushes, and remote execution of synchronizations are extremely inefficient. Multi-core CPUs use hardware coherence protocols, such as MESI, to overcome this problem but prior work has observed that hardware coherence protocols are a poor fit for conventional heterogeneous applications [71, 141].

In an attempt to improve performance for these emerging applications without overly complicating the coherence protocol, recent work has introduced a new consistency model, sequentially

consistent for heterogeneous-race-free (SC-for-HRF, or HRF). HRF uses scoped synchronization to associate a synchronization access with a level of the memory hierarchy where the synchronization should occur [63, 77]. For example, a synchronization access with a local scope indicates that it synchronizes only the data accessed by the threads within its own compute unit (CU, which all share the L1 cache). As a result, the synchronization can execute at the CU's L1 cache, without invalidating or flushing data to lower levels of the memory hierarchy. Thus, locally scoped synchronizations can significantly improve performance.

Although the introduction of scopes is an efficient solution to the problem of fine-grained synchronization in emerging heterogeneous applications, it increases programming complexity even further. Intrinsically scopes are a hardware-inspired mechanism that expose the memory hierarchy to the programmer. Previously, researchers had argued against such a hardware-centric view and proposed more software-centric models such as DRF0 [4]. Although DRF0 is widely adopted, it is still a source of much confusion [6]. Viewing the subtleties and complexities associated even with the so-called simplest models, we argue that GPU consistency models should not be even more complex than the CPU models. Moreover, scoped synchronization only helps when the programmer (or compiler) can identify the scope as local – if the programmer (or compiler) cannot determine this, then scoped synchronization offers no additional benefits over the traditional approach.

A second source of complexity is *relaxed atomics*, which make the DRF and HRF consistency models even more complicated. DRF0 provides high performance and programmability for many applications. However, DRF0 imposes strict constraints on all synchronization (used interchangeably with atomics) accesses. For some applications, such as graph analytics workloads, these overheads are too high and unnecessary. As a result, relaxed atomics were introduced in CPU and heterogeneous consistency models. Relaxed atomics relax the ordering constraints DRF0 imposes on all atomics to improve performance. However, relaxed atomics further increase programming complexity, because they violate the SC semantics that DRF0 and HRF provide. Consequently, it is very difficult for programmers to reason about the correctness of their code.

Moreover, relaxed atomics are extremely difficult to formalize [6, 29, 36, 37] and use correctly, so their use is strongly discouraged [36, 154]. Nevertheless, relaxed atomics are still used in modern heterogeneous systems. Relaxed atomics potentially provide significant efficiency improvements in

heterogeneous systems because heterogeneous systems traditionally assume that synchronizations occur infrequently, which is not the case for emerging workloads. Consequently, the traditional approach is extremely inefficient for these workloads – which makes using relaxed atomics very attractive in heterogeneous systems.

Finally, modern heterogeneous systems also use specialized memories like scratchpads to improve efficiency. Scratchpads are software-managed and directly addressed, so they avoid the overheads the more general-purpose caches face. However, scratchpads are unaffected by the recent adoption of a unified address space, because they use disjoint, private address spaces. Specifically, scratchpads are not globally addressable or visible, so they must eagerly, explicitly transfer data between the global address space and the scratchpads' private address space, which are difficult to program for and negates some of the benefits of using scratchpads.

This state of affairs led us to ask the following questions: *How can we have a global address space with simple coherence and consistency?* And: *can specialized memories be made part of this global address space?* This work achieves these goals by adjusting the division of complexity between the coherence protocol and consistency model. First, we introduce a new coherence protocol for heterogeneous systems, DeNovo for accelerators (DeNovoA), that extends the DeNovo coherence protocol for multi-core CPUs [46, 152, 153]. DeNovoA obtains ownership for written data and uses self-invalidations to invalidate potentially stale data. Obtaining ownership for written data and synchronization variables allows DeNovoA to improve performance of applications with fine-grained synchronization without the complexity of scoped synchronization. Instead, DeNovoA can use the simpler, standard DRF0 consistency model. Although DeNovoA's underlying concepts are not new, the insight that obtaining ownership for dirty data and self-invalidating valid data enables heterogeneous systems to balance efficiency and programmability is new. Next, we create a new memory consistency model, *Data-Race-Free-Relaxed* or *DRFrlx*, that extends the existing DRF consistency models to provide SC-centric semantics for all common use cases of relaxed atomics in heterogeneous systems *and* retains their efficiency benefits. The key insight behind DRFrlx is to focus on how programmers actually want to use relaxed atomics in real applications, instead of trying to reason about every possible use of relaxed atomics. Finally, we extend DeNovoA to integrate specialized memories into the global address space with low overhead. Here our key

insight is to use software information to create an efficient address mapping for the scratchpad in hardware. Overall, these changes makes heterogeneous systems more efficient and easier to use, especially for emerging applications like graph analytics workloads.

1.2 Summary of Contributions

Overall we make the following contributions:[1]

Efficient Coherence and Consistency for Heterogeneous Systems: We demonstrate that DeNovoA is a viable coherence protocol for heterogeneous CPU-GPU systems, although the ideas are also applicable to other accelerators. DeNovoA is able to exploit reuse of written data and synchronization variables across synchronization boundaries without the additional complexity of scopes. The results show that DeNovoA with a DRF consistency model significantly outperforms conventional GPU coherence with a DRF consistency model across a wide range of conventional and emerging applications. After enhancing GPU coherence with HRF's scoped synchronization, DeNovoA with DRF provides much better performance for microbenchmarks with globally scoped fine-grained synchronization (on average[2] 21% lower execution time and 45% lower energy). For microbenchmarks with mostly locally scoped synchronization, GPU+HRF does slightly better – on average 6% lower execution time and 4% lower energy. Enhancing DeNovoA with DRF to avoid invalidating valid, read-only data at acquires reduces this performance gap and provides the same performance and energy as GPU+HRF on average. For the cases where HRF's complexity is deemed acceptable, we also develop a version of DeNovoA that uses the HRF consistency model; DeNovoA with an HRF consistency model is the best performing protocol.

Efficient Support for and Evaluation of Relaxed Atomics: We make two important contributions. First, to understand how relaxed atomics are used in heterogeneous systems, we collected examples of how developers use relaxed atomics [28, 36, 39, 63, 154] and characterized how they used relaxed atomics to identify how their use cases could be fit into an SC-centric framework. In some use cases, the existing sequentially consistent for data-race-free-1 (SC-for-DRF1, or DRF1)

[1] I did this work in collaboration with other students at UIUC. I was the lead author for the coherence and consistency [138], relaxed atomics [139] and HeteroSync [140] work. I co-led the specialized memory work [91] work with Rakesh Komuravelli. Johnathan Alsop led the stall profiling work [16]. Chapters 2 - 6 are heavily based on the publications I led and co-led.

[2] We use arithmetic mean to represent the average throughout this thesis.

consistency model [8] can provide the same benefits as relaxed atomics without compromising SC semantics. However the remaining use cases benefit from using relaxed atomics in ways that may violate SC semantics. To handle these use cases, we propose a new consistency model, DRFrlx. DRFrlx extends DRF0 and DRF1 to provide SC-centric semantics for all common use cases of relaxed atomics in heterogeneous systems without affecting the performance benefits of relaxed atomics.

Second, we performed an evaluation to determine how beneficial relaxed atomics are in modern heterogeneous systems. In most cases the results show that DRF1 and DRFrlx provide only marginal benefit (on average, 4% execution time reduction for GPU coherence and 9% for DeNovoA). However, for two applications (BC and PageRank), DRF1 and DRFrlx's benefits were significant – depending on the input, DRFrlx reduces execution time up to 52% for DeNovoA and up to 51% for GPU coherence. The choice of coherence protocol also affects efficiency: by exploiting locality in atomic and written data, DeNovoA outperforms GPU coherence for DRF0, DRF1, and DRFrlx.

Integrating Specialized Memories Into the Unified Address Space: We introduce a new memory organization, **stash**, that combines the benefits of caches and scratchpads. Like a scratchpad, the stash provides compact storage and does not have overheads from indirect addressing. Like a cache, the stash is globally addressable and visible, enabling implicit and on-demand data movement and increased data reuse. We take advantage of the low overhead DeNovoA coherence protocol to efficiently integrate stash into the global address space. The results show that the stash effectively combines the benefits of scratchpads and caches. For microbenchmarks designed to exploit new use cases that the stash enables, on average, the stash reduces execution time and consumes less energy than the scratchpad, cache, and DMA configurations – 13%, 27%, and 14% lower execution time, respectively and 35%, 53%, and 32% less energy, respectively. For full-sized applications, stash improves both performance and energy: compared to the best scratchpad and cache versions, on average stash reduces execution time by 10% and 12% (max 22% and 31%), respectively, while decreasing energy by 16% and 32% (max 30% and 51%), respectively.

HeteroSync: Benchmark Suite for Fine-Grained Synchronization: We have also made several methodological contributions that underlie this thesis. First, we created a suite of microbench-

marks, HeteroSync. HeteroSync includes microbenchmarks implementing various synchronization primitives, along with annotations for locally and globally scoped atomics (Section 2.6), as well as relaxed atomics (Section 3.3).

We use the HeteroSync microbenchmarks to examine the scalability of various synchronization algorithms, coherence protocols, and memory consistency models for tightly coupled CPU-GPU systems in Chapters 2 and 3. For locally scoped microbenchmarks, DeNovoA with DRF and GPU coherence with HRF scale much better than the GPU coherence with DRF. For the hybrid and globally scoped SyncPrims, DeNovoA with DRF scales better than all other configurations. The relaxed atomics microbenchmarks show mixed scalability results: for some microbenchmarks relaxed atomics improve (strong) scalability, while for others they increase execution time, only provide small benefits, or do not impact the scalability.

Additionally, I helped develop GPU Stall Inspector (GSI), a new, detailed profiling framework that characterizes the sources of memory stalls in tightly coupled CPU-GPU systems. Unlike existing profiling tools [20, 95, 120], which do not accurately capture the subtle interactions that occur between memory requests in tightly coupled systems, GSI provides accurate profiling information about memory stalls and helps identify the bottlenecks in applications.

1.3 Long-Term Impact

Modern heterogeneous systems are inefficient and hard to use. State-of-the-art heterogeneous systems use simple, software-based cache coherence protocols and complex memory consistency models like HRF. This approach can provide good efficiency, but only if the scope is local – which is often not the case in emerging applications such as graph analytics workloads. Moreover, HRF's scoped synchronization increases programming complexity. This mirrors past efforts in multicore CPUs, which led to complicated, relaxed CPU memory consistency models. Eventually, the concurrency community adopted the DRF consistency model, but the legacy of using complex relaxed memory models still burdens DRF.

This work is the first to question whether this complexity is truly necessary, and shows that it is not. DeNovoA with a DRF consistency model effectively balances efficiency and programmability in heterogeneous systems while improving scalability and moving the complexity to the coherence

protocol, where it is better hidden from the programmer. As we discuss in Chapter 2, subsequent work in industry has built on our approach – which highlights the impact of our work.

Furthermore, the current approach to coherence and consistency in heterogeneous systems makes relaxed atomics even more appealing than they are in multi-core CPUs, because they can significantly improve efficiency. However, it is very difficult to use relaxed atomics correctly, because there are no acceptable formal semantics despite more than a decade of effort.

Unlike prior work, we focus on how programmers actually use relaxed atomics in heterogeneous systems. After examining numerous use cases and applications, we found that all uses of relaxed atomics in heterogeneous systems could be grouped into five use cases. Then we designed a new consistency model, DRFrlx, that extends existing DRF consistency models and provides SC-centric semantics for these uses while retaining their efficiency benefits. Thus, DRFrlx solves a long-standing, open problem in the concurrency community and makes it easier for everyone to use relaxed atomics safely.

Heterogeneous systems also use specialized memories like scratchpads. Scratchpads often improve efficiency for specific access patterns but require programmers to explicitly move data between the global address space and the scratchpads' private address space. Thus, specialized memories like scratchpads improve efficiency but are difficult to use. Programmers could always use easier-to-program caches instead, but these are often inefficient.

We introduce a new memory organization, stash, that shows how to integrate specialized memories into the unified address space while retaining their benefits. Stash combines the benefits of caches and scratchpads into a single memory organization and makes it easier for programmers to use specialized memories. As a result, programmers no longer need to choose either caches and scratchpads.

Finally, HeteroSync provides a common set of microbenchmarks that use various kinds of synchronization. Researchers can use HeteroSync to explore the differences between various fine-grained synchronization algorithms, coherence protocols, and consistency models. Overall, our work creates a memory hierarchy that is more efficient and easier to program than the state-of-the-art.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 describes the inefficient division of complexity between cache coherence protocols and memory consistency models in modern heterogeneous systems. In this chapter, we also describe the DeNovoA cache coherence protocol that we combine with a DRF memory consistency model to adjust this division of complexity and examine how the various approaches scale. Chapter 3 extends the DRF consistency model to provide better support for relaxed atomics. We analyze how relaxed atomics are used in heterogeneous systems, how they scale, and introduce DRFrlx to provide SC-centric semantics for all common uses of relaxed atomics in heterogeneous systems. In Chapter 4 we introduce the stash, and show how to extend DeNovoA to make scratchpads part of the unified global address space. In Chapter 5, we explore the scalability of the HeteroSync algorithms, coherence protocols and memory consistency models from Chapters 2 and 3. Chapter 6 qualitatively compares our work to prior work. Finally, Chapter 7 summarizes the contributions and provides some directions for future work.

Chapter 2

Efficient Coherence and Consistency for Heterogeneous Systems

2.1 Motivation

Traditionally, heterogeneous systems, especially GPUs, focused on data-parallel, mostly streaming, applications which had little or no sharing or data reuse between CUs. Thus, GPUs used very simple, software-driven coherence protocols that assume data-race-freedom, regular data accesses, and mostly coarse-grained synchronization (typically at GPU kernel boundaries). These protocols invalidate the cache at acquires (typically the start of the kernel) and flush (writethrough) all dirty data before the next release (typically the end of the kernel) [71]. The dirty data flushes go to the next level of the memory hierarchy shared between all participating cores and CUs (e.g., a shared L2 cache). Fine-grained synchronization (implemented with *atomics*) was expected to be infrequent and executed at the next shared level of the hierarchy (i.e., bypassing private caches). We refer to this approach as *GPU-style coherence*, or *GPU coherence*.

Thus, unlike conventional multi-core CPU coherence protocols, conventional GPU-style coherence protocols are very simple, without need for writer-initiated invalidations, ownership requests, downgrade requests, protocol state bits, or directories. Further, although GPU memory consistency models have been slow to be clearly defined [13, 104, 143, 144], GPU coherence implementations were amenable to the familiar data-race-free model widely adopted for multi-cores today.

Historically GPUs have been optimized for streaming, throughput optimized applications. However, the rise of general-purpose GPU (GPGPU) computing has made using GPUs desirable for applications with more general sharing patterns and fine-grained synchronization [38, 42, 146], a trend that has been adopted by both academia and industry. Unfortunately, conventional GPU-style coherence schemes, which perform full cache invalidations, dirty data flushes, and remote execution of synchronizations, are inefficient for these workloads. To overcome these inefficiencies,

recent work has proposed associating synchronization accesses with a scope that indicates the level of the memory hierarchy where the synchronization should occur [63, 77]. For example, a synchronization access with a local scope indicates that it synchronizes only the data accessed by the CUDA thread blocks (TBs)[1] within its own CU (which share the L1 cache). As a result, the synchronization can execute at the CU's L1 cache, without invalidating or flushing data to lower levels of the memory hierarchy (since no other CUs are intended to synchronize through this access). For synchronizations that can be identified as having local scope, this technique can significantly improve performance by eliminating virtually all sources of synchronization overhead.

Although the introduction of scopes is an efficient solution to the problem of fine-grained GPU synchronization, it comes at the cost of programming complexity. Traditionally, GPU coherence implementations were amenable to the familiar DRF model widely adopted for multi-cores today. Unfortunately, data-race-free is not a viable memory consistency model when scopes are used since scoped synchronization accesses potentially lead to "synchronization races" that can violate sequential consistency in non-intuitive ways (even for programs deemed to be well synchronized by the data-race-free memory model). Recently, Hower et al. addressed this problem by formalizing a new memory model, heterogeneous-race-free (HRF), to handle scoped synchronization in heterogeneous systems [77]. The Heterogeneous System Architecture (HSA) Foundation [78], a consortium of several industry vendors, and OpenCL 2.0 [96] recently adopted a model similar to HRF with scoped synchronization.

Although HRF is a very well-defined model, it cannot hide the inherent complexity of using scopes. Intrinsically, scopes are a hardware-inspired mechanism that expose the memory hierarchy to the programmer. Using memory models to expose a hardware feature is consistent with the past evolution of memory models (e.g., the IBM 370 and total store order (TSO) models essentially expose hardware store buffers), but is discouraging when considering the past confusion generated by such an evolution. Previously, researchers have argued against such a hardware-centric view and proposed more software-centric models such as data-race-free [4]. Although data-race-free is widely adopted, it is still a source of much confusion [6]. Considering the subtleties and complexities associated with even the so-called simplest models, we argue that GPU consistency models should

[1]For simplicity and without loss of generality we use NVIDIA's CUDA [118] terminology.

not be even more complex than CPU consistency models.

We show that the added complexity of scoped synchronization is not necessary. Instead, we use a coherence protocol for GPUs, DeNovoA, that is close in simplicity to conventional GPU coherence protocols, gives the performance benefits of scoped synchronization, and is amenable to using the data-race-free memory model. DeNovoA does not require writer-initiated invalidations or directories (similar to conventional GPU coherence mechanisms), but does obtain ownership for written data. By obtaining ownership, we are able to exploit reuse of written data and synchronization variables across synchronization boundaries, without the additional complexity of scopes.

Our work is the first to show that GPUs can efficiently support fine-grained synchronization at modest hardware overhead, without requiring the complexity of the HRF consistency model. Our results (Section 2.7) show that DeNovoA with DRF provides a sweet spot for performance, energy, overhead, and memory model complexity, questioning the recent move towards memory consistency models for GPUs that are more complex than those for CPUs. Moreover, by providing a low overhead, simple solution for coherence and consistency in heterogeneous systems, we enable additional work such as efficient work stealing in heterogeneous systems. Recently, a subset of the HRF authors have published a new paper that says scopes are not needed and use a coherence protocol similar in many respects to DeNovoA [15], which further demonstrates the impact of our work. We further address this and other related work in Section 6.1.2.

2.2 Background

In this section we discuss the state-of-the-art in coherence and consistency for heterogeneous systems.

2.2.1 GPU Coherence

In conventional GPU coherence protocols, synchronization happens infrequently and at a coarse granularity (e.g., kernel boundaries). As a result, GPUs use simple, software-driven coherence protocols that rely on data-race-freedom, invalidate the entire cache on load acquires, write-through all dirty data to the shared last level cache (LLC) on store releases, and require all synchronization accesses (performed with atomic operations) to execute at the LLC (e.g., the L2). While this

scheme provides high performance for conventional GPU applications, it is sub-optimal for emerging applications with fine-grained synchronization. We provide additional details on conventional GPU coherence in Section 2.3.

2.2.2 Memory Consistency Models

Depending on whether the coherence protocol uses scoped synchronization or not, we assume either DRF [4, 36, 109] or heterogeneous-race-free (HRF) [77] as the memory consistency model.

Modern memory consistency models for heterogeneous systems, like HSA, OpenCL, and HRF, are largely influenced by the decades of work on multi-core CPU memory models. Programming languages such as C, C++, and Java recently converged around the data-race-free-0 memory model which promises sequential consistency (SC) to data-race-free programs (SC-for-DRF0 or DRF0) [4, 36, 109].[2] The popularity of DRF0 stems from its SC-centric nature. Programmers can reason with the familiar SC model as long as there are no data races, and the absence of data races allows the system to exploit many optimizations without violating SC.

DRF0 requires programmers to distinguish between data and synchronization accesses – any access that may be involved in a race (in any SC execution) must be explicitly identified as synchronization using the atomic (for C, C++, OpenCL, HSA) or volatile (for Java) declarations. A program is data-race-free if its memory accesses are distinguished as data or synchronization, and, for all its SC executions, all pairs of conflicting data accesses are ordered by DRF0's happens-before relation. The happens-before relation is the irreflexive, transitive closure of program order and synchronization order, where the latter orders a synchronization write (release) before a synchronization read (acquire) if the write occurs before the read and the accesses conflict.

DRF0 allows the hardware and compiler to optimize data accesses, but imposes strict constraints on atomics (referred to as *SC atomics* because DRF0 requires atomics to be SC with one another). However, since atomics are relatively infrequent and data races are generally considered to be bugs [32], DRF0 provides a reasonable balance between performance and programmability.

HRF is defined similar to DRF0 except that each synchronization access has a scope attribute and HRF's synchronization order only orders synchronization accesses with the same scope. There

[2]The C, C++, and Java memory models also utilize some minor aspects of SC-for-DRF1 (DRF1). We describe DRF1 in Section 3.2.

	Invalidation Initiator	Tracking up-to-date copy	Supports different scopes?
Conventional HW	writer	ownership	yes
Software	reader	writethrough	yes
Hybrid	reader	ownership	yes

Table 2.1: Classification of protocols covering conventional HW (e.g., MESI), Software (e.g., GPU), and Hybrid (e.g., DeNovoA) coherence protocols.

are two variants of HRF: HRF-Direct, which requires all threads that synchronize to use the same scope, and HRF-Indirect, which builds on HRF-Direct by providing extra support for transitive synchronization between different scopes. One key issue that this creates the prospect of synchronization races – conflicting synchronization accesses to different scopes that are not ordered by HRF's happens-before. Such races are not allowed by the model and cannot be used to order data accesses.

Common implementations of DRF0 and HRF enforce a *program order requirement:* an access X must *complete* before an access Y if X is program ordered before Y and either (1) X is an acquire and Y is a data access, (2) X is a data access and Y is a release, or (3) X and Y are both synchronization. If HRF is being used, the synchronization must use the appropriate scope. For systems with caches, the underlying coherence protocol governs the program order requirement by defining what it means for an access to *complete*, as discussed in the next section.

2.3 A Classification of Coherence Protocols

In order to obtain a global address space with a simple coherence and consistency, one first needs to understand how coherence protocols operate. The end-goal of a coherence protocol is to ensure that a read returns the correct value from the cache. For the DRF and HRF models, this is the value from the last conflicting write as ordered by the happens-before relation for the model. Following the observations made for the DeNovo protocol for multi-core CPUs [46, 152, 153], we divide the task of a coherence protocol into the following:

(1) *No stale data*: A load hit in a private cache should never see stale data.

(2) *Locatable up-to-date data*: A load miss in a private cache(s) must know where to get the up-to-date copy.

Table 2.1 classifies three classes of cache coherence protocols in terms of how they enforce

these requirements. Modern coherence protocols accomplish the first task through invalidation operations, which may be initiated by the writer or the reader of the data. The responsibility for the second task is usually handled by the writer, which either registers its ownership (e.g., at a directory) or uses writethroughs to keep a shared cache up-to-date. The HRF consistency model adds an additional dimension of whether a protocol can be enhanced with scoped synchronization.

Although this taxonomy is by no means comprehensive, it covers the space of protocols commonly used in CPUs and GPUs as well as recent work on hybrid software-hardware protocols. We next describe example implementations from each class. Without loss of generality, we assume a two level cache hierarchy with private L1 caches and a shared last-level L2 cache. In a GPU, the private L1 caches are shared by all threads executing on the corresponding GPU CU.

Conventional Hardware Protocols used in CPUs

CPUs conventionally use pure hardware coherence protocols (e.g., MESI) that rely on writer-initiated invalidations and ownership tracking. They typically use a directory (or snoopy coherence if there are only a few cores) to maintain the list of (clean) sharers or the current owner of (dirty) data (at the granularity of a cache line). If a core issues a write to a line that it does not own, then it requests ownership from the directory, sending invalidations to any sharers or the previous owner of the line. For the purpose of invalidations and ownership, data and synchronization accesses (and any fences that may be required with the synchronization accesses) are typically treated uniformly. For the program order constraint described in Section 2.2.2, a write is complete when its invalidations reach all sharers or the previous owner of the line. A read completes when it returns its value and that value is globally visible.

Although such protocols have not been explored with the HRF memory model, it is possible to exploit scoped synchronization with them. However, the added benefits, are unclear. Furthermore, as discussed in Chapter 1, conventional CPU protocols are a poor fit for GPUs and are included here primarily for completeness.

Software Protocols used in GPUs

GPUs use simple, primarily software-based coherence mechanisms, without writer-initiated invalidations or ownership tracking. We first consider the protocols without scoped synchronization.

GPU protocols use reader-initiated invalidations. An acquire synchronization (e.g., atomic reads or kernel launches) invalidates the entire cache so future reads do not return stale values. A write results in a writethrough to a cache (or memory) shared by all the cores participating in the coherence protocol (the L2 cache with our assumptions) – for improved performance, these writethroughs are buffered and coalesced until the next release (or until the buffer is full). Thus, a (correctly synchronized) read miss can always obtain the up-to-date copy from the L2 cache.

Since GPU protocols do not have writer-initiated invalidations, ownership tracking, or (traditionally) scoped synchronization, they perform synchronization accesses at the shared L2 (more generally, the closest memory shared by all participating cores). For the program order requirement, preceding writes are now considered complete by a release when their writethroughs reach the shared L2 cache. Synchronization accesses are considered complete when they are performed at the shared L2 cache.

The GPU protocols are simple, do not require protocol state bits (other than valid bits), and do not incur invalidation and other protocol traffic overheads. However, synchronization operations are expensive – the operations are performed at the L2 (or the closest shared memory), an acquire invalidates the entire cache, and a release must wait until all previous writethroughs reach the shared L2. Scoped synchronizations reduce these penalties for local scopes.

In the two level hierarchy, there are two scopes – private L1 (shared by all threads on a CU) and shared L2 (shared by all cores and CUs). We refer to these as *local* and *global* scopes, respectively. A locally scoped synchronization does not have to invalidate the L1 (on an acquire), does not have to wait for writethroughs to reach the L2 (on a release), and is performed locally at the L1. Globally scoped synchronization is similar to synchronization accesses without scopes. While scopes reduce the performance penalty, they complicate the programming model, effectively exposing the memory hierarchy to the compiler and the programmer.

DeNovo: A Hybrid Hardware-Software Protocol

DeNovo is a recent hybrid hardware-software protocol that uses reader-initiated invalidations with hardware tracked ownership [46, 152, 153]. We first discuss its general functionality (DeNovo), then our extensions to it for heterogeneous systems (DeNovoA). Since there are no writer-initiated invalidations, there is no directory needed to track sharers lists. DeNovo uses the shared L2's data banks to track ownership – either the data bank has the up-to-date copy of the data (no L1 cache owns it) or it keeps the ID of the core that owns the data. DeNovo refers to the L2 as the registry and the obtaining of ownership as registration. DeNovo has three states – Registered, Valid, and Invalid – similar to the Modified, Shared, and Invalid states of the MSI protocol. All store misses need to obtain registration (analogous to MESI's ownership) from the LLC/directory. A key difference with MSI is that DeNovo has precisely these three states with no transient states, because DeNovo exploits per-word data-race-freedom and does not have writer-initiated invalidations. A consequence of exploiting data-race-freedom is that the coherence states are stored at word granularity (although tags and data communication are at a larger conventional line granularity, like sector caches).[3]

Like GPU protocols, DeNovo invalidates the cache on an acquire; however, these invalidations can be selective in several ways. Our baseline DeNovoA protocol for heterogeneous systems exploits the property that data in registered state is up-to-date and thus does not need to be invalidated (even if the data is accessed globally by multiple CUs). Previous DeNovo work has also explored additional optimizations such as software regions and touched bits. We explore a simple variant that only identifies read-only data regions and does not invalidate those on acquires. The read-only region is a hardware oblivious, program level property and is easier to determine than annotating all synchronization accesses with (hardware- and schedule-specific) scope information.

For synchronization accesses, DeNovoA uses the DeNovoSync0 protocol [152] which registers both read and write synchronizations. That is, unless the location is in registered state in the L1, it is treated as a miss for both synchronization reads and writes and requires a registration operation. This potentially provides better performance than conventional GPU protocols, which perform all

[3]This does not preclude byte granularity accesses as discussed in [46]. DeNovo allocates byte granularity regions at a word granularity where possible. If this is not possible, DeNovo "clones" the cache line (at the cost of potentially poorer cache utilization). None of our benchmarks, however, have byte granularity accesses.

synchronization at the L2 (i.e., no synchronization hits).

DeNovoSync0 serves racy synchronization registrations immediately at the registry, in the order in which they arrive. For an already registered word, the registry forwards a new registration request to the registered L1. If the request reaches the L1 before the L1's own registration acknowledgment, it is queued at the L1's MSHR. In a high contention scenario, multiple racy synchronizations from different cores will form a distributed queue. Multiple synchronization requests from the same CU (from different threads) are coalesced within the CU's MSHR and all are serviced before any queued remote request, thereby exploiting locality even under contention. As noted in previous work, the distributed queue serializes registration acknowledgments from different CUs – this throttling is beneficial when the contending synchronizations will be unsuccessful (e.g., unsuccessful lock accesses) but can add latency to the critical path if several of these synchronizations (usually readers) are successful. As discussed in [152], the latter case is uncommon.

To enforce the program order requirement, DeNovoA considers a data write and a synchronization (read or write) complete when it obtains registration. As before, data reads are complete when they return their value.

Prior to our work, DeNovo has not been evaluated with scoped synchronization, but can be extended in a natural way. Local acquires and releases do not invalidate the cache or flush the store buffer. Additionally, local synchronization operations can delay obtaining ownership.

2.4 Qualitative Analysis of the Protocols

In this section we qualitatively analyze the GPU and DeNovoA protocols, with and without scopes, as described in Section 2.3.

2.4.1 Qualitative Performance Analysis

In order to understand the advantages and disadvantages of each protocol, Table 2.2 qualitatively compares coherence protocols across several key features that are important for emerging workloads with fine-grained synchronization: exploiting reuse of data across synchronization points (in L1), avoiding bursty traffic (especially for writes), decreasing network traffic by avoiding overheads like invalidations and acknowledgment messages, only transferring useful data by decoupling the co-

Feature	Benefit	GD	GH	DD	DH
Reuse Written Data	Reuse written data across synch points	✗	✓ (if local scope)	✓	✓
Reuse Valid Data	Reuse cached valid data across synch points	✗	✓ (if local scope)	✗[4]	✓ (if local scope)
No Bursty Traffic	Avoid bursts of writes	✗	✓ (if local scope)	✓	✓
No Invalidations/ACKs	Decreased network traffic	✓	✓	✓	✓
Decoupled Granularity	Only transfer useful data	✗	✗	✓	✓
Reuse Synchronization	Efficient support for fine-grained synch	✗	✓ (if local scope)	✓	✓
Dynamic Sharing	Efficient support for work stealing	✗	✗	✓	✓

Table 2.2: Comparison of studied coherence protocols.

herence and transfer granularity, exploiting reuse of synchronization variables (in L1), and efficient support for dynamic sharing patterns such as work stealing. The coherence protocols have different advantages and disadvantages based on their support for these features:

GPU coherence, DRF consistency (*GD*): Conventional GPU protocols with DRF do not require invalidation or acknowledgment messages because they self-invalidate all valid data at all synchronization points and write through all dirty data to the shared, backing LLC. However, there are also several inefficiencies which stem from poor support for fine-grained synchronization and not obtaining ownership. Because GPU coherence protocols do not obtain ownership (and don't have writer-initiated invalidations), they must perform synchronization accesses at the LLC, they must flush all dirty data from the store buffer on releases, and they must self-invalidate the entire cache on acquires. As a result, *GD* cannot reuse any data across synchronization points (e.g., acquires, releases, and kernel boundaries). Flushing the store buffer at releases and kernel boundaries also causes bursty writethrough traffic. GPU coherence protocols also transfer data at a coarse granularity to exploit spatial locality; for emerging workloads with fine-grained synchronization or strided accesses, this can be sub-optimal.[5] Furthermore, algorithms with dynamic sharing must synchronize at the LLC to prevent stale data from being accessed.

GPU coherence, HRF consistency (*GH*): Changing the memory model from DRF to HRF removes several inefficiencies from GPU coherence protocols while retaining the benefit of no invalidation or acknowledgment messages. Although globally scoped synchronization accesses have the same behavior as *GD*, locally scoped synchronization accesses occur locally and do not require bursty writebacks, self-invalidations, or flushes, improving support for fine-grained synchronization and allowing data to be reused across synchronization points. However, scopes do not provide

[4]Mitigated by the read-only enhancement.

[5]Some examples of this are the graph analytics benchmarks, which are discussed further in Chapter 3. In these workloads, different words from the same cache line are accessed by different CUs.

efficient support for algorithms with dynamic sharing because programmers must conservatively use a global scope for these algorithms to prevent stale data from being accessed.

DeNovoA coherence, DRF consistency (*DD*): The DeNovoA coherence protocol with DRF has several advantages over *GD*. *DD*'s use of ownership enables it to provide several of the advantages of *GH* without exposing the memory hierarchy to the programmer. For example, *DD* can reuse written data across synchronization boundaries since it does not self-invalidate registered data on an acquire. With the read-only optimization, this benefit also extends to read-only data. *DD* also sees hits on synchronization variables with temporal locality both within a TB and across TBs on the same CU. Obtaining ownership also allows *DD* to avoid bursty writebacks at releases and kernel boundaries. Unlike *GH*, obtaining ownership specifically provides efficient support for applications with dynamic sharing and also transfers less data by decoupling the coherence and transfer granularity.

Although obtaining ownership usually results in a higher hit rate, it can sometimes increase miss latency; e.g., an extra hop if the requested word is in a remote L1 cache or additional serialization for some synchronization patterns with high contention (Section 2.3). The benefits, however, dominate in the results.

DeNovoA coherence, HRF consistency (*DH*): Using the HRF memory model with the DeNovoA coherence protocol combines all the advantages of ownership that *DD* enjoys with the advantages of local scopes that *GH* enjoys.

2.4.2 Protocol Implementation Overheads

Each of these protocols has several sources of implementation overhead:

GD: Since *GD* does not track ownership, the L1 and L2 caches only need 1 bit (a valid bit) per line to track the state of the cache line. GPU coherence also needs support for flash invalidating the entire cache on acquires and buffering writes until a release occurs.

GH: GPU coherence with the HRF memory model additionally requires a bit per word in the L1 caches to keep track of partial cache block writes (3% overhead compared to *GD*'s L1 cache). Like *GD*, *GH* also requires support for flash invalidating the cache for globally scoped acquires and releases and has an L2 overhead of 1 valid bit per cache line.

DD and DH: DeNovoA needs per-word state bits for the DRF and HRF memory models because DeNovoA tracks coherence at the word granularity. Since DeNovoA has 3 coherence states, at the L1 cache we need 2 bits per 32 bits or 32 bits for a 64B cache line (3% overhead over *GH*). At the L2, DeNovoA needs one valid and one dirty bit per line and one bit per word – 18 bits per 64 Bytes or (3% overhead versus *GH*).

DD with read-only optimization (DD+RO): Logically, DeNovoA needs an additional bit per word at the L1 caches to store the read-only information. However, to avoid incurring additional overhead, we reuse the extra, unused state from DeNovoA's coherence bits. There is some overhead to convey the region information from the software to the hardware. We pass this information through an opcode bit for memory instructions.

2.5 Methodology

Our work is significantly influenced by previous work on DeNovo [46, 152, 153]. We extended the project's existing infrastructure [46] to support GPU synchronization operations based on the DeNovoSync0 coherence protocol for multi-core CPUs [152]. We discuss the extensions further in Section 2.5.3.

2.5.1 Baseline Heterogeneous Architecture

Figure 2.1 shows the baseline heterogeneous architecture, a tightly integrated CPU-GPU system with a unified shared memory address space and coherent caches. The system connects all CPU cores and GPU Compute Units (CUs) via an interconnection network. Each GPU Compute Unit (CU), has a separate node on the network. All CPU and GPU cores have an attached block of SRAM. For CPU cores, this is an L1 cache, while for GPU cores, it is divided into an L1 cache and a scratchpad. Each node also has a bank of the L2 cache, which is shared by all CPU and GPU cores. The coherence protocol, consistency model, and write policy depend on the system configuration studied (Section 2.5.4).

HRF [77] uses a three-level cache hierarchy in its evaluation; we use two levels because the GEMS simulation environment only supports two levels. We believe these results are not qualitatively affected by the depth of the memory hierarchy. To understand this claim, it helps to first consider

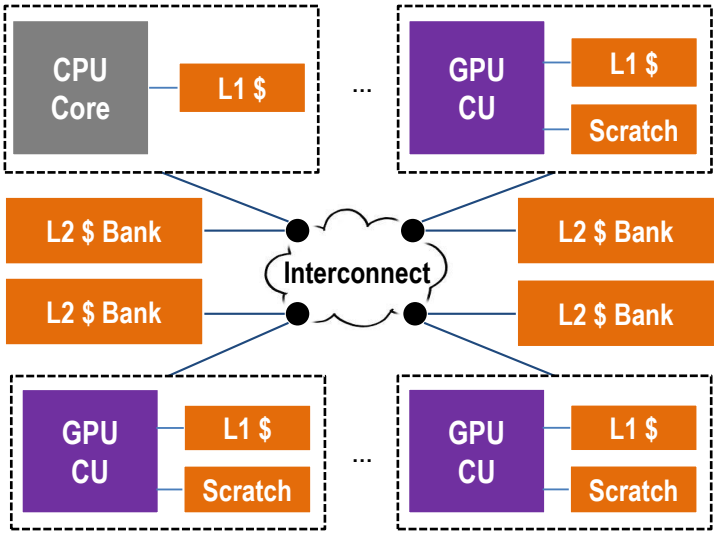


Figure 2.1: Baseline heterogeneous architecture.

how DeNovoA would behave with a 3-level cache hierarchy. L1 would be private to a core, L2 would be shared by a core cluster, and L3 would be global. The registry would be similarly hierarchical: L3 would keep track of cluster ownership and L2 would keep track of core ownership. In this configuration, synchronization at the L1 scope would still be efficient with DeNovoA because a write by a producer thread would place data in the owned (registered) state for a consumer thread.

In the case of synchronization at the L2 scope, even a scopeless DeNovoA coherence protocol is no less effective than GPU coherence with HRF. When synchronizing at L2 using GPU coherence and HRF, after the producer thread writes some data but before the consumer reads it the producer thread must write back any pending stores to L2 and the consumer thread must invalidate its L1 cache. Data sharing between threads can thus complete without sending any messages beyond L2. With a hierarchical registry, DeNovoA can achieve this as well. In the case of a L2 synchronization, the producer core would obtain ownership of its written data on a release, and the consumer core would invalidate its cache on an acquire. However, it would not need to invalidate the data in the L2 cache because it would find that the data is owned in its cluster. Thus, when reading the data, it would request the data from L2, which would instruct the producer's cache to forward the data directly to the consuming core. Therefore the DeNovoA protocol with DRF consistency would still perform as well as GPU coherence with HRF in a deeper cache hierarchy. In fact, the core-to-core transfer of data that occurs in DeNovoA reduces network traffic, an effect that increases with scope

depth. The fact that DeNovoA can achieve efficiency in this type of hierarchical synchronization without scope information from the software is even more important as cache hierarchies grow deeper and the relative locations of various threads becomes harder to predict.[6]

2.5.2 Simulation Environment and Parameters

We simulate our architecture using an integrated CPU-GPU simulator built from the Simics full-system functional simulator to model the CPUs, the Wisconsin GEMS memory timing simulator [110], and GPGPU-Sim v3.2.1 [24] to model the GPU (the GPU is similar to an NVIDIA GTX 480). The simulator also uses Garnet [10] to model a 4x4 mesh interconnect with a GPU CU or a CPU core at each node. We use CUDA 3.1 [118] for the GPU kernels in the applications since this is the latest version of CUDA that is fully supported in GPGPU-Sim. Table 2.3 summarizes the common key parameters of the system. For the scaling study, we vary the number of CUs between 1 and 15; in all other experiments we use 15 CUs.

For energy modeling, GPU CUs use GPUWattch [97] and the NoC energy measurements use McPAT v.1.1 [100] (our tightly coupled architecture more closely resembles a multi-core system's NoC than the NoC modeled in GPUWattch). We do not model the CPU core or CPU L1 energy since the CPU is only functionally simulated and not the focus of this work. However, we do measure the network traffic traveling to and from the CPU in order to capture any network traffic variations caused by adding fine-grained support.

Finally, we also provide an API for manually inserted annotations for region information (for *DD+RO*) and distinguishing synchronization instructions and their scope (for HRF).

2.5.3 Protocol Extensions and Assumptions

DeNovo was originally designed for multi-core CPUs. With DeNovoA, we extended DeNovo to heterogeneous systems. We added two major extensions to the DeNovo protocol to support GPU memory requests. First, we added support for multi-word requests. Second, we modified the GPU core to coalesce requests at a 64-Byte granularity instead of a 128-Byte granularity to have uniform

[6]The overhead for a three-level cache hierarchy will be different. Section 2.4.2 assume a two-level cache hierarchy.

CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
CUs	15
Memory Hierarchy Parameters	
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
Store Buffer Size	256 entries
L1 MSHRs	128 entries
L1 hit latency	1 cycle
Remote L1 hit latency	35−83 cycles
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

Table 2.3: Simulated heterogeneous system parameters.

cache line sizes for all cores.[7] We also assume that the GPU cores in the system will have TLB support, e.g., using the scheme of Pichai et al. [122] or Power & Hill [126].

Global Memory Requests

Supporting coalesced memory requests required adding support for multi-word requests, since DeNovo only performed single word requests. To address this, we modified GPU requests to send information about what specific words in a cache line they are requesting. GPU loads are also non-blocking. Without the proper support, this could lead to half-warps from multiple warps on the same CU issuing requests for the same cache line. To resolve this issue, we use the MSHRs to check if a request for this cache line is already in progress. If it is, then we simply append this half-warp's memory request to this entry in the MSHR and replay that request when the response from L2 is received.

GPUs coalesce global memory requests at the granularity of their L1 cache line size (typically 128-Byte cache lines). We modified the coalescing rules to coalesce requests at a 64-Byte granularity because the CPU cache lines are only 64-Bytes and we don't support requests that span multiple cache lines (which the 128-Byte requests become with 64-Byte cache lines). We chose to modify the GPU coalescing scheme to use a granularity of 64-Bytes instead of making all L1 caches in the system use 128-Byte cache lines; 128-Byte cache lines can harm performance for CPU applications

[7] Uniform cache line sizes are not mandatory for any of the coherence protocols we study, although they do simplify the implementation.

and we wanted to have uniform cache line sizes for all CPU cores and GPU cores.

Benchmark Conversions

The compilation process involved three steps. In the first step, we used NVCC to generate the PTX code for GPU kernels and an intermediate C++ code with CUDA specific annotations and functions. In the second step we edit these function calls so that they can be intercepted by Simics and can be passed on to GPGPU-Sim during the simulation. This step is automated in the implementation. Finally, we compile the edited C++ files using g++ (version 4.5.2) to generate the binary. We did not introduce any additional compile-time overheads compared to a typical compilation of a CUDA program. Even when a CUDA application is compiled normally, there are two steps involved - NVCC emitting the PTX code and an annotated C++ program and g++ converting this C++ code into binary (these steps are hidden from the user).

2.5.4 Configurations

We evaluate the following configurations with GPU and DeNovoA coherence protocols combined with DRF and HRF consistency models, using the implementations described in Section 2.3. The CPU always uses the DeNovoA coherence protocol.[8] For all configurations we assume 256 entry coalescing store buffers next to the L1 caches. We also assume support for performing synchronization accesses (using atomics) at the L1 and L2.

GD[9]: *GD* combines the baseline DRF memory model (no scopes) with GPU coherence and performs all synchronization accesses at the L2 cache.

GH: *GH* uses GPU coherence and HRF's HRF-Indirect memory model. GH performs locally scoped synchronization accesses at the L1s and globally scoped synchronization accesses at the L2.

DD: *DD* uses the DeNovoSync0 coherence protocol (without regions), a DRF memory model, and performs all synchronization accesses at the L1 (after registration).

DD with read-only optimization (DD+RO): *DD+RO* augments *DD* with selective invalidations to avoid invalidating valid read-only data on acquires.

[8] For the GPU coherence configurations, when dirty data is written through, if a CPU core owns the data from a previous phase, the L2 revokes the CPU cores ownership before completing the request.

[9] This is the implementation described in [63, 77] but limited to a global synchronization scope.

Benchmark	Input
No Synchronization	
Backprop (BP)[43]	32 KB
Pathfinder (PF)[43]	10 x 100K matrix
LUD[43]	256x256 matrix
NW[43]	512x512 matrix
SGEMM[145]	medium
Stencil (ST)[145]	128x128x4, 4 iters
Hotspot (HS)[43]	512x512 matrix
NN[43]	171K records
SRAD v2 (SRAD)[43]	256x256 matrix
LavaMD (LAVA)[43]	2x2x2 matrix
Global Synchronization	
FA Mutex (FAM_G), Sleep Mutex (SLM_G), Spin Mutex (SPM_G), Spin Mutex with backoff (SPMBO_G), Spin Semaphore (SS_G), Spin Semaphore with backoff (SSBO_G),	3 TBs/CU, 100 iters/TB/kernel, 10 Ld&St/thr/iter
Local or Hybrid Synchronization	
FA Mutex (FAM_L), Sleep Mutex (SLM_L), Spin Mutex (SPM_L), Spin Mutex with backoff (SPMBO_L), Tree Barrier with local exchange (TRBEX_LG), Tree Barrier (TRB_LG), Lock-Free Tree Barrier with local exchange (LFTRBEX_LG), Lock-Free Tree Barrier (LFTRB_LG),	3 TBs/CU, 100 iters/TB/kernel, 10 Ld&St/thr/iter
Spin Semaphore (SS_L), Spin Semaphore with backoff (SSBO_L)[146]	3 TBs/CU, 100 iters/TB/kernel, readers: 10 Ld/thr/iter writers: 20 St/thr/iter
UTS[77]	16K nodes

Table 2.4: Benchmarks with input sizes. All thread blocks (TBs) in the synchronization microbenchmarks execute the critical section or barrier many times. Microbenchmarks with local and global scope are denoted with a '_L' and '_G', respectively.

DH: *DH* combines DD with the HRF-Indirect memory model. Like *GH*, local scope synchronizations always occur at the L1 and do not require invalidations or flushes.

2.5.5 Benchmarks

Evaluating the configurations is challenging because previously there are very few GPU application benchmarks that use fine-grained synchronization. Thus, we use a combination of application benchmarks and microbenchmarks to cover the space of use cases with (1) no synchronization within a GPU kernel, (2) synchronization that requires global scope, and (3) synchronization with mostly local scope. We ported all of the applications to use a unified address space with coherent

caches. All codes execute GPU kernels on 15 GPU CUs and use a single CPU core.

Applications without Intra-Kernel Synchronization

We examine 10 applications from modern heterogeneous computing suites such as Rodinia [43, 44] and Parboil [145]. None of these applications use synchronization within the GPU kernel and are also not written to exploit reuse across kernels. These applications therefore primarily serve to establish DeNovoA as a viable protocol for today's use cases. The top part of Table 2.4 summarizes these applications and their input sizes.

(Micro)Benchmarks with Intra-Kernel Synchronization

Most GPU applications do not use fine-grained synchronization because it is not well supported on current GPUs. Thus, to examine the performance for benchmarks with various kinds of synchronization we use a set of synchronization primitive (SyncPrims) microbenchmarks, developed by Stuart and Owens [146] – these include mutex locks, semaphores, and barriers. We also use the Unbalanced Tree Search (UTS) benchmark [77], the only benchmark that uses fine-grained synchronization in the HRF paper.[10] The microbenchmarks include centralized and decentralized algorithms with a wide range of stall cycles and scalability characteristics. The amount of work per thread also varies: the mutex and tree barrier algorithms access the same amount of data per thread while UTS and the semaphores access different amounts of data per thread. The bottom part of Table 2.4 summarizes the benchmarks and their input sizes.

We discuss the SyncPrims microbenchmarks in detail in Section 2.6. The working set fits in the L1 cache for all microbenchmarks except TRBEX_LG and TRB_LG, which have larger working sets because they repeatedly exchange data across CUs. The UTS benchmark utilizes both local and global synchronization. By performing local synchronization accesses, UTS quickly completes its work. However, since the tree is unbalanced, it is likely that some TBs will complete before others. To mitigate load imbalance, CU's push to and pull from a global task queue when their local queues become full or empty, respectively.

[10] RemoteScopes [121] uses several GPU benchmarks with fine-grained synchronization from Pannotia [42] but these benchmarks were not publicly available when this work was originally done. The results in Section 3.6 show that the same trends we observe for the SyncPrims microbenchmarks and UTS hold for the full-sized Pannotia benchmarks.

Microbenchmark	Description
Mutexes	
Spin Mutex (SPM)	Test-and-set lock
Spin Mutex with backoff (SPMBO)	Test-and-set lock with backoff
FA Mutex (FAM)	Centralized ticket lock
Sleep Mutex (SLM)	Decentralized ticket lock
Semaphores	
Spin Semaphore (SS)	Semaphore with mutex lock
Spin Semaphore with backoff (SSBO)	Semaphore with mutex lock and backoff
Barriers	
Tree Barrier (TRB)	Two-level atomic tree barrier
Tree Barrier with local exchange (TRBEX)	Two-level atomic barrier with local exchange
Lock-Free Tree Barrier (LFTRB)	Two-level lock-free tree barrier
Lock-Free Tree Barrier with local exchange (LFTRBEX)	Two-level lock-free tree barrier with local exchange

Table 2.5: Synchronization primitive microbenchmarks.

2.6 Synchronization Primitive Microbenchmarks

The SyncPrims microbenchmarks, which make up one part of the HeteroSync benchmark suite, were originally designed by Stuart and Owens to study the performance of synchronization primitives on discrete GPUs [146]. These microbenchmarks, listed in Table 2.5, include mutexes (Section 2.6.1), semaphores (Section 2.6.2), and barriers (Section 2.6.3) with a wide range of stall cycles and scalability characteristics. Stuart & Owens' focus was on the performance of the atomic operations used to implement the synchronization primitives, not the overheads associated with properly synchronizing the global data accessed in a critical section. Consequently, the original SyncPrims algorithms do not have any global data accesses. To fix this, we updated the SyncPrims microbenchmarks to use global data accesses (instead of scratchpad accesses) in the critical sections.

We also created two versions of the SyncPrims microbenchmarks to share data at different levels of the memory hierarchy to enable synchronization with HRF's local and global scopes. The mutex microbenchmarks have two versions: one performs local synchronization and accesses unique data per CU while the other uses global synchronization because the same data is accessed by all TBs. We also changed the globally synchronized barrier microbenchmarks to use local and global synchronization with tree barriers: all TBs on a CU access unique data and join a local barrier before one TB from each CU joins the global barrier. After the global barrier, TBs exchange data for the subsequent iteration of the compute phase. We also added a version of the tree barrier where each CU exchanges data locally before joining the global barrier. Additionally, we changed the semaphores to use a reader-writer format with local synchronization: each CU has one writer

TB and two reader TBs. Each reader reads half of the CU's data. The writer shifts the reader TB's data to the right such that all elements are written except for the first element of each TB. To ensure that no stale data is accessed, the writers obtain the entire semaphore.

The locally scoped version only shares data locally between TBs on a CU, so they can improve efficiency and reduce contention by performing the synchronization accesses locally (with a per-CU mutex or semaphore), reusing data locally, and avoiding expensive cache invalidations and store buffer flushes. The globally scoped version shares data across multiple CUs, so it must use global synchronization and cannot benefit from HRF.

Finally, we updated the SyncPrims microbenchmarks to provide better ordering support. GPUs employ a relaxed memory consistency model and provided some fence primitives to help enforce ordering. However, GPU memory consistency model behavior has been slow to be clearly defined [13, 104, 143, 144]. As a result, synchronization and consistency on modern GPUs is neither well-defined nor intuitive. Unsurprisingly, some of this work identified situations where the fences in SyncPrims did not properly enforce ordering, partly due to the opaqueness and complexity of the CUDA memory consistency model [13, 144]. To resolve this issue, we use synchronization loads and stores to enforce ordering instead of using inefficient (and sometimes incorrect) fences. On a synchronization load, all potentially stale data is invalidated and on a synchronization store all dirty data must be flushed from the store buffer.

The basic structure of the Spin Mutex microbenchmark is shown in Algorithm 1; the other SyncPrims microbenchmarks use a similar structure. The barrier microbenchmarks are also similar except the TBs join a barrier at the end of the loop. We include pseudo-code for all synchronization primitives. For the microbenchmarks where we have significantly altered the algorithms, we include the original algorithm and our new algorithm. Since the locally and globally scoped algorithms are very similar to each other, we only show pseudo-code for the globally scoped algorithms. Without loss of generality we describe the globally scoped version of each algorithm (since the locally scoped versions are identical except for their use of per-CU mutexes or semaphores).

Algorithm 1 Basic structure of SyncPrims microbenchmarks. The other SyncPrims microbenchmarks use a similar structure.

```
for i = 0: numCSIters do
    SpinMutexLock(Mutex)
    // Perform global data accesses
    SpinMutexUnlock(Mutex)
end for
```

Algorithm 2 Spin Mutex Algorithm.

```
function SpinMutexLock(Mutex, DoBackoff)
    Acquired ← false
    while Acquired = false do
        OldVal ← atomicCAS(Mutex, 0, 1) ⟨⟨Acquire⟩⟩
        if OldVal = 0 then
            Acquired ← true
        else if Acquired = false && DoBackoff = true then
            Backoff()
        end if
    end while
end function

function SpinMutexUnlock(Mutex) ⟨⟨Release⟩⟩
    atomicExch(Mutex, 0)
end function
```

2.6.1 Mutexes

Spin Mutex (**SPM**): Algorithm 2 contains the pseudo-code for the Spin Mutex algorithm from [146]. In this algorithm, a straightforward port of spin mutex algorithms on CPUs, each TB repeatedly tries to obtain a single mutex lock. Once the TB is done with the critical section, it releases the lock. However, SPM may suffer from high contention and is neither deterministic nor fair.

Spin Mutex with Backoff (**SPMBO**): One way to optimize the performance of the SPM algorithm is to add backoff. By requiring a TB to wait a while after it fails to acquire the lock, Spin Mutex with Backoff reduces contention by performing exponential backoff after a failed attempt to acquire the lock.

Fetch-and-Add Mutex (**FAM**): Algorithm 3 contains the pseudo-code for the Fetch-and-Add (FA) Mutex, which operates like a centralized ticket lock. Unlike the Spin Mutex algorithms, the FA Mutex is fair and has a deterministic number of atomic accesses. Instead of directly acquiring a lock, each TB obtains a "ticket" that represents when the TB will be able to acquire the lock.

Algorithm 3 Fetch-and-Add (FA) Mutex.

```
function FAMutexLock(Mutex)
    TicketNumber ← atomicInc(Mutex.ticket)
    while TicketNumber ≠ atomicLd(Mutex.turn) do ⟨⟨Acquire⟩⟩
        ;
    end while
end function

function FAMutexUnlock(Mutex)
    atomicInc(Mutex.turn) ⟨⟨Release⟩⟩
end function
```

Each TB then polls the current ticket number. When the ticket number equals this TBs ticket number, then the TB has acquired the lock. To release the lock, the TB simply increments the current ticket number. Although this algorithm is fair, because it uses two centralized counters (one for the head, one for the tickets) it can suffer from heavy contention.

Sleep Mutex (**SLM**): Algorithm 4 shows the original Sleep Mutex algorithm. In the original Sleep Mutex algorithm, each TB places itself on the end of a ring buffer [146]. Afterwards, the TB repeatedly checks if it is at the front of the buffer by checking the shared head pointer. To release the lock, a TB increments the head pointer. However, this algorithm scales poorly because it requires more reads than FAM.

Algorithm 5 shows the new Sleep Mutex algorithm. In the original paper, the authors did not consider a decentralized algorithm because of poor GPU support for linked data structures. However, we found that we could overcome this shortcoming by making Sleep Mutex a *decentralized ticket lock*. After getting a ticket (similar to FAM), each TB spins on a unique location in the ring buffer. Reducing contention for a given location in the ring buffer improves performance by allowing each TB to spin locally[11] on its location. To transfer ownership of the lock to the next TB, instead of updating the head pointer, a TB instead updates the value in the next location in the ring buffer. Upon seeing this update, the TB spinning on that location now owns the lock and can proceed.

[11]The TB can only spin locally in the L1 cache if the scope is local and GPU coherence with HRF consistency is used, or if DeNovoA coherence is used. If GPU coherence is used with DRF consistency, then the TB cannot spin locally and must perform its atomic accesses at the LLC (L2).

Algorithm 4 Original Sleep Mutex Algorithm [146].

```
function SleepMutexLock(Mutex, BlockID)
    RingBufferLoc ← atomicInc(Mutex.tailPtr)
    atomicExch(buffer[RingBufferLoc], BlockID)
    while atomicLd(buffer[Mutex.head]) ≠ BlockID do
        ;
    end while
end function

function SleepMutexUnlock(Mutex)
    atomicInc(Mutex.headPtr)
end function
```

Algorithm 5 New, Decentralized Sleep Mutex Algorithm. Each thread spins on its own location in the ring buffer instead of all threads accessing the head pointer.

```
function SleepMutexLock(Mutex, BlockID)
    RingBufferLoc ← atomicInc(Mutex.tailPtr)
    while atomicLd(buffer[RingBufferLoc]) = 0 do ⟨⟨Acquire⟩⟩
        ;
    end while
    atomicDec(buffer[RingBufferLoc])
end function

function SleepMutexUnlock(Mutex, NextBufferLoc)
    atomicExch(buffer[NextBufferLoc], 1) ⟨⟨Release⟩⟩
end function
```

2.6.2 Semaphores

Originally, the semaphore algorithms were reader-only [146]. To make the critical section more practical, we updated the semaphore algorithms to be reader-writer: each CU has one writer TB and $N-1$ reader TBs. Each reader reads a subset of the data and the writer writes all of the data.

Spin Semaphore (**SS**): The original Spin Semaphore Algorithm is shown in Algorithm 6. To enter the critical section, a thread first acquires a mutex lock, then checks to see if there is room in the semaphore for it; if there is it updates the semaphore [146]. After this check, the TB releases the lock. When a TB is leaving the critical section, it uses a similar process: acquire lock, update semaphore to remove self, release lock. By using a semaphore, multiple TBs can enter the critical section simultaneously (as long as the max semaphore size is > 1). This scheme requires that the data being accessed by each of these TBs is either independent or read-only. Moreover, if the centralized semaphore and mutex lock are heavily contended, performance suffers.

Algorithm 6 Original Spin Semaphore Algorithm [146]. This algorithm assumes that all TBs are reading the data in the critical section.

```
function SpinSemaphoreWait(Sem)
    Acquired ← false
    while Acquired = false do
        OldValue ← atomicCAS(Sem.Lock, 0, 1)
        if OldValue ≠ 0 then ⟨⟨Acquire⟩⟩
            if atomicLd(Sem) > 1 then
                atomicSub(Sem, 1)
                Acquired ← true
            end if
        end if
        atomicExch(Sem.Lock, 1) ⟨⟨Release⟩⟩

        if Acquired = false && DoBackoff = true then
            Backoff()
        end if
    end while
end function

function SpinSemaphorePost(Sem)
    Acquired ← false
    while Acquired = false do
        OldValue ← atomicCAS(Sem.Lock, 0, 1)
        if OldValue ≠ 0 then ⟨⟨Acquire⟩⟩
            Acquired ← true
        end if
    end while

    atomicAdd(Sem, 1)
    atomicExch(Sem.Lock, 1) ⟨⟨Release⟩⟩
end function
```

Algorithm 7 shows the updated reader-writer Spin Semaphore. The new algorithm does not require the data to be independent or read-only. Instead, the writers obtain the entire semaphore to prevent any readers from reading stale data. This introduces the potential for starvation of the writers: as soon as one reader enters the critical section, a writer must wait while other readers may be able to continue entering the critical section. The algorithm adds a flag to prevent any more readers from entering the critical section when a writer is waiting.

Spin Semaphore with Backoff (**SSBO**): Similar to SPMBO, the amount of contention in SS (for the mutex lock and semaphore) can be reduced by introducing exponential backoff.

Algorithm 7 New Spin Semaphore Algorithm. This algorithm was redesigned to use a reader-writer semaphore where some TBs read the data in the critical section and other TBs write the data.

```
function SpinSemaphoreWait(Sem, IsWriter)
    Acquired ← false
    while Acquired = false do
        OldValue ← atomicCAS(Sem.Lock, 0, 1) ⟨⟨Acquire⟩⟩
        if OldValue ≠ 0 then
            if IsWriter then
                if Sem = Sem.MaxValue then
                    Sem = 0
                    Acquired ← true
                end if
            else
                if Sem > 1 then
                    Sem = Sem - 1
                    Acquired ← true
                end if
            end if
        end if
        atomicExch(Sem.Lock, 1) ⟨⟨Release⟩⟩

        if Acquired = false && DoBackoff = true then
            Backoff()
        end if
    end while
end function

function SpinSemaphorePost(Sem, IsWriter)
    Acquired ← false
    while Acquired = false do
        OldValue ← atomicCAS(Sem.Lock, 0, 1) ⟨⟨Acquire⟩⟩
        if OldValue ≠ 0 then
            Acquired ← true
        end if
    end while

    if IsWriter then
        Sem = Sem + Sem.MaxValue
    else
        Sem = Sem + 1
    end if
    atomicExch(Sem.Lock, 1) ⟨⟨Release⟩⟩
end function
```

Algorithm 8 Original Barrier Algorithm [146].

```
function BARRIER(Count, NumTBs)
    atomicInc(Count)
    while atomicCAS(Count, NumTBs, 0) ≠ 0 do
        ;
    end while
end function
```

Algorithm 9 New Tree Barrier Algorithm. This algorithm reduces contention by only needing one thread per CU to join the global barrier, while the remaining threads on a given CU spin locally on the local barrier.

```
function TREEBARRIERGLOBAL(Count, NumCUs)
    atomicInc(Count) ⟨⟨Release⟩⟩
    while atomicCAS(Count, NumCUs, 0) ≠ 0 do ⟨⟨Acquire⟩⟩
        ;
    end while
end function

function TREEBARRIERLOCAL(CountLocal, NumLocal)
    atomicInc(CountLocal) ⟨⟨Release⟩⟩
    while atomicCAS(CountLocal, NumLocal, 0) ≠ 0 do ⟨⟨Acquire⟩⟩
        ;
    end while
end function
```

2.6.3 Barriers

The original barrier microbenchmarks used globally scoped synchronization [146]. As in previous sections, we optimized the barrier algorithms. First, we changed the single-level barriers into tree barriers where all TBs on a CU access unique data and join a local barrier before one TB from each CU joins the global barrier. After the global barrier, TBs exchange data across CUs for the subsequent iteration.

Tree Barrier (**TRB**): The original Barrier algorithm, in Algorithm 8, uses a single global counter. Upon reaching the barrier, each TB atomically increments the counter, then spins waiting for the other TBs to join the barrier. To ensure that a TB does not miss the end of the barrier, the algorithm uses a sense-reversing barrier. As the number of TBs increases, this algorithm scales poorly, because all TBs are contending for the same counter variable. Using a tree barrier allows all TBs on a given CU to access a separate counter for the local barrier, and reduces contention for the global barrier because fewer TBs need to join it.

Algorithm 10 Original Lock-Free Barrier Algorithm [146, 160].

```
function LFBarrier(InArr, OutArr, BlockID, NumTBs)
    atomicExch(InArr[BlockID], 1) ⟨⟨Release⟩⟩

    Done ← true
    if BlockID = 0 then
        repeat
            for i = 0:NumTBs do
                if atomicLd(InArr[i]) = 0 then ⟨⟨Acquire⟩⟩
                    Done ← false
                    break
                end if
            end for
            if Done = false then
                Backoff()
            end if
        until Done = true

        for i = 0:NumTBs do
            InArr[i] = 0
            atomicExch(OutArr[i], 1) ⟨⟨Release⟩⟩
        end for
    end if

    while atomicLd(OutArr[BlockID]) != 1 do ⟨⟨Acquire⟩⟩
        Backoff()
    end while
    atomicExch(OutArr[BlockID], 0)
end function
```

Lock-Free Tree Barrier (**LFTRB**): The Lock-Free Barrier, shown in Algorithm 10, decentralizes the single-level barrier to reduce contention and improve both efficiency and scalability [146]. Each TB joins the barrier by setting a unique location in an array, then waits (using exponential backoff) for all other TBs to join the barrier. As Algorithms 12 and 11 show, we converted the lock-free barrier into a two-level lock-free tree barrier, similar to the atomic tree barrier, and ported it to the tightly coupled system. This algorithm combines the contention-reducing benefits of Algorithms 10 (threads spin on separate variables for the local half of the barrier) and 9 (fewer threads join the global half of the barrier).

Algorithm 11 Local half of new Lock-Free Tree Barrier Algorithm. All threads on a given CU will join the same local barrier. Once all of the threads have joined the local barrier, one thread joins the global barrier while the remaining threads each spin a separate variable.

```
function LFTreeBarrierLocal(InArrLocal, OutArrLocal, blockID, NumTBs)
    atomicExch(InArrLocal[blockID], 1) ⟨⟨Release⟩⟩

    Done ← true
    if blockID = 0 then
        repeat
            for i = 0:NumTBs do
                if atomicLd(InArrLocal[i]) = 0 then ⟨⟨Acquire⟩⟩
                    Done ← false
                    break
                end if
            end for
            if Done = false then
                Backoff()
            end if
        until Done = true

        for i = 0:NumTBs do
            InArrLocal[i] = 0
            atomicExch(OutArrLocal[i], 1) ⟨⟨Release⟩⟩
        end for
    end if

    while atomicLd(OutArrLocal[blockID]) != 1 do ⟨⟨Acquire⟩⟩
        Backoff()
    end while
    atomicExch(OutArrLocal[blockID], 0)
end function
```

2.7 Results

Figures 2.2, 2.3 and 2.4, show the results for the applications without fine-grained synchronization, for microbenchmarks with locally scoped or hybrid synchronization, and for codes with globally scoped fine-grained synchronization, respectively. Our tightly coupled simulator is deterministic, so we do not show error bars. Moreover, for all applications, our simulator models the effect of shared L2 and memory contention. Parts (a)-(c) in each figure show execution time, energy consumed, and network traffic, respectively. Energy is divided into multiple components based on the source

Algorithm 12 Global half of new Lock-Free Tree Barrier Algorithm. Only one thread from each CU needs to join the global barrier.

```
function LFTreeBarrierGlobal(InArrGlobal, OutArrGlobal, cuID, NumCUs)
    atomicExch(InArrGlobal[cuID], 1) ⟨⟨Release⟩⟩

    Done ← true
    if cuID = 0 then
        repeat
            for i = 0:NumCUs do
                if atomicLd(InArrGlobal[i]) = 0 then ⟨⟨Acquire⟩⟩
                    Done ← false
                    break
                end if
            end for
            if Done = false then
                Backoff()
            end if
        until Done = true

        for i = 0:NumCUs do
            InArrGlobal[i] = 0
            atomicExch(OutArrGlobal[i], 1) ⟨⟨Release⟩⟩
        end for
    end if

    while atomicLd(OutArrGlobal[cuID]) != 1 do ⟨⟨Acquire⟩⟩
        Backoff()
    end while
    atomicExch(OutArrGlobal[cuID], 0)
end function
```

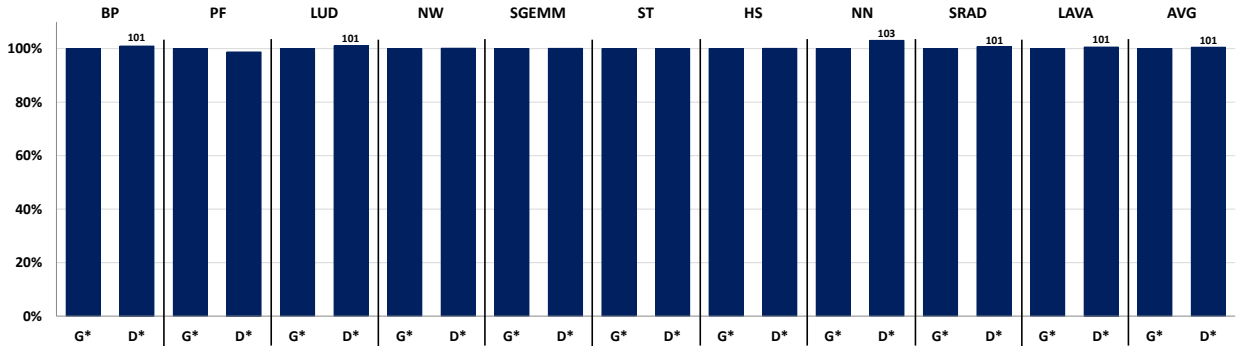
of energy: GPU core+,[12] scratchpad, L1, L2, and network. Network traffic is measured in flit crossings and is also divided into multiple components: data reads, data registrations (writes), writebacks/writethroughs, and atomics.

In Figures 2.2 and 2.4, we only show the *GD* and *DD* configurations because HRF does not affect these cases (there is no local synchronization) – we denote the systems as *G** to indicate that *GD* and *GH* obtain the same results and *D** to indicate that *DD* and *DH* obtain the same results.[13] For Figure 2.3, we show all five configurations.

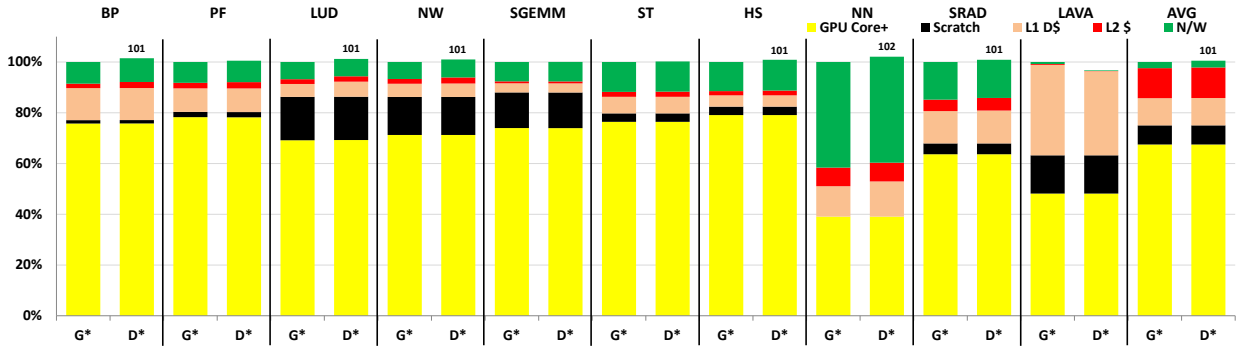
Overall, compared to the best GPU coherence protocol (*GH*), we find that *DD* is comparable

[12] GPU core+ includes the instruction cache, constant cache, register file, SFU, FPU, scheduler, and the core pipeline.

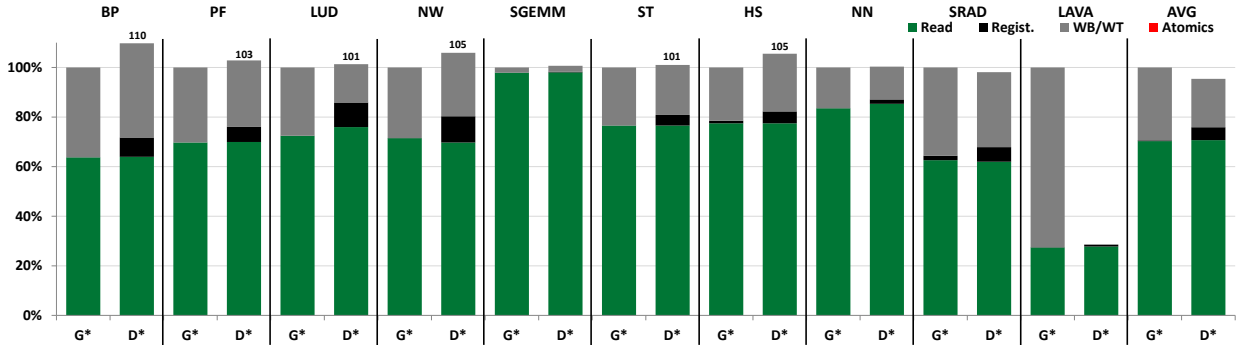
[13] *DD+RO* and *DD* provide similar performance, so we do not show *DD+RO* here.



(a) Execution time



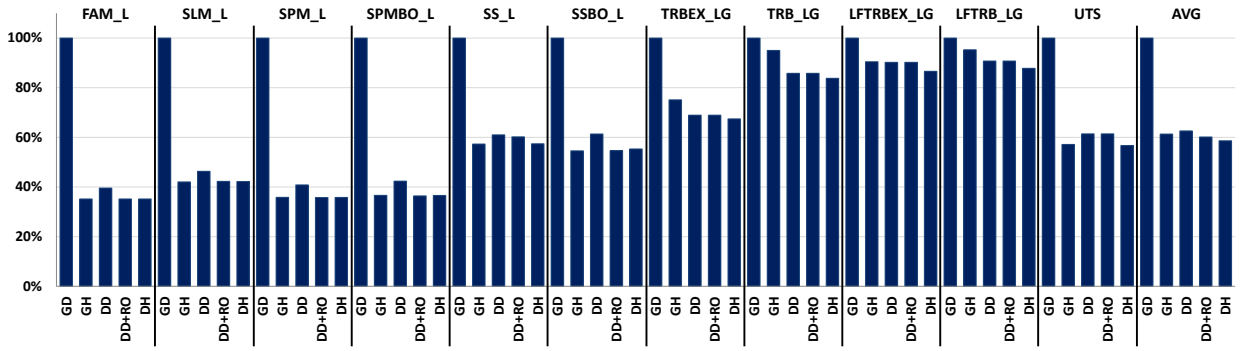
(b) Dynamic energy



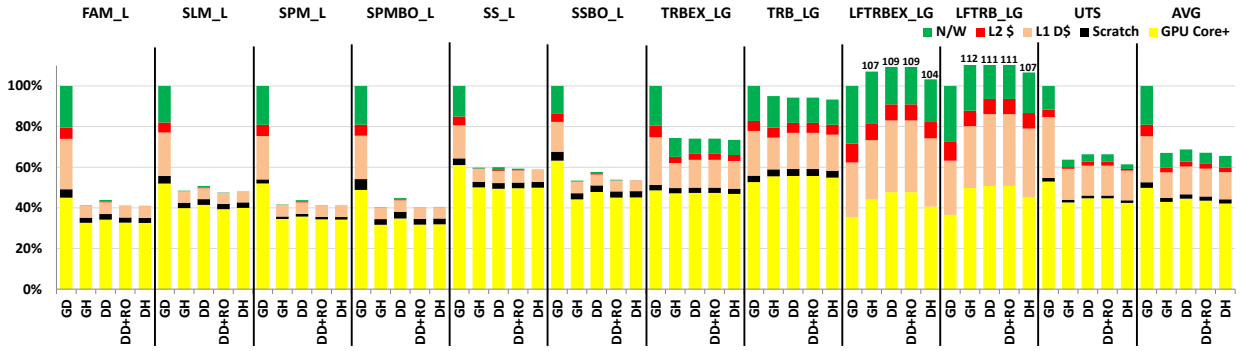
(c) Network traffic

Figure 2.2: *G** and *D**, normalized to *D**, for benchmarks without synchronization.

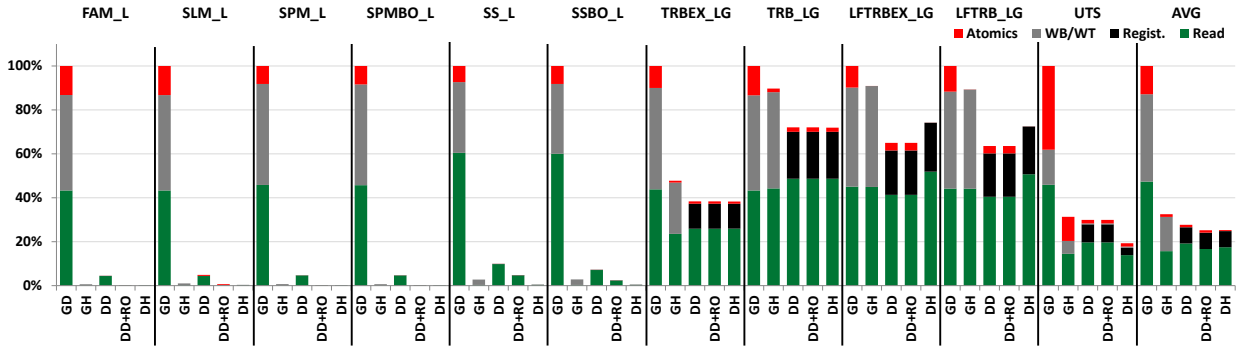
for applications with no fine-grained synchronization and better for microbenchmarks that employ synchronization with only global scopes (average of 21% lower execution time, 45% lower energy). For microbenchmarks with mostly locally scoped synchronization, *GH* is better (on average 5% lower execution time and 3% lower energy) than *DD*. This modest benefit of *GH* comes at the cost of a more complex memory model – adding a read-only region enhancement with *DD* removes most of this benefit and using HRF with *DeNovoA* makes it the best performing protocol.



(a) Execution time



(b) Dynamic energy



(c) Network traffic

Figure 2.3: All configurations with synchronization benchmarks that use mostly local synchronization, normalized to *GD*.

2.7.1 GD vs. GH

Figure 2.3 shows that when locally scoped synchronization can be used, *GH* can significantly improve performance over *GD*, as noted in prior work [77]. On average *GH* decreases execution time by 39% and energy by 33% for benchmarks that use local synchronization. There are two main sources of improvement. First, the latency of locally scoped acquires is much smaller because they are performed at L1 (which reduces atomic traffic by an average of 95%). Second, local acquires

do not invalidate the cache and local releases do not flush the store buffer. As a result, data can be reused across local synchronization boundaries. Since accesses hit more frequently in the L1 cache, execution time, energy, and network traffic improve. On average, the L1, L2, and network energy components decrease by 58% for *GH* while data (non-atomic) network traffic decreases by an average of 64%.

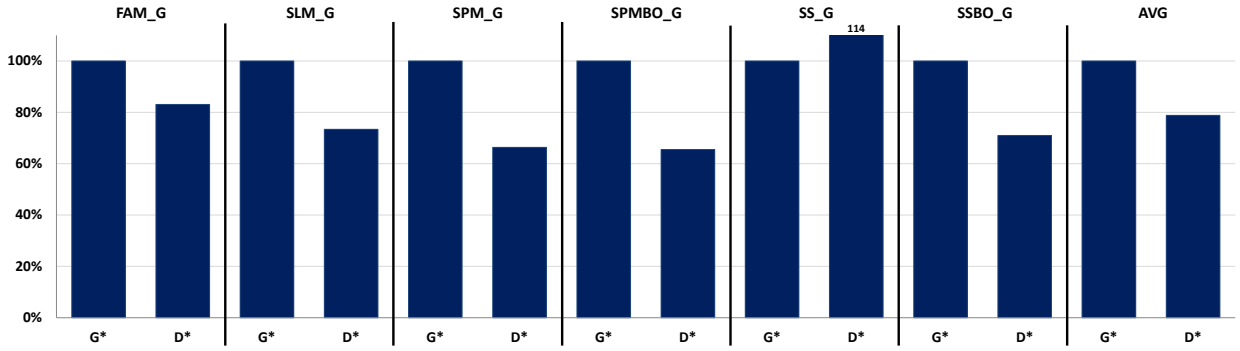
2.7.2 DD vs. GPU Coherence

Traditional GPU Applications

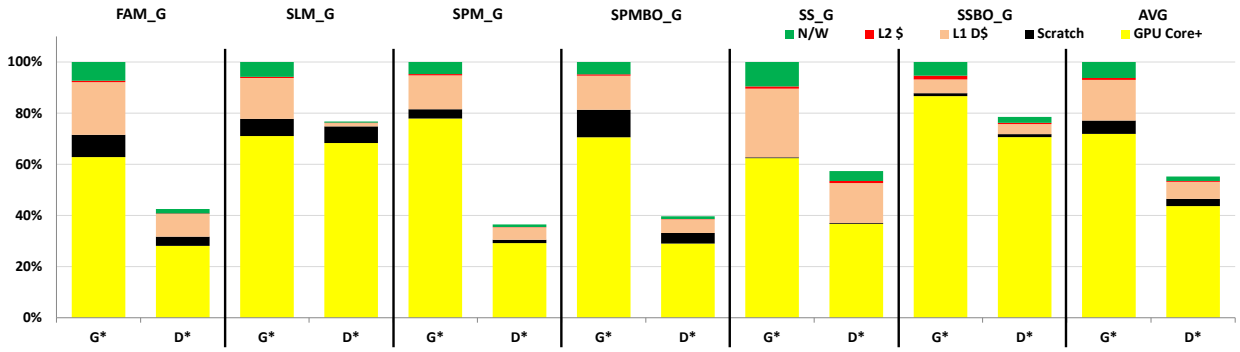
For the ten applications studied that do not use fine-grained synchronization, Figure 2.2 shows there is generally little difference between *DeNovoA** and *GPU**. *DeNovoA** increases execution time and energy by 0.5% on average and reduces network traffic by 5% on average.

For LavaMD, *DeNovoA** significantly decreases network traffic because LavaMD overflows the store buffer, which prevents multiple writes to the same location from being coalesced in *GPU**. As a result, each of these writes has to be written through separately to the L2. Unlike *GPU**, after *DeNovoA** obtains ownership to a word, all subsequent writes to that word hit and do not need to use the store buffer.

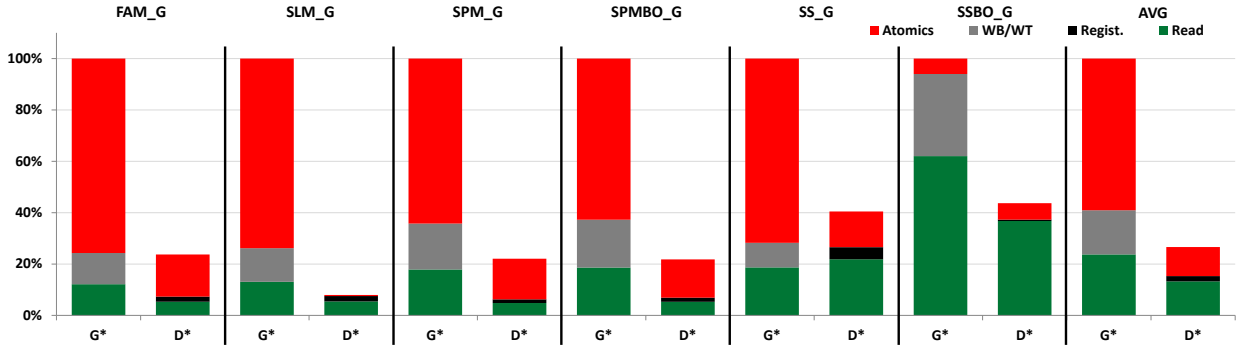
For some other applications, obtaining ownership causes *DeNovoA** to slightly increase network traffic and energy. First, multiple writes to the same word may require multiple ownership requests if the word is evicted from the cache before the last write. *GPU** may be able to coalesce these writes in the store buffer and incur a single writethrough to the L2. Second, *DeNovoA** may incur a read or registration miss for a word registered at another core, requiring an extra hop on the network compared to *GPU** (which always hits in the L2). In the applications, however, these sources of overheads are minimal and do not affect performance. In general, the first source (obtaining ownership) is not on the critical path for performance and the second source (remote L1 miss) can be partly mitigated (if needed) using direct cache to cache transfers as enabled by *DeNovoA* [46].



(a) Execution time



(b) Dynamic energy



(c) Network traffic

Figure 2.4: *G** and *D**, normalized to *G**, for globally scoped synchronization benchmarks.

Global Synchronization Benchmarks

Figure 2.4 shows the execution time, energy, and network traffic for the four benchmarks that use only globally scoped fine-grained synchronization. For these benchmarks, HRF has no effect because there are no synchronizations with local scope.

The main difference between *GPU** and *DeNovoA** is that *DeNovoA** obtains ownership for written data and global synchronization variables, which gives the following key benefits for the benchmarks with global synchronization. First, once *DeNovoA** obtains ownership for a synchro-

nization variable, subsequent accesses from all TBs on the same CU incur hits (until another CU is granted ownership or the variable is evicted from the cache). These hits reduce average synchronization latency and network traffic for *DeNovoA**. Second, *DeNovoA** also benefits because owned data is not invalidated on an acquire, resulting in data reuse across synchronization boundaries for all TBs on a CU. Finally, release operations require getting ownership for dirty data instead of writing through the data to L2, resulting in less traffic.

As discussed in Section 2.4.1, obtaining ownership can incur overheads relative to GPU coherence in some cases. One notable exception to this is SS_G. In SS_G, obtaining ownership for written data increases execution time (by 14% relative to *G**) because the data cannot be reused due to the next thread that accesses the data usually being remote. By adding backoff to the semaphore, semaphore contention is decreased and written data can be reused more often. Overall, for the benchmarks with global synchronization, these overheads are compensated by the reuse effects mentioned above. As a result, on average, *DeNovoA** reduces execution time, energy, and network traffic by 21%, 45%, and 73%, respectively, relative to *GPU**.

Local Synchronization Benchmarks

For the microbenchmarks with mostly locally scoped synchronization, we focus on comparing *DD* with *GH* since Figure 2.3 shows that the latter is the best GPU protocol.

DD and *GH* both increase reuse and synchronization efficiency relative to *GD* for applications that use fine-grained synchronization, but they do so in different ways. *GH* enables data reuse across local synchronization boundaries, and can perform locally scoped synchronization operations at L1. Therefore, these benefits can only be achieved if the application can explicitly define locally scoped synchronization points. In contrast, DeNovoA enables reuse implicitly because owned data can be reused across any type of synchronization point. In addition, *DD* obtains ownership for all synchronization operations, so even global synchronization operations can be performed locally. Like the globally scoped benchmarks, obtaining ownership for atomics also improves reuse and locality for benchmarks like TRB_LG, TRBEX_LG, LFTRB_LG, and LFTRBEX_LG that have both global and local synchronization.

Since *GH* does not obtain ownership, on a globally scoped release, it must flush and downgrade

all dirty data to the L2. As a result, if the store buffer is too small, then *GH* may see limited coalescing of writes to the same location, as described in Section 2.7.2. TRB_LG and TRBEX_LG, which like real world applications contain both local and global synchronization, exhibit this effect. *DD* also occasionally suffers from full store buffers for these benchmarks, but its cost for flushing is lower – each dirty cache line only needs to send an ownership request to L2. Furthermore, once *DD* obtains ownership, any additional writes will hit and do not need to use the store buffer, effectively reducing the number of flushes of a full store buffer. By obtaining ownership for the data, *DD* is able to exploit more reuse. In doing so, *DeNovoA* reduces network traffic and energy relative to *GH* for these applications.

Conversely, *DeNovoA* only enables reuse for owned data; i.e., there is no reuse across synchronization boundaries for read-only data. This hurts *DD*'s performance and increases network traffic with locally scoped synchronization. SS_L is also hurt by the order that the readers and writers enter the critical section: many readers enter first, so read-write data is invalidated until the writer enters and obtains ownership for it. *DD* also performs slightly worse than *GH* for UTS because DD uses global synchronization and must frequently invalidate the cache and flush the store buffer. Although ownership mitigates many disadvantages of global synchronization, frequent invalidations and store buffer flushes limit the effectiveness of *DD*.

On average, *GH* shows 5% lower execution time and 4% lower energy than *DD*, with maximum benefit relative to *GH* of 13% and 10% respectively. However, *GH*'s advantage comes at the cost of increased memory model complexity.

2.7.3 DD with Selective (RO) Invalidations

DD's inability to avoid invalidating read-only data is a key reason *GH* outperforms it for the locally scoped microbenchmarks. Using the read-only region enhancement for *DD*, however, removes any performance and energy benefit from *GH* on average. In some cases, *GH* is better, but only up to 7% for execution time and 4% for energy. Although *DD+RO* needs more program information, unlike HRF, this information is hardware agnostic.

2.7.4 Applying HRF to DeNovoA

DH enjoys the benefits of ownership for data accesses and globally scoped synchronization accesses as well as the benefits of locally scoped synchronization. Reuse in L1 is possible for owned data across global synchronization points and for all data across local synchronization points. Local synchronization operations are always performed locally, and global synchronization operations are performed locally once ownership is acquired for the synchronization variable.

Compared to *DD*, *DH* provides some additional benefits. With *DD* many synchronization accesses that would be locally scoped already occur at L1 and much data locality is already exploited through ownership. However, by explicitly defining local synchronization accesses, *DH* is able to reuse read-only data and data that is read multiple times before it is written across local synchronization points. It is also able to delay obtaining ownership for both local writes and local synchronization operations. As a result, compared to *DD*, *DH* reduces execution time, energy, and network traffic for all applications with local scope.

Although *DD+RO* allows reuse of read-only data, *DH*'s additional advantages described above also provide it a slight benefit over *DD+RO* in a few cases.

Compared to *GH*, *DH* is able to exploit more locality because owned data can be reused across any synchronization scope and because registration for synchronization variables allows global synchronization requests to also be executed locally.

These results show that *DH* is the best configuration of those studied because it combines the advantages of ownership (from *DD*) and scoped synchronization (from *GH*) to minimize synchronization overhead and maximize data reuse across all synchronization points. However, *DH* significantly increases memory model complexity and does not provide significantly better results than *DD+RO*, which uses a simpler memory model but has some overhead to identify the read-only data.

2.8 Summary

As CPUs and GPUs become tightly integrated into a single, unified address space with coherent caches [78, 79], data can be accessed simultaneously on both CPUs and GPUs without explicit

copies. Thus, GPUs need better support for issues such as coherence, synchronization, and memory consistency. The emerging interest in using GPUs for applications with more general-sharing patterns and fine-grained synchronization [38, 42, 146] only exacerbates these issues. Unfortunately, conventional GPU coherence protocols do not efficiently support fine-grained synchronization. Furthermore, modern GPU consistency models are difficult to use correctly and have been slow to be clearly defined [13, 104, 143].

In recognition of these issues and the need to support fine-grained synchronization on GPUs and accelerators, OpenCL 2.0 and the HSA Foundation recently adopted an HRF-like [63, 77] memory model (based on scoped synchronization) as their standards. HRF allows heterogeneous systems to efficiently support applications with fine-grained synchronization and avoids the complexity of CPU solutions like MESI, which provide poor performance for conventional GPU workloads [71, 141].

Our work is the first to question the conventional wisdom on the necessity of scoped synchronization and HRF in heterogeneous systems and to provide a high performance alternative. We show that DeNovo can be extended to heterogeneous systems to provide similar performance for conventional GPGPU applications **and** efficiently support fine-grained synchronization on GPUs *while retaining the common and less complex data-race-free memory model.*

Scoped synchronization and HRF are hardware-inspired concepts that essentially expose the memory hierarchy to software to enable hardware optimizations – synchronization scopes are intricately tied to the knowledge of which levels of the memory hierarchy are tied to which threads. This is a familiar but unfortunate trajectory taken by CPU memory models over the last several decades. CPU models were also developed in a similar hardware-centric fashion [7]; e.g., the IBM 370 and TSO models essentially exposed hardware write buffers. The complexity and confusion that resulted from such an approach with CPU memory models is legendary – there are still unresolved issues that are thorny largely because hardware evolved without sufficient consideration of the impact of the consistency model on software [6, 37].

This chapter shows that GPUs do not need the complexity of scoped synchronization and HRF to obtain high performance and energy efficiency. DeNovoA with DRF provides a sweet spot for performance, energy, hardware overhead, and memory model complexity – HRF's complexity is not needed to efficiently support fine-grained synchronization on GPUs. This shows that hetero-

geneous systems can avoid the trajectory taken by CPU memory models for the emerging world of heterogeneous systems – where simplicity and efficiency are only more important – without the painful, decades long process that was required for CPUs to converge towards data-race-free style models [4, 6, 36, 109]. In Chapter 3, we show how to extend this work to address the additional issues that relaxed atomics pose and in Chapter 4 we show how to extend DeNovoA to make specialized memories part of the unified address space.

Chapter 3

Efficient Support for and Evaluation of Relaxed Atomics

3.1 Motivation

In Chapter 2, we showed that the DRF0 memory consistency model (with the DeNovoA cache coherence protocol) provides a good both performance and programmability. However, in practice there are cases where DRF0's constraints on atomics can be relaxed with acceptable results, including some acceptable violations of SC. This motivated the addition of *relaxed atomics* to DRF0 for C++ (and later for other languages) and a departure from SC-centric semantics. Unfortunately, this departure has resulted in one of the most significant challenges in specifying concurrency semantics; despite more than a decade of effort, semantics that are weak enough to accommodate all desired optimizations but strong enough to enable reasonable analysis of programs have remained elusive [6, 29, 36, 37] (Section 3.1.1).

Furthermore, it is generally acknowledged that relaxed atomics are extremely difficult to use correctly; therefore, it is widely recommended that they be avoided and their use be left to experts [36, 154]. This was reasonable for CPUs since atomics are generally infrequent and SC (non-relaxed) atomics are implemented relatively efficiently, leveraging decades of experience with sophisticated coherence protocols. The situation, however, has been different for accelerators, exemplified by GPUs. As discussed in Chapter 2, current GPUs implement consistency through heavyweight coherence actions on conventional SC atomics (Section 2.2), making such atomics far more expensive than on CPUs and the potentially more lightweight relaxed atomics more tempting.

To demonstrate the benefits of using relaxed atomics in existing GPUs, we identified several GPU applications that use relaxed atomics (Section 3.5.2) and evaluated them on an NVIDIA GeForce GTX680. Figure 3.1 shows the speedup of using relaxed atomics instead of SC atomics for the 9 applications [38, 42, 61, 119, 124, 137, 145] with the highest percentage of atomics (as de-

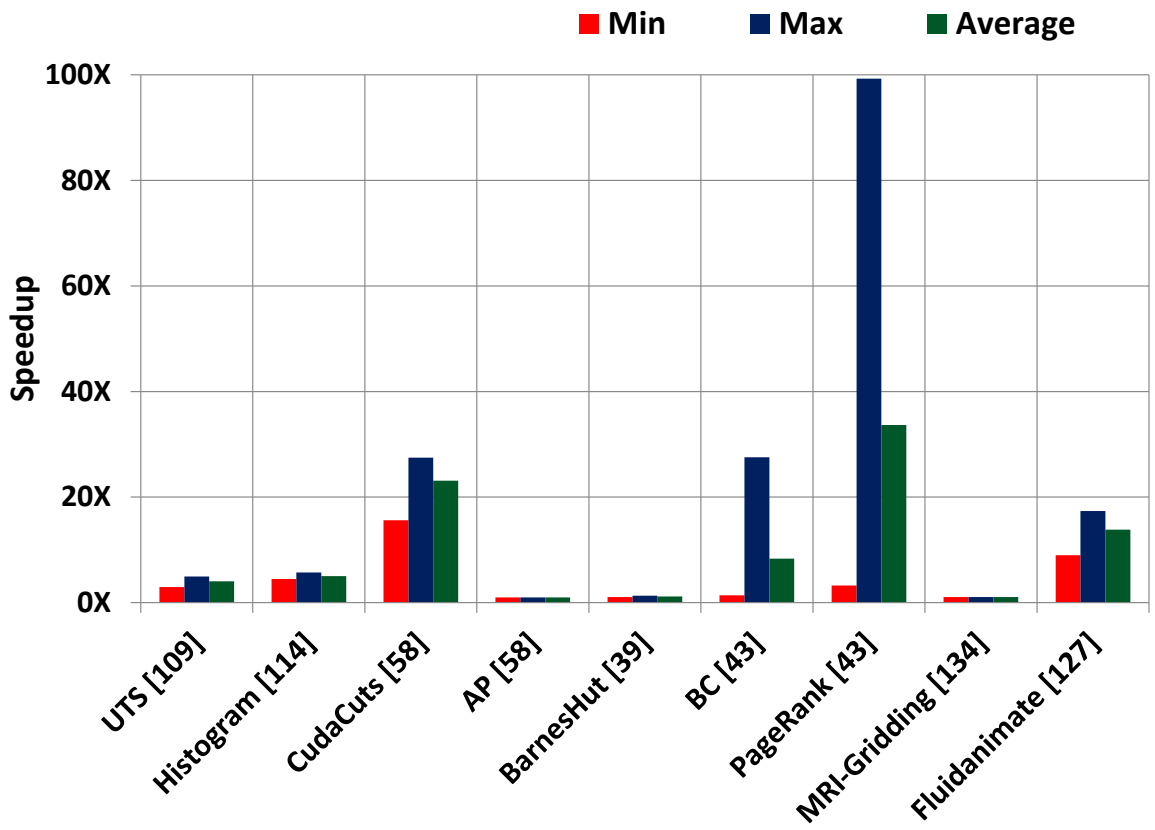


Figure 3.1: Relaxed atomics speedup on a discrete GPU.

termined from dynamic instruction profiling). Although relaxed atomics do not affect performance for some benchmarks, other benchmarks see huge benefits (e.g., up to a 99X speedup for PageRank), motivating the need for reasonable semantics for relaxed atomics. The results of this chapter (Section 3.6) show that for future integrated CPU-GPU systems with coherent caches and a global address space (with arguably faster synchronization than current discrete GPUs), the benefits are less dramatic, but high enough that relaxed atomics will still be tempting.

We make two broad contributions. First, we develop new SC-centric semantics for relaxed atomics using a distinct approach that extends the DRF consistency model from Chapter 2. Second, we provide, to our knowledge, the first quantitative evaluation of relaxed atomics on integrated CPU-GPU systems. We next give an overview of the state-of-the-art for relaxed atomics and then describe our approach and contributions.

3.1.1 Current Semantics with Relaxed Atomics

The key difficulty underlying efforts to formalize relaxed atomics has been the requirement to prohibit executions that generate values from "out-of-thin-air" due to self-satisfying speculations within causal loops [36, 109]. Much has been written about the many subtleties of this problem [6, 28, 29, 36, 37, 83, 123]; we accept its difficulty and only discuss the state-of-the-art for dealing with it.

The Java memory model was the first to struggle with the out-of-thin-air issue. It attempted to define semantics for programs with data races that were weak enough to enable all desirable optimizations but strong enough to preserve security considerations by minimally prohibiting out-of-thin-air values. The resulting model is extremely complex [109], was later discovered to have a flaw that unintentionally precluded key optimizations, and remains unfixed [158].

C++11 is different from Java in that it was deemed reasonable to give programs with data races undefined semantics (C++ gives undefined semantics for other situations as well). However, it ran into Java-like problems when attempting to formalize semantics for relaxed atomics that were purported to avoid the thin-air problem while enabling desirable optimizations for such accesses. Unfortunately, a fatal flaw was again discovered in the C++11 formalism that remains to be fixed [34, 36]. C++14 therefore just gives informal wording that systems should not produce out-of-thin-air values [35].

Boehm and Demsky have proposed a system restriction to deal with the problem, but it has not yet been accepted by vendors [37]. Furthermore, two models have recently been proposed that claim to enable all desired optimizations and prevent out-of-thin-air values [83, 123]; however, they are based on complex theories such as event structures and promises, which seem difficult for most programmers to use.

In summary, all current approaches to formalize relaxed atomics are acknowledged to have significant limitations. Furthermore, all *require giving up the familiar interface of SC, even if there is a single relaxed atomic in the program, including one buried in invisible library code.* We discuss related work further in Section 6.2.

3.1.2 Approach for Semantics

Prior approaches focused on defining a system that enables desirable optimizations for relaxed atomics (specifically, reordering relaxed atomics with respect to each other and data accesses) without allowing "out-of-thin-air" values. So far, this approach has failed because what constitutes "out-of-thin-air" has been difficult to pin down and arbitrary accesses may be distinguished as relaxed atomics.

We take a different approach motivated by how we see developers wanting to use relaxed atomics. Specifically we ask the following questions. *What are the common uses of relaxed atomics? Can we characterize these uses in terms of their properties in SC-centric executions? Can we then express the model as ensuring SC-centric behavior for programs that use relaxed atomics only for the specified use cases (i.e., only if the specified properties are obeyed by the program)?* This is precisely the approach that led to the programmer-centric data-race-free class of models [4, 5]. By stating a priori a set of requirements for accesses that can be distinguished as relaxed atomics, we reduce the scope of the problem and make it easier to find a reasonable solution.

Comparing again to the approach of the original DRF models [4, 8], that work examined the optimizations that were being proposed by the "hardware-centric" models of the day (e.g., weak ordering [57], processor consistency [66], release consistency [65], etc.) and determined how to characterize memory accesses where such optimizations would be safe (i.e., not violate SC). Thus, a key insight of DRF0 was that memory accesses not involved in a race, informally referred to as data accesses, could be reordered without violating SC. Later versions of the models discovered other characterizations that led to more optimizations; e.g., the DRF1 model characterized paired vs. unpaired atomics where unpaired atomics did not require any ordering constraints relative to data accesses [8].

To identify use cases for relaxed atomics, we reached out to vendors, developers, and researchers active in this area. We developed a new model, *DRFrlx*, that captures these use cases within an SC-centric form. We discovered five use cases.

(1) *Unpaired atomics*: Several relaxed atomics were the unpaired atomics already characterized by DRF1 [8], which is already SC-centric (Section 3.2).

(2) *Commutative atomics*: These relaxed atomics incurred racy interactions only using operations

that are commutative. They required a minor adjustment to the definition of SC to accommodate standard relaxed atomic optimizations within an SC-centric framework.

(3) *Non-ordering atomics*: These atomics are involved in racy interactions, but these interactions are never responsible for creating an order between other accesses. Again, relaxed atomics style optimizations can be performed on such accesses without violations of SC.

(4) *Speculative atomics*: To avoid the high overhead of synchronizing, some applications (e.g., seqlocks [33]) speculatively read shared data to enable concurrent readers, without proper synchronization. If a write occurs concurrently, the speculative reads are discarded. Even though the speculative reads may produce inconsistent, non-SC values, these values do not affect the final result. we call such accesses speculative atomics and provide SC-centric semantics for them by effectively adjusting the definition of SC to ignore accesses that do not affect the final result.

(5) *Quantum atomics*: Some uses of relaxed atomics truly violate SC. Programmers justify such atomics as being truly robust and resilient to a large range of approximate (non-SC) values (e.g., split counters [111]). We call such cases quantum atomics and explicitly exploit the intuition that their values are resilient – we require programmers to reason about correctness given that a quantum load may return (almost) any value. To facilitate this, we define a quantum-equivalent program that (logically) replaces quantum accesses with functions returning random values and require SC semantics for such programs (with some additional properties). This may seem bizarre at first; however, these uses of relaxed atomics have been justified in the past as being resilient to many bizarre outcomes, we simply make that expectation explicit in the model, and only for this sub-class of relaxed atomics. Our expectation is that by clearly stating this requirement, the use of such atomics will be restricted to scenarios where such analysis is reasonable; e.g., where a quantum atomic cannot affect the address of a reference or lead to intuitively impossible control flow.

In summary, like other DRF models, DRFrlx is specified as a contract between the programmer and the system. It requires that all atomics be distinguished as SC atomics or one of the above relaxed atomics (which must obey the above properties). In return, the system will appear SC (for that program or its quantum-equivalent program). Thus, DRFrlx solves a long-standing problem in the concurrency community. Although DRF0 and DRF1 are simpler than DRFrlx, in practice

their implementations in modern programming languages are made complicated by the addition of relaxed atomics. DRFrlx provides the same semantics as DRF0 and DRF1 when relaxed atomics are not used and simpler semantics for relaxed atomics than the state-of-the-art.

We do not claim that our approach covers every possible use case of relaxed atomics. Further, we focus on the `memory_order_relaxed` version of relaxed atomics as defined by C++. We did not explore other relaxed orderings such as `memory_order_acquire` and `memory_order_release` (briefly discussed in Section 6.2). Instead we cover all common use cases of `memory_order_relaxed` with reasonable-to-use semantics. In particular, a relaxed atomic within a library function of a legal DRFrlx program does not require a user to understand the function's implementation as long as the library writer can convey the expected pre- and post-conditions for SC executions of the (quantum-equivalent) program.

Although DRFrlx also applies to multi-core CPUs, we evaluate DRFrlx for GPU based systems since, as discussed above, heavyweight GPU actions on atomics make relaxed atomics attractive. To determine if the complexity of relaxed atomics is worthwhile for CPU-GPU systems, we created benchmarks based on the use cases we gathered and identified applications in standard benchmark suites that use relaxed atomics. Our evaluation (Section 3.6 show mixed results: for some applications relaxed atomics significantly improve performance, energy, and network traffic, while for others the benefits are small. Regardless, on average DeNovoA provides better performance, energy, and network traffic than GPU coherence, which confirms our findings in Chapter 2.

3.2 Background: DRF1 Consistency Model

The DRF1 memory consistency model removes some ordering constraints from DRF0 to improve performance. DRF1 utilizes additional information from the software to distinguish between *paired* read-write synchronization atomics that order racing data operations and *unpaired* atomics do not order data operations [8]. DRF1 allows unpaired atomics to be reordered with respect to data operations without violating SC for DRF1 programs (defined below).

3.2.1 Terminology

We use the following terminology throughout the rest of this chapter [8]. The runtime system (usually the hardware) executes the program for some input data, and performs the updates specified in the program – potentially resulting in several possible executions of the program. An *operation* is a memory access that either reads a memory location (a load) or modifies a memory location (a store). Two memory operations *conflict* if they access the same memory location and at least one of them is a store. Two memory operations, *op1* and *op2*, are ordered by *program order* (*op1* $\xrightarrow{\textbf{po}}$ *op2*) if and only if both are from the same thread and *op1* is ordered before *op2* by the program text. An execution is *sequentially consistent (SC)* if there exists a total order, T, on all its memory operations such that (i) T is consistent with *program order* and (ii) a load L returns the value of the last store S to the same location ordered before L by T. We refer to T as an *SC total order* for the execution.[1]

3.2.2 DRF1 Formal Definition

All memory operations are distinguished to the system as either data or atomic. An atomic operation is distinguished as either paired or unpaired.[2]

Definitions for an SC Execution with *SC total order T*:

Synchronization Order 1 ($\xrightarrow{\textbf{so1}}$): Let X and Y be two memory operations in an execution. $X \xrightarrow{\textbf{so1}} Y$ if and only if X and Y conflict, X is a paired synchronization write, Y is a paired synchronization read, and X is ordered before Y in the *SC total order*.

Happens-before-1 ($\xrightarrow{\textbf{hb1}}$): The happens-before-1 relation is defined on the memory operations of an execution as the irreflexive transitive closure of po and so1: $hb1 = (po \cup so1)+$.

Race: Two operations X and Y in an execution form a race (under DRF1) if and only if X and Y are from different threads, they conflict with each other, and they are not ordered by happens-before-1.

Data Race: Two operations X and Y form a data race (under DRF1) if and only if they form a race and at least one of them is distinguished as a data operation.

[1] For brevity, we refer the reader to [5] for formal definitions of several intuitive notions. Informally, an *execution* must obey correctness requirements for a single core. To accommodate *read-modify-writes* (RMW), the read (load) and write (store) of a RMW must appear together in an SC total order.

[2] Paired atomics are the equivalent of SC atomics in the C and C++ models [36].

Relaxed Atomic Category	Application
Unpaired (Section 3.3.1)	Work Queue [63]
Commutative (Section 3.3.2)	Event Counter [36, 39, 124, 154]
Non-Ordering (Section 3.3.3)	Flags [154]
Speculative (Section 3.3.4)	Multiple Locks [64], Seqlocks [33]
Quantum (Section 3.3.5)	Split Counter [111], Reference Counter [114, 154]

Table 3.1: GPU relaxed atomic use cases.

Program and Model Definitions:

DRF1 Program: A program is DRF1 if and only if for every SC execution of the program, all operations can be distinguished by the system as either data, paired atomic, or unpaired atomic, and there are no data races (under DRF1) in the execution.

DRF1 Model: A system obeys the DRF1 memory model if and only if the result of every execution of a DRF1 program on the system is the result of an SC execution of the program.

Optimizations: In addition to allowing all of the optimizations of DRF0, DRF1 also allows unpaired atomics to be reordered with respect to data operations, without violating SC for DRF1 programs.

Mechanism for distinguishing memory operations:

Data-race-free-1 requires that data operations can be distinguished from atomic operations, and that paired atomics can be distinguished from unpaired atomics. We reuse existing C++ support to distinguish data and atomic operations, and we add a new annotation to distinguish paired and unpaired operations, similar to how C++ distinguishes SC atomics and relaxed atomics [142].

3.3 Relaxed Atomic Use Cases and DRFRlx Model

We collected examples of how programmers use relaxed atomics and categorized them in Table 3.1 based on what type of race occurs in the program: unpaired, commutative, non-ordering, quantum, or speculative. We turn these examples into microbenchmarks in the HeteroSync benchmark suite. Although our sources contain additional examples that use relaxed atomics, we do not discuss them because they are similar to these examples. Based on these use cases, we introduce DRFrlx, which extends DRF0 and DRF1 [5] to allow certain relaxed atomics to be reordered without compromising SC-centric semantics.

```
struct Task;
struct MsgQueue {
  atomic<int> _occupancy = 0;

  Task * dequeue() {
    if (_occupancy.atomic_load(mem_order_seq_cst) == 0) {
      return NULL;
    } else { ... }
  }
  int occupancy() {
    return _occupancy.atomic_load(mem_order_relaxed);
  }
  ...
} globalQueue;

// Thread t1 (service thread):
  void periodicCheck() {
    if (globalQueue.occupancy() > 0) {
      Task * t = globalQueue.dequeue();
      if (t != NULL)
        t.execute();
    }
  }
```

Listing 3.1: Work Queue example [63].

3.3.1 Unpaired Atomics

Unpaired Atomics Use Case

Work Queue [63]: In Listing 3.1, a service thread and client thread use a shared work queue to communicate.[3] The service thread periodically checks whether the client thread has requested service from it by reading from an incoming message queue, and is guaranteed to service all the clients' requests. When there are no new messages (the common case), nothing needs to be done and the service thread continues to do other work (on other data, not shown). Although the occupancy checks occur frequently, the service threads' atomics only need to order data when the queue is not empty.

If this example were constrained to using paired (i.e., SC) atomics, then every occupancy check must invalidate the entire L1 cache with current GPU protocols. In the common case of an empty queue, this invalidation is unnecessary and precludes data reuse. Moreover, Work Queue can use relaxed atomics in `occupancy` without violating SC, because all memory accesses will be ordered by the SC atomic in `dequeue`. By using an unpaired atomic for the occupancy check, DRF1 removes the need to invalidate the cache when the queue is empty, enables unpaired operations to

[3] All listings use C/C++ convention – mem_order_seq_cst and mem_order_relaxed identify SC and relaxed atomics, respectively.

```
atomic<int> count[NUM_BINS]; // all bins initialized to 0

// Threads 1..N:
threadNum = ...
chunkSize = ...
baseLoc = (threadNum * chunkSize);
...
for (i = 0; i < chunkSize; ++i) {
  binNum = data[baseLoc + i] % NUM_BINS;
  count[binNum].atomic_inc(mem_order_relaxed);
}
...

// Main Thread:
main() {
  launch_workers(); // launch worker threads
  ...
  join_workers();
  for (i = 0; i < NUM_BINS; ++i) {
    int r1 = count[i].atomic_load(mem_order_relaxed);
    ... // uses r1
  }
}
```

Listing 3.2: Event counters example [36, 39, 113, 124, 154].

be reordered with respect to data operations, and provides benefits similar to relaxed atomics.[4]

DRF1 provides most of the benefits of relaxed atomics for **Work Queue** by removing the ordering constraint between data and unpaired atomics (while preserving SC); however, unlike relaxed atomics, DRF1 constrains unpaired atomics to respect program order with respect to other unpaired atomics. New classes of relaxed atomics discussed in the rest of this section relax this constraint.

3.3.2 Commutative Atomics

Commutative Atomics Use Case

Event Counter [36, 39, 113, 124, 154]: In event counters, such as histograms, multiple worker threads concurrently increment shared global counters, as illustrated in Listing 3.2. Since these increments form a race, they must be distinguished as atomic.[5] Straightforward DRF0/1 implementations would serialize program ordered increments and, for current GPU protocols, invalidate

[4] If Work Queue uses multiple occupancy queues, then relaxed atomics could potentially violate SC. However, since these accesses are amenable to approximation, and the queues' values will be double-checked by the `dequeue` function, SC-centric semantics are retained by distinguishing these accesses as quantum atomics (Section 3.3.5).

[5] Programmers can use private, per-core counters to stage updates locally before doing a single update of the global counters [113, 124, 163]. However, eventually these updates must be done globally, and these updates will conflict with one another.

the L1 cache, and flush the store buffer. On inspection, however, one can reason that reordering the increments produces acceptable results; therefore, common uses distinguish the increments as relaxed atomics.

Commutative Atomics Informal Intuition

We make the following key observations that enable us to formalize the intuition behind the safe reordering of the increments in Listing 3.2: (i) racing increments in an execution of Listing 3.2 are commutative and give the same result regardless of their order of execution, (ii) the values they load are not used elsewhere (and so need not be considered as part of the result of the execution), and (iii) the final incremented value is loaded only after another paired synchronization interaction (a barrier from the join in the listing). We refer to atomics with the above properties as *commutative atomics*, formalized below.

We can now reason that the execution order of racy commutative atomics does not impact the final result of the execution and cannot be used to infer the ordering of other operations in the execution. Further, the load of the final value after all the racy, conflicting commutative atomics is always separated by paired (SC) synchronization; therefore, the load does not rely on the ordering of commutative atomics (with respect to other relaxed atomics) for its correct value. Thus, reordering commutative atomics with respect to other program ordered relaxed atomics (unpaired, commutative, and others discussed later) produces a result that is equivalent to an SC execution of the program.

The above reasoning uses a slight departure from the conventional notion of what constitutes the "result" of an (SC) execution. Much literature considers the value returned by each load to be part of the result. We define the result to be the memory state at the end of the (SC) execution.[6] Thus, we can ignore the values of reads that do not affect the final memory state when considering if an execution is SC.

[6]For brevity, our formalism assumes finite SC executions. We can handle infinite executions as in [5] – we assume that any prefix of an SC total order is finite and consider the memory state at the end of appropriate finite prefixes as the results. For simplicity, we also ignore external outputs in our definition of result; again, this can be easily incorporated similar to [5] and does not affect our model specifications.

DRFrlx Formal Definition (Version 1)

Since DRFrlx extends DRF1, we only list the components of DRFrlx version 1 that differ from DRF1. All memory operations need to be identified as data, paired, unpaired, or *commutative*.

Definitions for an SC Execution:

Result of an execution: The memory state at the end of the execution.

Commutativity: Two stores or RMWs to a single memory location M are *commutative with respect to each other* if they can be performed in any order and yield the same value for M.

Commutative Race: Two operations X and Y form a commutative race if and only if:

1. X and Y form a race,
2. at least one of X or Y is distinguished as a commutative operation, and
3. X and Y are not commutative with respect to each other or the value loaded by either operation is used by another instruction in its thread.

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of the program:

- all operations can be identified by the system as either data or as paired, unpaired, or commutative atomics, and
- there are no data races or commutative races in the execution.

DRFrlx Model: A system obeys the DRFrlx memory model if and only if the result of every execution of a DRFrlx program on the system is the result of an SC execution of the program.

3.3.3 Non-Ordering Atomics

Non-Ordering Atomics Use Cases

Flags [154]: Listing 3.3 uses shared global flags to communicate between threads. Worker threads repeatedly loop until the `stop` flag is set. Within the loop, if certain conditions are met, a worker sets the `dirty` flag to signify something has been accessed that the main thread needs to clean up later (`cleanup_dirty_stuff`). Once the main thread has set `stop`, the workers will all exit. A

```
atomic<bool> dirty = false, stop = false;

// Threads 1..N:
...
while (!stop.atomic_load(mem_order_relaxed)) {
  if (...) {
    dirty.atomic_store(true, mem_order_relaxed);
    ...
  }
  ...
}

// Main Thread:
main() {
  launch_workers(); // launch threads 1..N
  ...
  stop.atomic_store(true, mem_order_relaxed);
  join_workers();
  if (dirty.atomic_load(mem_order_relaxed))
    cleanup_dirty_stuff();
}
```

Listing 3.3: Flags example [154].

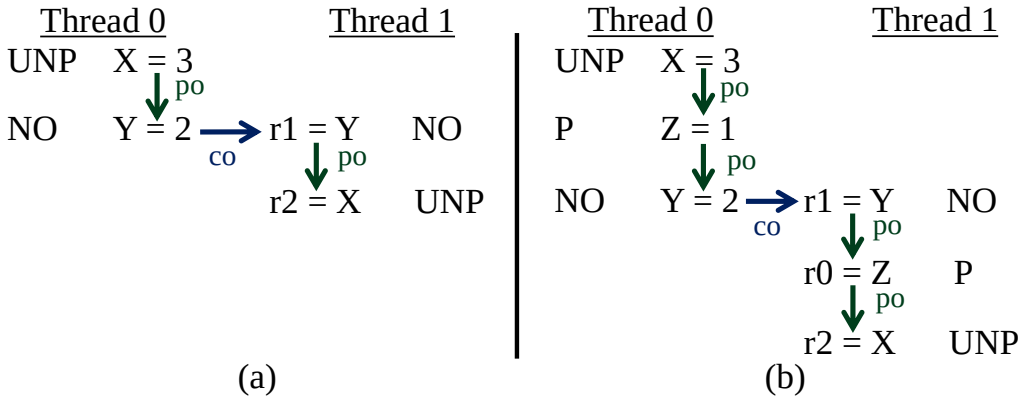


Figure 3.2: Executions with program/conflict graphs and ordering paths, (a) with a non-ordering race and (b) without a non-ordering race. UNP = unpaired, NO = non-ordering, P = paired.

global barrier (`join_workers`) ensures that all worker threads exit before the main thread loads the `dirty` flag.

Both `dirty` and `stop` must be distinguished as atomics. The stores to `dirty` can be distinguished as commutative since they obey all the necessary requirements. Before the barrier, `stop` is simultaneously accessed by all threads without any intervening paired atomic. We can informally reason that making the operations to `stop` relaxed will not violate SC for this code. Similarly, we can also reason that the load of `dirty` after the global barrier can also be relaxed.

Non-Ordering Atomics Informal Intuition

The key intuition behind why the operations to `stop` and `dirty` can be relaxed is that they are not being used to order any other operations – the global barrier orders any conflicting operations that need to be ordered. Thus, reordering the operations to `stop` and `dirty` with respect to other relaxed operations will not violate SC.

To exploit the above intuition, we formalize what it means for a pair of conflicting (racing) operations to "not order" other operations (using formalism developed in [5]), and call such operations *non-ordering* atomics. We use the notion of a program/conflict graph, which captures program order and the execution order of conflicting operations in an execution. For SC, this graph must be acyclic. That is, if there is a path in this graph from an operation X to a conflicting operation Y, then X must execute before Y to prevent a cycle. In general, there can be many such "ordering paths" from X to Y. As long as the system guarantees that one such path is enforced, a cycle will be avoided – operations on other paths between X and Y may be reordered. Thus, non-ordering operations are those that either don't occur on ordering paths, or are absolved of ordering responsibilities because there is always another path that enforces the ordering. We refer to the latter path as a *valid path*. These concepts are formalized next and illustrated in Figure 3.2.

DRFrlx Formal Definition (Version 2)

For simplicity, we only show the new DRFrlx components and the components that are modified from Section 3.3.2 to support non-ordering atomics. In version 2 of DRFrlx, all memory operations must be distinguished as data, paired, unpaired, commutative, or *non-ordering*.

Definitions for an SC Execution with *SC total order T*:

Conflict Order ($\xrightarrow{\text{co}}$): $X \xrightarrow{\text{co}} Y$ if and only if X and Y conflict and X is ordered before Y in T.

Program/Conflict Graph and a Path [5]: The **program/conflict graph** of an execution is a directed graph where the vertices are the (dynamic) operations of the execution and the edges represent program order and conflict order. Below all paths refer to paths in the program/conflict graph.

Ordering Path [5]: A path from X to Y is called an **ordering path** if it has at least one program order edge and X and Y conflict.

Valid Path [5]: An ordering path is **valid** if all its edges are: (1) $\xrightarrow{\textbf{hb1}}$, or (2) between atomic accesses to the same address, or (3) between paired and/or unpaired accesses.

Non-ordering Race: Two operations, X and Y form a non-ordering race if and only if:

1. X and Y form a race, both are distinguished as atomics, and at least one of them is distinguished as a non-ordering atomic, and
2. $X \xrightarrow{\textbf{co}} Y$ is on an ordering path from A to B, but there is no valid path from A to B.

Figure 3.2 shows two example executions with their program/conflict graphs and ordering paths. In Figure 3.2(a), there is only one ordering path between the conflicting operations on X: $X = 3 \xrightarrow{\textbf{po}} Y = 2 \xrightarrow{\textbf{co}} r1 = Y \xrightarrow{\textbf{po}} r2 = X$. Since this path contains a non-ordering atomic, a non-ordering race occurs. Figure 3.2(b) adds a new ordering path: $X = 3 \xrightarrow{\textbf{po}} Z = 1 \xrightarrow{\textbf{co}} r0 = Z \xrightarrow{\textbf{po}} r2 = X$. Since the operations on Z are paired, this forms a valid path between the operations on X and there is no longer a non-ordering race in this execution.

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of the program:

- all operations can be distinguished by the system as either data or as paired, unpaired, commutative, or non-ordering atomics, and
- there are no data races, commutative races, or non-ordering races in the execution.

3.3.4 Speculative Atomics

Speculative Atomics Use Cases

Multiple Locks ([64]): Some applications, such as the example shown in Listing 3.4, use multiple locks `x` and `y`. In this listing, each thread initially holds one of the locks, then releases it and loops until it obtains the other lock. If the unlock accesses are relaxed such that the lock accesses can be overlapped with them, then it is possible to initially see the lock as held, and repeatedly try to obtain the lock – a violation of SC.

Seqlocks [33]: In applications where updates are infrequent, it is often safe for a thread to load shared data without acquiring a lock because usually there are no concurrent writes. In Listing 3.5,

```
atomic<int> x, y;

// Thread 0 (initially holds x)
...
A = 1;
unlock x;
lock y; // loops until it obtains the lock
r1 = B;
...

// Thread 1 (initially holds y)
...
B = 1;
unlock y;
lock x; // loops until it obtain the lock
r2 = A;
...
```

Listing 3.4: Multiple locks example [64].

a reader *speculatively* loads shared data (`data1`, `data2`). If there are no concurrent writers (the common case), then the readers can safely use `data1` and `data2` in subsequent instructions (not shown in Listing 3.5). However, the reader must reload the shared data if a writer is concurrently updating the shared data.

Seqlocks uses a shared sequence number (`seq`) to synchronize the concurrent loads and stores to the shared data. A reader loads `seq` before and after the speculative data loads to check for concurrent writers. If the reader's sequence numbers do not match or are odd, then there is a concurrent writer. Writers make `seq` odd to indicate that an update is in progress. Once the update is complete, the writer updates `seq` to be the next even value.

Both `data` and `seq` must be distinguished as atomics. However, as discussed previously, requiring SC atomics unnecessarily hurts performance. The data accesses can be relaxed – the stores only race with loads and the results of racy loads get discarded, ensuring that these races do not affect the final result. The `seq` accesses ensure that the final data accesses whose values are used do get properly synchronized and ordered.[7]

Speculative Atomics Informal Intuition

Although Multiple Locks violates SC, this violation can be ignored because it does not affect the final result. Eventually each thread will see that the other thread has released the other

[7]The reader's `seq` accesses can also be relaxed to acquire and release ordering; we discuss this further in Section 6.2. Note that the `seq1` access uses an unusual "read-don't-modify-write" operation (instead of a plain read) to generate release semantics as explained further in [33].

```
atomic<unsigned> seq;
atomic <int> data1, data2;

T reader() {
  int r1, r2;
  unsigned seq0, seq1;
  do {
    seq0 = seq.atomic_load(mem_order_seq_cst);
    r1 = data1.atomic_load(mem_order_relaxed);
    r2 = data2.atomic_load(mem_order_relaxed);
    seq1 = seq.atomic_fetch_add(0, mem_order_seq_cst);
  } while ((seq0 != seq1) || (seq0 & 1));
  // uses r1 and r2
}

void writer(...) {
  unsigned seq0 = seq.atomic_load(mem_order_seq_cst);
  while ((seq0 & 1) || !seq.atomic_cmp_exchange_weak(seq0, seq0+1)) { ; }
  data1.atomic_store(..., mem_order_relaxed);
  data2.atomic_store(..., mem_order_relaxed);
  seq.atomic_store(seq0 + 2, mem_order_seq_cst);
}
```

Listing 3.5: Seqlocks example [33].

lock, and will obtain that lock. Although relaxing Seqlocks' loads to `data1` and `data2` may read some inconsistent, non-SC values, any misspeculated values will not be used because the sequence numbers will not match. Thus, speculatively accessing the shared data does not violate SC. The stores `data1` and `data2` can also be relaxed without violating SC because they only race with the misspeculated loads. To exploit this intuition, we formalize what it means for a racing access to be "speculative" and call such operations *speculative atomics*.One way to formalize this and ensure the final result is always SC is to require that values returned by racy speculative loads are never used, as in Multiple Locks and Seqlocks.[8] We formalize this next.

DRFrlx Formal Definition (Version 3)

We only show the parts that change from Section 3.3.3. All memory operations must be distinguished as data, paired, unpaired, commutative, non-ordering, or *speculative*.

Definitions for an SC Execution:

Speculative Race: Two operations, X and Y, form a speculative race if and only if they form a

[8]This concept can be generalized to allow speculative atomics to use their returned values, but only within the speculative part of the program (so they do not affect the final result). It can also be potentially generalized to "read-copy-update" (RCU) patterns [54, 111]. These generalizations are left for future work, but exploit PL_{pc}'s [5, 64] observation that unessential operations can be ignored when reasoning about an execution's DRF properties because they do not affect the final result.

```
atomic<unsigned long> myCount[NUM_THREADS];
add_split_counter(v, tID) {
  val = myCount[tID].atomic_load(mem_order_relaxed);
  newVal = val + v;
  myCount[tID].atomic_store(newVal, mem_order_relaxed);
}
read_split_counter(tID) {
  sum = 0;
  for (i = 0; i < NUM_THREADS; ++i) {
    loc = ((tID + i) % NUM_THREADS);
    sum += myCount[loc].atomic_load(mem_order_relaxed);
  }
  return sum;
}

add_split_counter(1, 0); // Thread 0
r1 = read_split_counter(1); // Thread 1
add_split_counter(2, 2); // Thread 2
r2 = read_split_counter(3); // Thread 3
```

Listing 3.6: Split counters example [111].

race, at least one of X or Y is distinguished as a speculative atomic, and either:

- both operations are stores, or
- the result of the load is observed by another instruction in the execution (i.e., the returned value is used by another instruction in the thread).

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of its program:

- all operations can be distinguished by the system as either data or as paired, unpaired, commutative, non-ordering, or *speculative* atomics, and
- there are no data races, commutative races, non-ordering races, or *speculative races* in the execution.

3.3.5 Quantum Atomics

Quantum Atomics Use Cases

So far, the relaxed atomics use cases either do not violate SC or produce a result that is equivalent to the final result of an SC execution. However, in some situations, SC violations may be tolerable. Two use cases where this occurs are split counter, described next, and reference counter. To

incorporate these use cases, we extend DRFrlx with a new category, quantum, that is more complex than the extensions we have discussed thus far.

Split Counter [111]: In Listing 3.6, some threads update their counter while other threads load the current partial sum of all counters, all without adequate synchronization to preserve mutual exclusion. Since multiple threads can concurrently load and store the counters in `myCount`, the operations form races and need to be distinguished as atomics. Commutative atomics may not be used here because the return value of a racing operation is observed by other instructions in the thread. Non-ordering atomics may not be used because these operations form unique ordering paths between other racing operations. More fundamentally, relaxing these atomics can cause non-SC behavior. Even so, relaxed atomics are often used for split counter operations because developers are willing to trade off SC semantics (and precise partial sums) for improved performance [111]. By allowing atomics in `read_split_counter` to be reordered with both data and other atomics, Split Counter provides a fast, reasonable approximation of the current partial sum.

Quantum Atomics Informal Intuition

To obtain the performance benefits of relaxed atomics in Split Counter with SC-centric semantics, we define another class of relaxed atomics: *quantum atomics*. Quantum atomics can be reordered with respect to all relaxed atomics (and data). To isolate the non-SC behavior that may result from quantum relaxation, we conceptually build a new program where each quantum load returns a random, arbitrary value and each quantum store stores a random, arbitrary value. The transformed program must obey the appropriate race-free properties and appear SC. This transformation isolates the parts of the application that are not dependent on the quantum atomics from the parts that are dependent on it, thereby allowing the reasoning about the latter part in terms of SC. We refer to this transformation as the *quantum transformation*, the transformed program as the *quantum-equivalent program*, and use the term SC-centric to refer to models that provide SC semantics but only to quantum-equivalent programs (with the appropriate race-free properties, formalized below).

When inspecting a quantum-equivalent program for illegal races, quantum accesses are only allowed to race with other quantum accesses. In this way, quantum differs from other types of atomics, which can safely upgrade to a stronger atomic type without introducing new races. The

reason quantum accesses may not race with stronger atomic types is because the presence of a racy non-quantum access imposes constraints on the possible intermediate values of the target data in a quantum-equivalent program. These constraints may be inconsistent with the (possibly non-SC) behavior of the original program on a compliant DRFrlx system (defined in Section 3.3.7).

To provide some constraint on the values of quantum operations, we impose happens-before consistency and per-location SC (sometimes referred to as cache coherence) on these operations (similar to relaxed atomics in C/C++). However, these constraints are post facto – programmers must still commit to reasoning about race-free properties and SC executions only with the quantum-equivalent programs. While it may appear bizarre and against the grain of SC to require the programmer to reason about paths taken for arbitrary values for a quantum load, it directly exploits the fact that the reason programmers want to use relaxed loads in Split Counter is that they are amenable to imprecision.

DRFrlx Formal Definition (Version 4)

We only show the new components of DRFrlx and the components that are modified from Section 3.3.4. All memory operations must be distinguished as data, paired, unpaired, commutative, non-ordering, speculative or *quantum*.

Definitions for a SC-Centric Execution:

Happens-Before Consistency: A load L must always return the value of a store S to the same memory location M in the execution. It must not be the case that $L \xrightarrow{\textbf{hb1}} S$ or that there exists another store S' to M such that $S \xrightarrow{\textbf{hb1}} S' \xrightarrow{\textbf{hb1}} L$.

Per-Location SC: There is a total order, T_{loc}, on all operations to a given memory location M such that T_{loc} is consistent with **happens-before-1**, and that a load L returns the value of the last store S to M that is ordered before L by T_{loc}.

Quantum-Equivalent Program: We generate a quantum-equivalent program Pq from a program P as follows. Each quantum atomic load *ri = Y;* in P is replaced with a call to a conceptual arbitrary function *ri = arbitrary();* in Pq. Similarly, each quantum atomic store $Y = rj$ is replaced with a quantum store of an arbitrary value $Y = arbitrary()$. A quantum RMW's load and store are both replaced as above. Exactly what values may be returned by $arbitrary()$ is implementation specific.

```
atomic<unsigned long> refcount1, refcount2;

// Thread 1
refcount1.atomic_inc(mem_order_relaxed);
refcount2.atomic_inc(mem_order_relaxed);
...
if (refcount1.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr1 to be deleted
if (refcount2.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr2 to be deleted

// Thread 2
refcount1.atomic_inc(mem_order_relaxed);
refcount2.atomic_inc(mem_order_relaxed);
...
if (refcount2.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr2 to be deleted
if (refcount1.atomic_dec(mem_order_relaxed) == 0)
  mark cb_ptr1 to be deleted
```

Listing 3.7: Reference counter example [154].

Quantum Race: Two operations, X and Y form a quantum race if and only if they form a race, either X or Y is a quantum atomic, and the other is not a quantum atomic.

Program and Model Definitions:

DRFrlx Program: A program is DRFrlx if and only if for every SC execution of its quantum-equivalent program:

- all operations can be distinguished by the system as either data or as paired, unpaired, commutative, non-ordering, or quantum atomics, and
- there are no data races, commutative races, non-ordering races, or quantum races in the execution.

DRFrlx Model: A system obeys the DRFrlx memory model if and only if the result of every execution E of a DRFrlx program P on the system is the same as the result of an SC execution Eq of the quantum-equivalent program Pq of P. In addition, E must obey **happens-before consistency** and **per-location SC**.

Using Quantum Atomics in RefCounter

Reference Counter [154]: Quantum atomics can also be used for some reference counters. The reference counter example in Listing 3.7 has shared global counters that are incremented and decremented by multiple threads to track the number of threads accessing shared objects. The

constructor increments the shared reference counters to signify that a thread is now accessing the shared objects; similarly, the destructor decrements the counters to signify that it is no longer accessing the object. If this thread is the last thread accessing the shared counter, then the thread marks the object to be deleted because it is the last thread accessing it. Later, after some synchronization (e.g., a global barrier), a thread will check if the object has been marked for deletion and delete it (not shown).[9]

Since multiple threads concurrently increment and decrement the reference counters, the operations need to be distinguished as atomics. Although relaxing the counter accesses can cause SC violations, like **Split Counter**, **Reference Counter** can tolerate some SC violations. For example, it does not matter whether the accesses to the set of reference counters are sequentially consistent, as long as the final decrement to each counter marks the shared object to be freed. Because of this, **Reference Counter** implementations often trade SC semantics for improved performance by using relaxed atomics.

DRFrlx cannot use commutative atomics, because the decrements return and use the intermediate values. However, DRFrlx can use quantum atomics for the increments and decrements, as long as any potentially racy accesses that delete the object are protected by some non-relaxed synchronization (e.g., a global barrier). In the quantum-equivalent program, quantum increments may write an arbitrary value and quantum decrements may return (and write) an arbitrary value. Therefore, extra care must be taken to avoid race conditions in any possible quantum-equivalent SC execution. If races in the quantum-equivalent SC executions are avoided, then DRFrlx guarantees SC execution for all non-quantum accesses and per-location SC and hb-consistency for all quantum accesses. Although this constraint can limit the use of quantum atomics, the resulting guarantees are stronger than those provided for relaxed atomics in existing consistency models.

3.3.6 Distinguishing Memory Operations

DRFrlx requires a mechanism in the programming language for distinguishing data operations from atomics, and for distinguishing paired, unpaired, commutative, non-ordering, quantum, and speculative atomics from one another. We reuse the C++ mechanism that DRF0 already uses to

[9]This example differs slightly from Sutter's [154] in order to emphasize the benefits of relaxation in a multi-counter context.

distinguish data and atomics. To distinguish the different types of atomics, we introduce five new keywords, unpaired, commutative, non-ordering, quantum, and speculative, to allow programmers to identify which type of relaxed atomics they are using (analogous to how C and C++ specify relaxed atomics). In practice, for the last four categories, the distinctions are important only to enable reasoning about the correctness of the program. For system optimizations, all four can be merged into a single category of relaxed since they allow the same optimizations.

3.3.7 DRFrlx Systems

Based on the intuition developed in the previous sections, a heterogeneous system obeys DRFrlx if it is DRF1 compliant, it enforces happens-before consistency and per-location SC for atomics, and it constrains commutative, non-ordering, quantum, and speculative operation completion/propagation in the same way as data operations. A formal proof of correctness would follow the structure of DRF proofs in prior work [5]. The system used in the evaluation in Section 3.6 conforms to the above properties.

3.3.8 Mechanizing DRFrlx

We mechanize DRFrlx with Herd [14], a tool for axiomatically specifying and simulating memory models. Given a model definition and a program, Herd produces all possible executions of the program as constrained by the model, and flags any relations of interest as specified by the model (e.g., race conditions). Since Herd does not support reads/writes of random values, our model is only able to identify races in SC executions of the original program, not the quantum-equivalent program. Therefore it is not an exhaustive exploration, and some manual inspection is necessary when quantum atomics are used. Additionally, Herd does not have a built-in way to determine if the value returned by a memory operation is observable by any other instruction in the thread. Therefore, for commutative and speculative atomics we approximate observability by defining it as any return value which (directly or indirectly) affects the address used by a future memory access, the value stored by a future memory access, or the path taken by a future branch. This is also imprecise and requires some manual inspection when using racy commutative and speculative accesses.

```
let at-least-one a = a*_ | _*a

let PairedR = (Paired & R)
let PairedW = (Paired & W)
let so1 = (PairedW * PairedR) & (rf | fr | co)+
let hb1 = (po | so1)+
let conflict = at-least-one W & loc
let race = (conflict & ext & ~(hb1 | hb1^-1)) \ (IW*_)
let data-race = race & (at-least-one Data)

(* comm-pair relates any two memory operations which are pairwise commutative omitted) *)
(* commutative race: a race involving a commutative access where either a) the accesses are
     not pairwise commutative *)
let comm-race1 = (race & (at-least-one Comm)) \ comm-pair
(* or b) the return value of an operation is observable *)
let comm-race2 = (race & (at-least-one Comm)) ; (addr | data | ctrl)
let comm-race = comm-race1 | comm-race2

(* pco: program-conflict order, pcoPO: pco that contains a po edge *)
let pco = (po | co | rf | fr)+
let pco-po = po | (po ; pco) | (pco ; po ; pco) | (pco ; po)

(* opath-aloNO: ordering path with at least one NO atomic *)
let aloNO = (at-least-one NonOrder)
let pcoPO-NO-pco = (pcoPO & aloNO) ; pco
let pco-NO-pcoPO = pco ; (pcoPO & aloNO)
let pcoPO-aloNO = (pcoPO & aloNO) | pcoPO-NO-pco | pco-NO-pcoPO
let opath-aloNO = pcoPO-aloNO & conflict

(* valid ordering path 1: accesses to the same address *)
let valid-pco1 = ((po | co | rf | fr) & loc)+
let valid-po1 = po & loc
let valid-pcoPO1 = valid-po1 | (valid-po1 ; valid-pco1) | (valid-pco1 ; valid-po1 ; valid-
    pco1) | (valid-pco1 ; valid-po1)
let valid-opath1 = valid-pcoPO1 & conflict

(* valid ordering path 2: Unpaired/Paired accesses *)
let valid-pco2 = ((po | co | rf | fr) & (Paired | Unpaired)*(Paired | Unpaired))+
let valid-po2 = po & (Paired | Unpaired)*(Paired | Unpaired)
let valid-pcoPO2 = valid-po2 | (valid-po2 ; valid-pco2) | (valid-pco2 ; valid-po2 ; valid-
    pco2) | (valid-pco2 ; valid-po2)
let valid-opath2 = valid-pcoPO2 & conflict

(* non-ordering race: there is an ordering path between two accesses which contains a
    NonOrdering edge, and there are no alternate valid ordering paths *)
(* note: for simpler herd construction, this relation is defined between the accesses at
    the ends of the ordering path *)
let non-order-race = ((race \ data-race \ comm-race) & opath-aloNO) \ valid-opath1 \ valid-
    opath2

(* quantum race: Quantum races with non-quantum *)
let quantum-race = (race & (at-least-one Quantum)) \ (Quantum * Quantum)

(* speculative race: race involving speculative access where a) both accesses are writes *)
let speculative-race1 = (race & (at-least-one Spec) & (W * W))
(* ... or b) the racy load is observable *)
let speculative-race2 = (race & (at-least-one Spec)) ; (addr | data | ctrl)
let speculative-race = speculative-race1 | speculative-race2

let illegal-race = data-race | comm-race | non-order-race | quantum-race | speculative-race

acyclic (po | rf | co | fr) (* limit to SC executions *)
empty rmw & (fre ; coe) (* RMWs to happen atomically *)

flag ~empty (illegal-race) as IllegalRace (* Identify any races in SC executions *)
```

Listing 3.8: DRFrlx's programmer-centric model in Herd: defining and identifying illegal races in a program.

```
let at-least-one a = a*_ | _*a

(* atom-pair relates any two atomic accesses (relaxed or not) *)
let per-loc-atomic = loc & (atom-pair) & (po | co | rf | fr)

(* Which po orderings are enforced in example system: accesses to the same address,
    successive unpaired/paired accesses, acq/rel fences*)
let inst-order = po & (loc | ((Acq | Rel | Unpaired) * (Acq | Rel | Unpaired)) | _*Rel |
    Acq*_)
(* control/data dependence requirement - don't speculatively issue stores *)
let control-order = (data | ctrl | addr)+

acyclic(per-loc-atomic) (* enforce per-location SC for atomics *)
(* make sure temporally enforced relations are acyclic *)
acyclic(inst-order | rf | fr | control-order)
empty rmw & (fre;coe) (* force RMWs to happen atomically *)

flag ~acyclic (po | rf | co | fr) as NonSC (* Identify any non-sc behavior *)
```

Listing 3.9: DRFrlx's system-centric model: Used to detect possible non-SC behavior in an example DRFrlx system

Our Herd evaluation consists of two models. Listing 3.8 shows the programmer-centric model, which defines and identifies illegal races under DRFrlx. Each illegal race type is specified using terms such as the program order, modification order, and reads-from relations in a dynamic execution. Given an example program, Herd generates all possible SC executions and determines whether any illegal race conditions exist in the generated executions. We also defined a system-centric model (Listing 3.9) which generates all possible executions in a straightforward example DRFrlx system. This model restricts program executions in a way that preserves intuitive atomic reordering invariants. For example, successive unpaired accesses must occur in program order, paired reads may not be reordered with subsequent memory accesses, and paired writes may not be reordered with prior memory accesses. Using this model one can determine whether a program can exhibit non-SC behavior on such a system.

We created numerous litmus tests to stress the models. These include the use cases in Table 3.1, incorrectly labeled versions of these use cases, and various other tests designed to stress various racy and non-racy patterns. For all litmus tests, the programmer-centric model correctly identifies races in the SC execution, and the system-centric model can only produce non-SC executions when the model allows it (i.e., when there is an illegal race or when quantum atomics are used). Furthermore, for programs without illegal races, even a non-SC execution is consistent with the DRFrlx guarantees of per-location SC for atomics, hb-consistency, and SC for non-quantum accesses.

Benefit	DRF0	DRF1 (if unpaired)	DRFrlx (if unpaired or relaxed)
Avoid cache invalidations at atomic loads	✗	✓	✓
Avoid store buffer flushes at atomic stores	✗	✓	✓
Overlap atomics in the memory system	✗	✗	✓

Table 3.2: Benefits of DRF0, DRF1, and DRFrlx.

3.4 Qualitative Analysis

In CPUs, the main benefit for relaxed atomics is overlapping relaxed atomics in the memory system. Unlike multi-core CPUs, heterogeneous systems largely use simple, software-based coherence protocols. As a result, relaxed atomics allow heterogeneous systems to reuse data across synchronization points by avoiding full cache invalidations on atomic loads and avoiding store buffer flushes on atomic stores. Table 3.2 qualitatively compares DRF0, DRF1, and DRFrlx.

DRF0 vs. DRF1: DRF0 treats all atomics as paired, so they cannot be overlapped, must invalidate all valid data on atomic loads, and must flush the store buffer on atomic stores. By distinguishing unpaired from paired atomics, DRF1 does not need to invalidate valid data or flush the store buffer on an unpaired atomic, which reduces overhead and improves valid data reuse compared to DRF0.

DRF1 vs. DRFrlx: Although DRF1 provides several benefits over DRF0, it does not allow atomics to be overlapped. DRFrlx improves performance and memory-level parallelism over DRF1 by allowing relaxed atomics to be overlapped in the memory system.

GPU coherence vs. DeNovoA: The choice of coherence protocol also affects performance. Since DeNovoA obtains ownership for written data and atomics, it can reuse them for all three consistency models. Obtaining ownership also allows DeNovoA's L1 MSHRs to locally coalesce multiple requests for the same address, which reduces network traffic, improves performance, and allows DeNovoA with DRFrlx to quickly service many overlapped atomic requests. However, obtaining ownership can hurt performance if an address is highly contended because DeNovoA may have to get ownership from a remote L1. Conversely, GPU coherence writes through all dirty data to the LLC on a store buffer flush. Thus, relaxed atomics are important because they allow GPU coherence to avoid flushing the store buffer. GPU coherence also performs all atomic operations at the LLC. As a result, it never needs to go to a remote core for ownership. Although this may help

for addresses where reuse is unlikely (e.g., highly contended or sparsely accessed addresses), GPU coherence also cannot coalesce multiple atomic requests for the same address. This exacerbates LLC contention for applications with large amounts of atomic parallelism.

3.5 Methodology

The simulations use a setup similar to Section 2.5.

3.5.1 Configurations

We evaluate all combinations of a traditional GPU and the DeNovoA coherence protocols with the DRF0, DRF1, and DRF2 memory models. Although state-of-the-art heterogeneous systems use the HRF memory model, we do not compare to HRF because only Flags and UTS would benefit from using local scopes.[10] The configurations we compare are:

GPU-DRF0 (GD0): Combines traditional GPU-style coherence, which performs all atomics at the L2, with the baseline DRF0 memory model.

GPU-DRF1 (GD1): Uses GPU coherence and the DRF1 memory model to distinguish between paired and unpaired atomics.

GPU-DRFrlx (GDR): *GDR* uses GPU coherence with the DRFrlx memory model, which allows it to overlap relaxed atomics.

DeNovoA-DRF0 (DD0): *DD0* uses DeNovoA coherence, which performs all atomic operations at the L1s and coalesces atomics at the L1 MSHRs, and the baseline DRF0 memory model.

DeNovoA-DRF1 (DD1): *DD1* combines DeNovoA coherence with the DRF1 memory model.

DeNovoA-DRFrlx (DDR): *DDR* uses DeNovoA coherence and the DRFrlx memory model.

3.5.2 Benchmarks

We evaluate the effectiveness of relaxed atomics on heterogeneous CPU-GPU systems with a mix of microbenchmarks (based on the examples discussed in Section 3.3) and benchmarks, summarized in Table 3.3. For all benchmarks, we found that that the CUDA compiler would put as much

[10]Section 2.7 showed that *DD* provides similar performance for UTS. Flags uses LFTRB_LG for its barrier, and *DD* provides better performance than *GH* for LFTRB_LG.

Benchmark	Input	Atomic Types
Microbenchmarks		
Flags[154]	90 TBs	Commutative, Non-Ordering
Histogram (H)[113, 124]	64 TBs, 256 KB, 256 bins	Commutative
Histogram_global (HG)[124]	64 TBs, 256 KB, 256 bins	Commutative
Histogram_global-2K (HG-2K)[124]	64 TBs, 256 KB, 2K bins	Commutative
Histogram_global-Non-Order (HG-NO)	64 TBs, 256 KB, 256 bins	Non-Ordering
Multiple Locks (ML)[64]	512 TBs	Speculative
RefCounter (RC)[154]	64 TBs	Quantum
Seqlocks (SEQ)[33]	512 TBs	Speculative
SplitCounter (SPC)[111]	112 TBs	Quantum
Benchmarks		
UTS[77, 119]	16K nodes	Unpaired
BC[42]	rome99 (1), nasa1824 (2), ex33 (3), c-22 (4)	Commutative, Non-Ordering
PageRank (PR)[42]	c-37 (1), c-36 (2), ex3 (3), c-40 (4)	Commutative

Table 3.3: Benchmarks, input sizes, and relaxed atomics used.

independent computation as possible between atomics. Although this optimization makes sense for current GPUs, where atomics are infrequent and slow, because they must be performed at the LLC, it also prevents some relaxed atomics from being overlapped. Thus, we wrote hand-optimized assembly to increase the overlap of relaxed atomics by grouping atomics together through loop peeling and loop unrolling. Moreover, for some applications (e.g., PageRank [74]) we used loop invariant code motion to optimize the baseline performance of the algorithms.

The microbenchmarks represent the use cases we obtained from developers. Historically, relaxed atomics are necessary to obtain high performance for these applications. We designed these microbenchmarks to stress the benefit that relaxed atomics could provide from overlapping atomics in the memory system – although relaxed atomics can also benefit from reusing data, the microbenchmarks have very few global data operations. We use a histogram [124] for the Event Counters example, and created several variants to highlight different types of access patterns. In Hist (H), each thread locally bins its values in the scratchpad before updating the shared global histogram once all its data has been binned. To model high contention, Hist_global (HG) performs all updates on the shared global histogram instead of locally binning its values first. To explore the impact of varying the number of histogram bins, we also created a version of HG with 2048 bins: HG-2K. Unlike H, HG, and HG-2K, HG-Non-Order (HG-NO) reads the final values of the histogram bins, like the bottom of Listing 3.2. To examine how this part of the Event Counter performs, we do not include the update portion (i.e., the HG portion) in its results. We wrote the remainder of the microbenchmarks based on the code listings in Section 3.3.

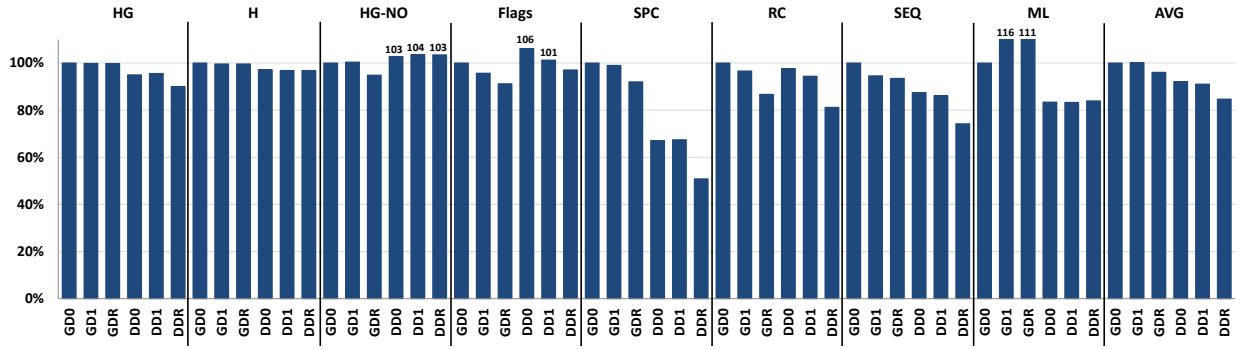
For the full benchmarks, we first identified which (standard) GPGPU benchmarks [24, 31, 38, 42, 43, 44, 51, 55, 60, 61, 68, 69, 70, 74, 116, 118, 119, 137, 145, 146, 150] use atomics and categorized them, focusing on the benchmarks that use relaxed atomics.[11] We use the two benchmarks from Figure 3.1 that obtain the highest max speedups: BC and PageRank. In addition, we chose UTS, which is representative of future workloads that perform dynamic load balancing. Unlike the microbenchmarks, these benchmarks benefit from overlapping relaxed atomics, reusing data that would invalidated by SC atomics, and avoiding store buffer flushes. UTS uses unpaired atomics, similar to the Work Queue example, while BC and PageRank use commutative and non-ordering atomics. For BC and PageRank, we studied 33 Matrix Market graphs [53] and 10^{th} DIMACS challenge graph suites [22, 23]; we show results for four representative graphs.

3.6 Results

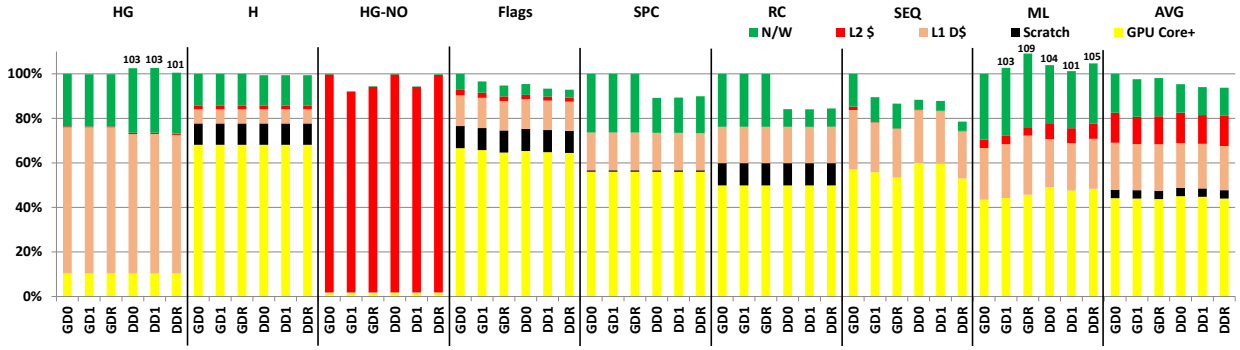
Figures 3.3 and 3.4 show results (normalized to the *GD0* configuration) for the microbenchmarks and benchmarks, respectively, for all 6 configurations (Section 3.5.1). Parts (a), (b), and (c) of the figures show the execution time, energy consumption, and network traffic, respectively. As in Section 2.7, energy is divided into multiple components based on the source of energy: GPU core+, scratchpad, L1, L2, and network. Similarly, network traffic is again measured in flit crossings and is also divided into multiple components: data loads, data registrations (stores), writebacks/writethroughs, and atomics.

The experiments show mixed results for the effectiveness of DRF1 and DRFrlx over DRF0. For the microbenchmarks, DRF1 and DRFrlx provide small benefits: on average, DRFrlx reduces execution time by 4% for GPU coherence and 9% for *DeNovoA*; DRF1's average benefits are negligible. Of the microbenchmarks, relaxed atomics help the most for SPC, RC, and SEQ: up to 13% reduction in execution time for GPU coherence and 25% for *DeNovoA*, compared to DRF0. For BC and PR, the benefits of DRF1 are higher, depending on the graph (up to 49% for *GD1* and 61% for *DD1* compared to *GD0* and *DD0*, respectively). DRFrlx further reduces execution time for several BC and PR graphs (up to 29% for *DDR* and 37% for *GDR* compared to *DD1* and *GD1*, respectively). In most cases, *DeNovoA*'s ability to reuse data and atomics also improves energy

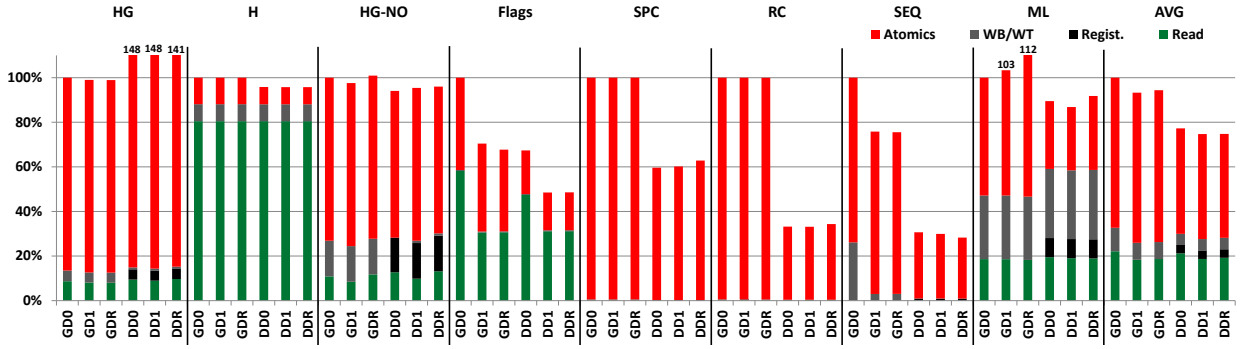
[11] Of the 227 benchmarks and use cases we examined, 47 (21%) of them used relaxed atomics.



(a) Execution time



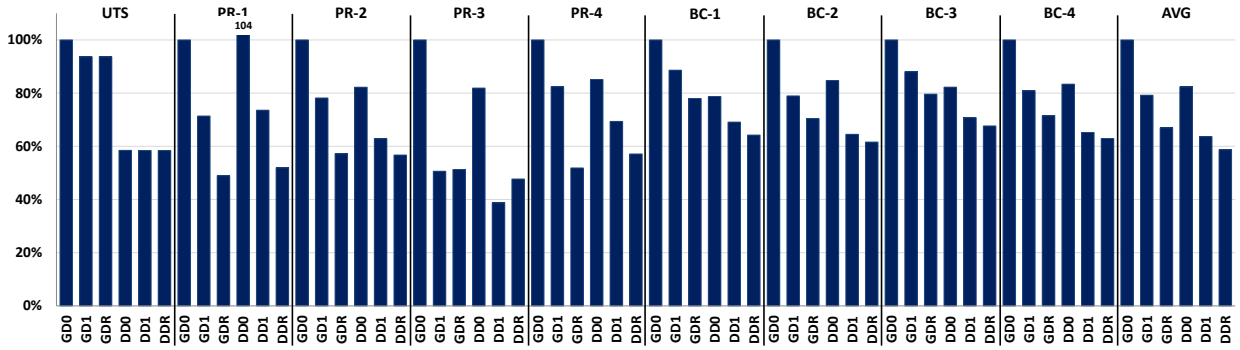
(b) Dynamic energy



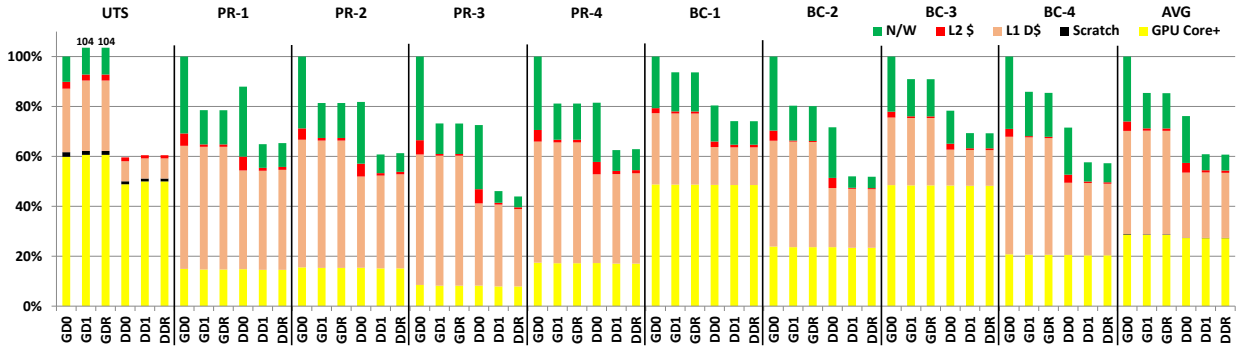
(c) Network traffic

Figure 3.3: Results for all microbenchmarks, normalized to *GD0*.

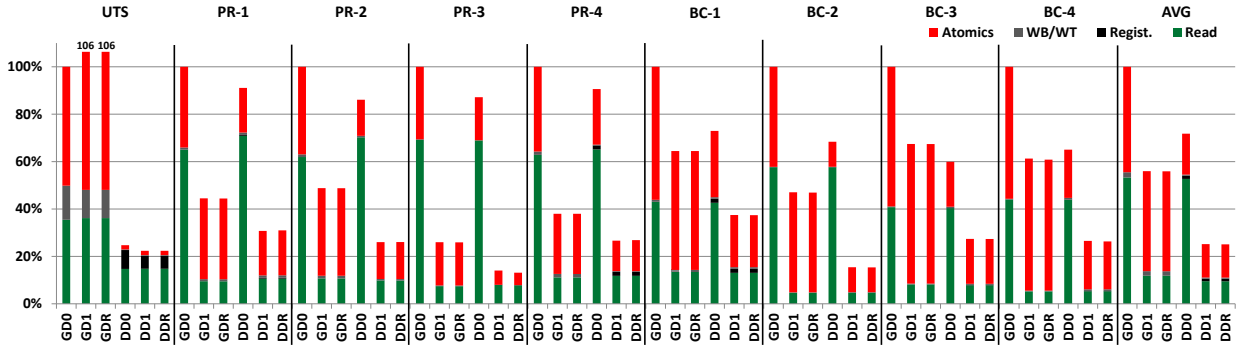
compared to GPU. However, accessing data remotely sometimes increases *DeNovoA*'s energy (e.g., HG). Comparing the interaction between the different protocols and consistency models, we find (as also shown in Chapter 2) that *DD0* generally provides improved or comparable performance relative to *GD0*, except for HG-NO, Flags, and PR-1. As we weaken the memory models (with DRF1 and DRFrlx), the gap between *DeNovoA* and GPU coherence stays roughly the same. On average, *DeNovoA* reduces execution time by 13% for DRF0, 14% for DRF1, and 11% for DRFrlx, energy by 15%, 17%, and 17%, and network traffic by 26%, 37%, and 38%.



(a) Execution time



(b) Dynamic energy

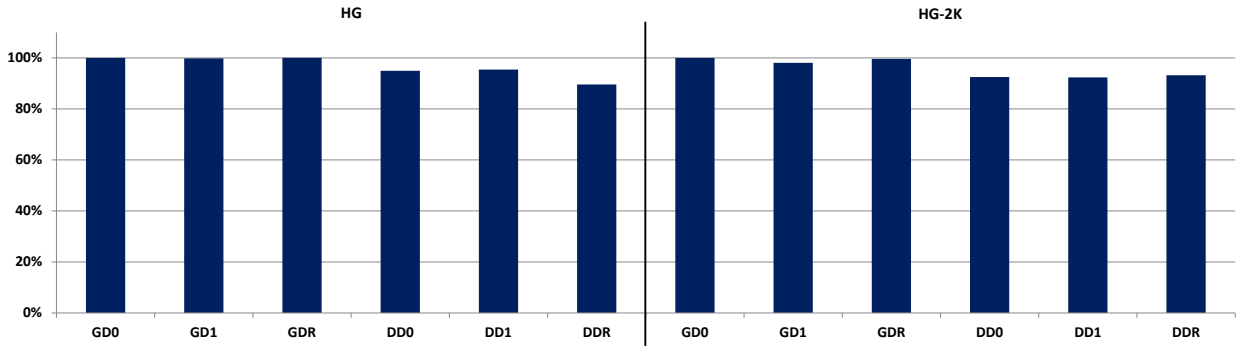


(c) Network traffic

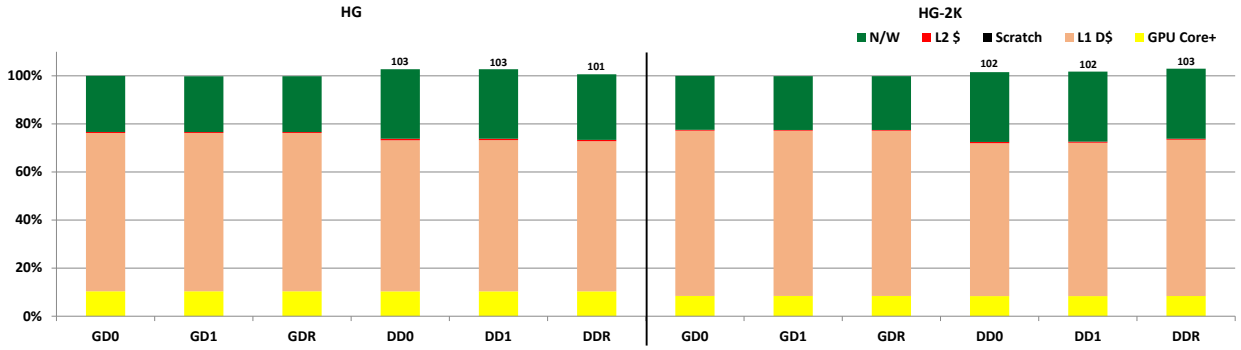
Figure 3.4: Results for all benchmarks, normalized to *GD0*.

3.6.1 Varying Number of Histogram Bins

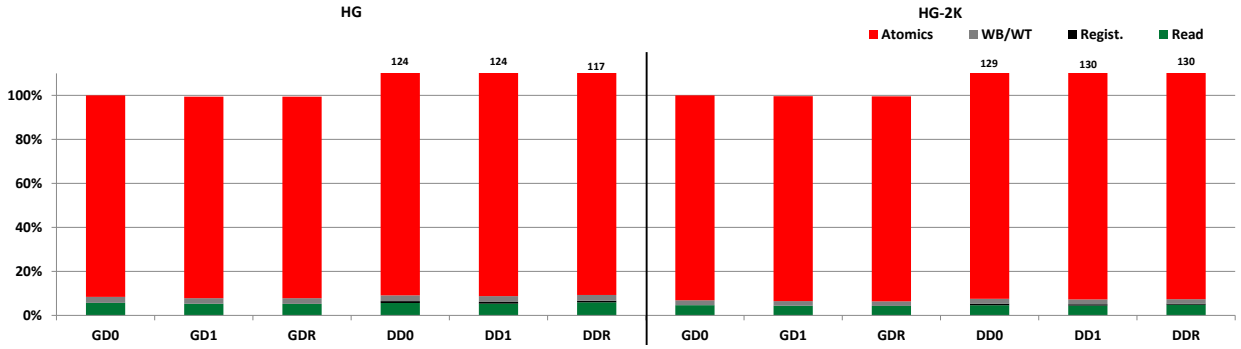
We examined different levels of contention and number of bins for the histogram applications. Figures 3.5 and 3.6 show how the execution time (cycles), energy, and network traffic are affected by increasing the number of histogram bins from 256 (HG) to 2048 (HG-2K). Increasing the number of bins reduces contention, decreases execution time (Figure 3.6a, it does not affect the performance trends when normalized to *GD0* (Figure 3.5a). Increasing the number of bins also does not significantly affect the normalized energy (Figure 3.5b) and network traffic (Figure 3.5c) trends,



(a) Execution time



(b) Dynamic energy

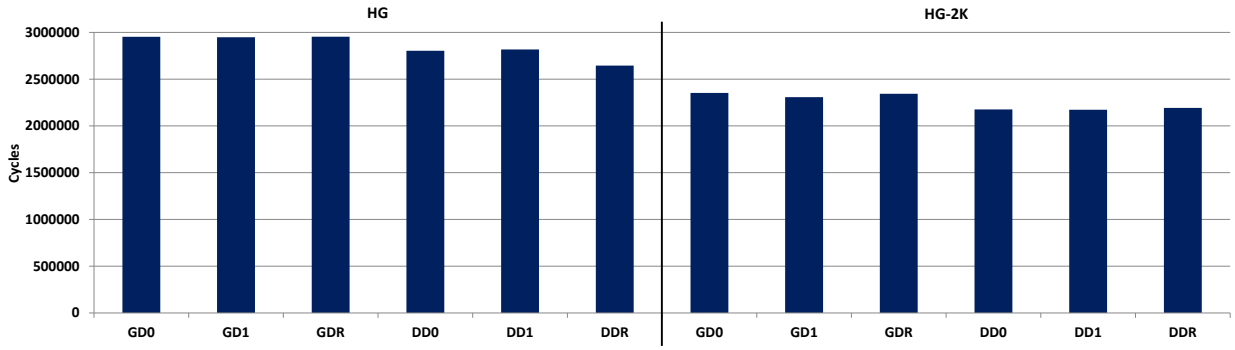


(c) Network traffic

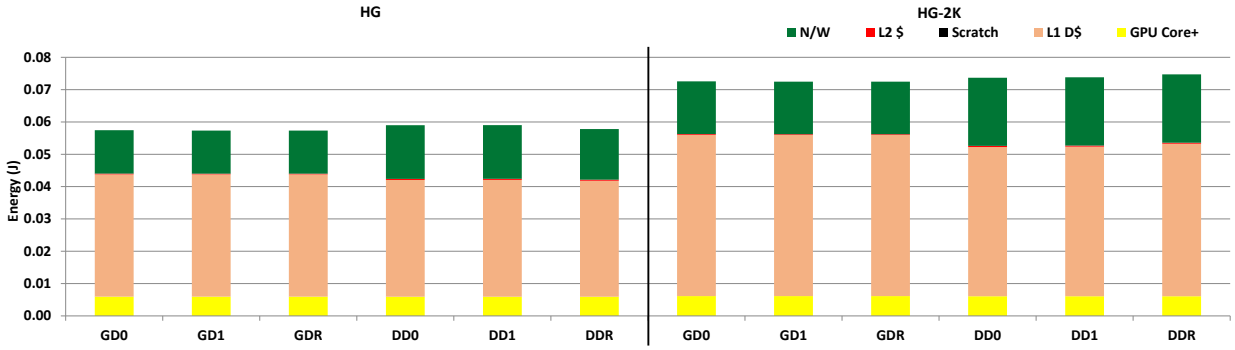
Figure 3.5: Results for varying number of histogram bins, normalized to *GD0*.

although it does increase the overall energy (Figure 3.6b) and network traffic (Figure 3.6c) due to increased at the L1 contention.

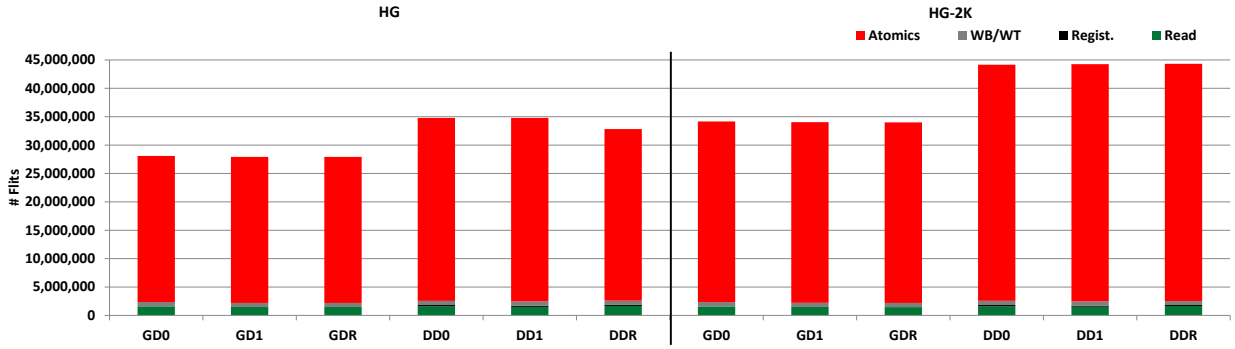
Since increasing the number of histogram bins does not affect the trends, we focus on HG in the remainder of this subsection.



(a) Execution time



(b) Dynamic energy



(c) Network traffic

Figure 3.6: Absolute results for varying number of histogram bins.

3.6.2 DRF0 vs. DRF1

DRF1's unpaired atomics can improve performance by avoiding the store buffer flushes and self-invalidations associated with paired synchronization read and write semantics. Fewer invalidations and flushes facilitates better cache reuse. However, since the microbenchmarks are designed to highlight the benefits of full relaxation (DRFrlx) and exhibit little data reuse, DRF1 has little impact on them. One notable exception is ML, where the unpaired atomics increase contention by increasing the rate the locks are checked, similar to the effect UTS sees (see below).

Unlike the microbenchmarks, the full-sized benchmarks often benefit from DRF1. BC and PageRank benefit the most from DRF1 because they have frequent relaxed atomics and high levels of data reuse – avoiding cache invalidations in DRF1 increases their data reuse compared to DRF0. On average, DRF1 improves BC's performance by 18% for *DeNovoA* (16% for GPU) and energy by 17% for *DeNovoA* (12% for GPU). Avoiding cache invalidations improves performance for UTS with GPU coherence (by 6% relative to *GD0*) by increasing both the rate the queue is polled and reuse of written data. UTS with *DeNovoA* with DRF1 does not see this same benefit because *DeNovoA* always obtains ownership for written and atomic data. The relaxed atomics in UTS also increase energy: the DRF0 ordering constraints regulate the rate at which atomics are issued to load a polled value. Removing these constraints in DRF1 leads to more atomic accesses that increase energy. Across all the benchmarks and microbenchmarks, DRF1 improves *DeNovoA*'s (GPU coherence's) performance by 11% (10%), energy by 11% (9%), and network traffic by 34% (26%) on average.

3.6.3 DRF1 vs. DRFrlx

DRFrlx allows relaxed atomics to be overlapped in the memory system, which increases memory-level parallelism over DRF1. All microbenchmarks except H and HG-NO with *DDR* see some benefit from this, although the benefit is sometimes small due to increased contention (e.g., ML). Since H locally bins its data before updating the global histogram, it has few atomics to overlap, while HG-NO with *DDR* suffers from the overhead of obtaining ownership from a remote core. Conversely, obtaining ownership for atomics enables *DeNovoA* to reuse them. As a result, *DDR* reduces SPC, RC, and SEQ's execution time by 25%, 14%, and 14%, respectively, compared to *DD1*.

As expected, DRFrlx does not affect UTS's execution time, because UTS uses unpaired atomics. However, BC and PR see benefits from DRFrlx (up to 29% reduction in execution time for *DDR* and 37% for *GDR* compared to *DD1* and *GD1*, respectively). PR benefits more than BC because it does not have as many control and data dependencies as BC, although in PR-3 the added contention increases execution time. In general, DRFrlx does not improve energy because the overhead from the increased memory contention cancels out the additional reuse benefits. On average, across all

the benchmarks and microbenchmarks, DRFrlx reduces *DeNovoA*'s (GPU coherence's) execution time by 7% (10%) and provides the same energy efficiency and network traffic as DRF1.

3.6.4 DeNovoA vs. GPU Coherence

Obtaining ownership for written data and atomics allows *DeNovoA* to reuse them regardless of consistency model. Normally this is beneficial, but in some cases the overhead of accessing data remotely increases execution time (PR-1, HG-NO, Flags) and energy (HG). However, obtaining ownership usually helps and on average *DD0* reduces execution time by 13%, energy by 15%, and network traffic by 26% compared to *GD0*.

DRF1 allows reuse of valid data across unpaired atomics and avoids excessive store buffer flushes. Increased reuse helps *GD1* for all of the full-sized benchmarks, especially BC, which has lots of potential data reuse. However, GPU coherence cannot reuse atomics, which is why *DD1* still outperforms *GD1*. On average *DD1* reduces execution time by 14%, energy by 17%, and network traffic by 37% compared to *GD1*.

By overlapping the relaxed atomics in the memory system, *GDR* is able to hide the latency of performing the atomics at the L2. This helps GPU coherence overcome its inability to reuse atomics and provide similar performance to *DeNovoA* with DRFrlx for some benchmarks. However, in many other cases (PR-3, BC 1-4, HG, SPC, RC, SEQ), *DeNovoA* provides additional benefits with DRFrlx by coalescing atomics in the L1 MSHR, which filters requests, reduces traffic, and allows *DeNovoA* to support a higher atomic access bandwidth. On average *DDR* reduces execution time by 11%, energy by 17%, and network traffic by 38% over *GDR*.

Overall, across all workloads, DRF1 improves *DeNovoA*'s performance by 11% (10% for GPU coherence), energy by 11% (9%), and network traffic by 34% (26%), on average. DRFrlx's improved memory-level parallelism further improves performance by 7% for *DeNovoA* (10% for GPU). Although *DeNovoA*'s indirection overhead sometimes hurts, for most workloads the improved cache reuse improves performance (on average 11%), energy (17%), and network traffic (38%) for *DDR* compared to *GDR*.

3.7 Summary

As we discussed in Section 2.8, tighter integration of accelerators requires better support for memory consistency. In the previous chapter, we showed how to retain the standard, less complex DRF memory consistency model for heterogeneous systems. Although DRF0 provides high performance for most applications, in certain applications (e.g., graph analytics workloads like BC and PageRank) programmers want the ability to relaxed DRF0's strict constraints on atomics to improve performance and energy.

Relaxed atomics can be reordered with all other memory operations, so they can significantly improve performance and energy. Moreover, since SC atomics are far more expensive in accelerators like GPUs than they are in CPUs, using relaxed atomics is even more tempting. Unfortunately, relaxed atomics are also extremely difficult to use correctly because languages such as C, C++, HSA, and OpenCL have no formal semantics for them. Despite more than a decade of research, no acceptable semantics for relaxed atomics have been found.

Unlike previous work, which tries to formalize the semantics by prohibiting "out-of-thin-air" executions, we focus on how developers want to use relaxed atomics in heterogeneous systems. After examining numerous GPGPU benchmarks and reaching out to vendors, developers, and researchers, we identified five relaxed atomic use cases: unpaired, commutative, non-ordering, quantum, and speculative. Next, we designed a new memory model, DRFrlx, that extends DRF0 and DRF1 to provide SC-centric semantics for these use cases. Thus, DRFrlx resolves a long-standing open problem and makes it easier for programmers to use relaxed atomics correctly while retaining their efficiency benefits.

We also evaluate relaxed atomics in heterogeneous CPU-GPU systems for these use cases. Compared to DRF0, we find that DRF1 and DRFrlx provide small benefits for all benchmarks except SplitCounter, RefCounter, Seqlocks, BC, and PageRank; BC and PageRank benefit significantly from DRF1 (up to 61% execution time reduction) and see additional benefits from DRFrlx (up to 37% execution time reduction compared to DRF1). The results also show that the DeNovoA coherence protocol outperforms a conventional GPU coherence protocol, regardless of memory consistency model: on average DeNovoA reduces execution time over GPU coherence by 13%, 14%, and 11%, energy by 15%, 17%, and 17%, and network traffic by 26%, 37%, and 38% for DRF0,

DRF1, and DRFrlx, respectively.

Chapter 4

Integrating Specialized Memories Into the Unified Address Space

4.1 Motivation

Another source of inefficiency in the memory hierarchy of current heterogeneous systems are the memories. Software-managed scratchpads and hardware-managed caches are two widely used memory organizations in heterogeneous SoCs. However, both have inefficiencies that make it harder to build an energy efficient memory hierarchy for heterogeneous systems.

Caches are a popular organization unit for their ease of use and software transparency. However, they have several inefficiencies. The **indirect, hardware-managed addressing** of caches requires TLB lookups and tag comparisons for every single access (hit or miss), incurring overheads. Caches also have unpredictable hit rates, causing performance anomalies. Caches also have **poor storage efficiency** since data is organized at fixed cache line granularity.

Scratchpads are software-managed and directly addressable. Thus, unlike caches they do not have overheads from TLB lookups, tag comparisons, or conflict misses. They also provide a fixed access latency (100% hit rate) and compact storage because they only bring in useful data. However, unlike caches, scratchpads use a private address space, so they are **not globally addressable**. As a result, they require explicit data movement between the global address space and the scratchpad's private address space, which negates some of the benefits of using scratchpads and pollutes the core's L1 cache and registers. Scratchpads also do not perform well for applications with on-demand loads because today's scratchpads preload all elements. Furthermore, scratchpads are **not globally visible**. Consequently, dirty data needs to be explicitly written back to the global address space by the end of the GPU kernel, which precludes any inter-kernel data reuse.

To mitigate the inefficiencies of caches and scratchpads, we introduce ***stash***. The stash combines the best properties of the cache and scratchpad organizations (as shown in Table 4.1). Like a

Feature	Benefit	Cache	Scratchpad	Stash
Directly addressed	No address translation hardware access	✗ (if physically tagged)	✓	✓ (on hits)
	No tag access	✗	✓	✓
	No conflict misses	✗	✓	✓
Compact storage	Efficient use of SRAM storage	✗	✓	✓
Global addressing	Implicit data movement from/to structure No pollution of other memories On-demand loads into structures	✓	✗	✓
Global visibility	Lazy writebacks to global address space (AS) Reuse across application phases	✓	✗	✓

Table 4.1: Comparison of cache, scratchpad, and stash.

scratchpad, the stash is software managed and is **directly addressable**. It also has the benefits of **compact storage** by bringing in only the needed data. Like a cache, the stash can directly **access the global address space** using a mapping (provided by the software and maintained in the hardware) from a contiguous set of stash addresses to a possibly non-contiguous set of global addresses. Global addressability also helps eliminate other scratchpad inefficiencies such as pollution of the L1 cache and registers and the need for always preloading data. Because the coherence protocol makes stash data **globally visible**, the stash can employ lazy writebacks for dirty data and reuse data across kernels. However, this requires extensions to the coherence protocol; in Section 4.5.4, we discuss how to extend coherence protocols for the stash. Our implementation uses the DeNovoA coherence protocol, because it is simpler and has lower overhead than the alternatives.

Our results (Section 4.7) show that the stash provides better performance and energy efficiency than either caches or scratchpads. Moreover, stash enables new use cases and demonstrates how to integrate private, specialized memory into the coherent, unified, global address space. Although there has been a significant amount of prior work on optimizing the behavior of private memories, none of them mitigate all of the inefficiencies of both scratchpads and caches like the stash does. Qualitatively comparing to all of the prior work is outside the scope of this thesis, so we provide a detailed qualitative comparison to all of them in Section 6.3 and quantitatively compare the stash to the closest technique: a scratchpad enhanced with a DMA engine [30, 80].

4.2 Background: Memory Organizations

In this section we show how caches and scratchpads are used in current heterogeneous systems and discuss their advantages and disadvantages.

4.2.1 Cache

Caches are a common memory organization in modern systems. Their software transparency makes them easy to program, but caches have several inefficiencies:

Indirect, hardware-managed addressing: Cache loads and stores specify addresses that hardware must translate to determine the physical location of the accessed data. This *indirect addressing* implies that each cache access (a hit or a miss) incurs (energy) overhead for TLB lookups and tag comparisons. Virtually tagged caches do not require TLB lookups on hits, but they incur additional overhead, including dealing with synonyms, page mapping and protection changes, and cache coherence [26]. Furthermore, the indirect, hardware-managed addressing also results in unpredictable hit rates due to cache conflicts, causing pathological performance (and energy) anomalies, a notorious problem for real-time systems.

Inefficient, cache line based storage: Caches store data at fixed cache line granularities which wastes SRAM space when a program does not access the entire cache line (e.g., when a program phase traverses an array of large objects but accesses only one field in each object).

4.2.2 Scratchpad

Scratchpads (referred to as shared memory in CUDA) are local memories that are managed in software, either by the programmer or through compiler support. Unlike caches, scratchpads are directly addressed so they do not have overhead from TLB lookups or tag comparisons (this provides significant savings: 34% area and 40% power [25] or more [99]). Direct addressing also eliminates the pathologies of conflict misses and has a fixed access latency (100% hit rate). Scratchpads also provide compact storage since the software only brings useful data into the scratchpad. These features make scratchpads appealing, especially for real-time systems [21, 156, 157]. However, scratchpads also have some inefficiencies:

Not Globally Addressable: Scratchpads use a separate address space disjoint from the global address space, with no hardware mapping between the two. Thus, extra instructions must *explicitly* move such data between the two spaces, incurring performance and energy overhead. Furthermore, in current systems the additional loads and stores typically move data via the core's L1 cache and its registers, *polluting* these resources and potentially evicting (spilling) useful data. Scratchpads

```
func_scratch(struct* aosA, int myOffset, int myLen)
{
  __scratch__ int local[myLen];
  // explicit global load and scratchpad store
  parallel for(int i = 0; i < myLen; i++) {
    local[i] = aosA[myOffset + i].fieldX;
  }
  // do computation(s) with local(s)
  parallel for(int i = 0; i < myLen; i++) {
    local[i] = compute(local[i]);
  }
  // explicit scratchpad load and global store
  parallel for(int i = 0; i < myLen; i++) {
    aosA[myOffset + i].fieldX = local[i];
  }
}
```

```
func_stash(struct* aosA, int myOffset, int myLen)
{
  __stash__ int local[myLen];
  //AddMap(stashBase, globalBase, fieldSize,
           objectSize, rowSize, strideSize,
           numStrides, isCoherent)
    AddMap(local[0], aosA[myOffset], sizeof(int),
         sizeof(struct), myLen, 0, 1, true);


  // do computation(s) with local(s)
  parallel for(int i = 0; i < myLen; i++) {
    local[i] = compute(local[i]);

  }
}
```

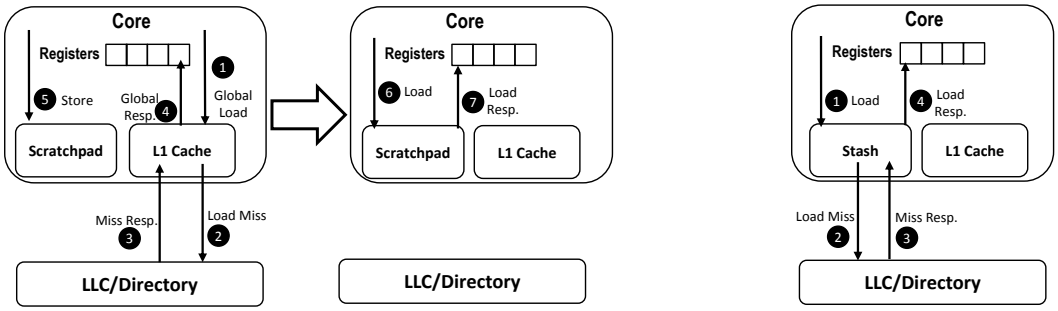


Figure 4.1: Codes and hardware events to copy data from the corresponding global address for (a) scratchpad and (b) stash (events to write data back to the global address are not shown).

also do not perform well for applications with *on-demand loads* because today's scratchpads preload all elements. In applications with control/data dependent accesses, only a few of the preloaded elements will be accessed.

Not Globally Visible: A scratchpad is visible only to its local CU. Thus dirty data in the scratchpad must be explicitly written back to the corresponding global address before it can be used by other CUs. Further, all global copies in the scratchpad must be freshly loaded if they could have been written by other CUs. In typical GPU programs, there is no data sharing within a kernel;[1] therefore, the writebacks must complete and the scratchpad space deallocated by the end of the kernel. With more fine-grained sharing, these actions are needed at the more frequent fine-grained synchronization phases. Thus, the scratchpads' lack of global visibility incurs potentially unnecessary *eager writebacks* and precludes *reuse* of data across multiple application phases (e.g., multiple GPU kernels).

[1] A kernel is the granularity at which the CPU invokes the GPU and it executes to completion on the GPU.

4.2.3 Example and Usage Modes

Figure 4.1a shows how scratchpads are used. The code at the top reads one field, $fieldX$ (of potentially many), from an array of structs (AoS) data structure, $aosA$, into an explicit scratchpad copy. It performs some computations using the scratchpad copy and then writes back the result to $aosA$. The bottom of Figure 4.1a shows some of the corresponding steps in hardware. First, the hardware must explicitly copy $fieldX$ into the scratchpad. To achieve this, the application issues an *explicit load* of the corresponding global address to the L1 cache (event 1). On an L1 miss, the hardware brings $fieldX$'s cache line into the L1, *polluting* it as a result (events 2 and 3). Next, the hardware sends the data value from the L1 to the core's register (event 4). Finally, the application issues an *explicit store* instruction to write the value from the register into the corresponding scratchpad address (event 5). At this point, the scratchpad has a copy of $fieldX$ and the application can finally access the data (events 6 and 7). Once the application is done modifying the data, the dirty scratchpad data is *explicitly written back* to the global address space, requiring loads from the scratchpad and stores into the cache (not shown in the figure).

We refer to this scratchpad usage mode, where data is moved explicitly from/to the global space, as the *global-unmapped* mode. Scratchpads can also be used in *temporary* mode for private, temporary values. Temporary values do not require global address loads or writebacks because they are discarded after their use (they trigger only events 6 and 7).

4.3 Stash Overview

The stash is a new SRAM organization that combines the advantages of scratchpads and caches, as summarized in Table 4.1. Stash has the following features:

Directly addressable: Like scratchpads, a stash is directly addressable and data in the stash is explicitly allocated by software (either the programmer or the compiler).

Compact storage: Since it is software managed, only data that is accessed is brought into the stash. Thus, like scratchpads, stash enjoys the benefit of a compact storage layout, and unlike caches, it only stores useful words from a cache line.

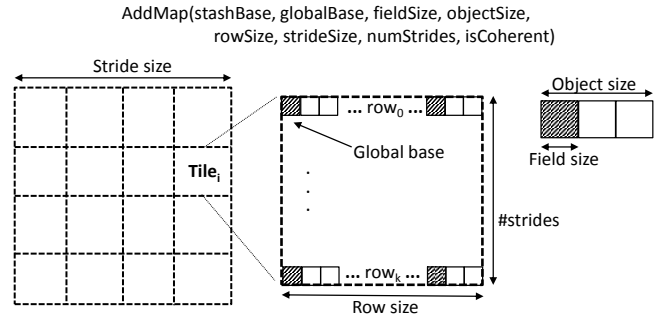
Physical to global address mapping: In addition to being able to generate a direct, physical

stash address, software also specifies a mapping from a contiguous set of stash addresses to a (possibly non-contiguous) set of global addresses. This architecture can map to a 1D or 2D, possibly strided, tile of global addresses.Hardware maintains the mapping between the stash and global space.

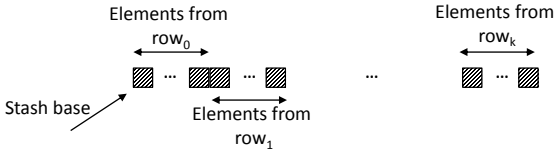
Global visibility: Like a cache, stash data is globally visible through a coherence mechanism (described in Section 4.5.4). A stash, therefore, does not need to eagerly writeback dirty data. Instead, data can be reused and lazily written back only when the stash space is needed for a new allocation (similar to cache replacements). If another CU needs the data, it will be forwarded through the coherence mechanism. In contrast, for scratchpads in current GPUs, data is written back to global memory (and flushed) at the end of a kernel, resulting in potentially unnecessary and bursty writebacks with no reuse across kernels.

The first time a load occurs to a newly mapped stash address, it implicitly copies the data from the mapped global space to the stash (analogous to a cache miss). Subsequent loads for that address immediately return the data from the stash (analogous to a cache hit, but with the energy benefits of direct addressing). Similarly, no explicit stores are needed to write back the stash data to its mapped global location. Thus, the stash enjoys all the benefits of direct addressing of a scratchpad on hits (which occur on all but the first access), without the overhead incurred by the additional loads and stores that scratchpads require for explicit data movement.

Figure 4.1b transforms the code from Figure 4.1a for a stash. The stash is directly addressable and stores data compactly just like a scratchpad but does not have any explicit instructions for moving data between the stash and the global address space. Instead, the stash has an *AddMap* call that specifies the mapping between the two address spaces (discussed further in Section 4.4). In hardware (bottom part of the figure), the first load to a stash location (event 1) implicitly triggers a global load (event 2) if the data is not already present in the stash. Once the load returns the desired data to the stash (event 3), it is sent to the core (event 4). Subsequent accesses directly return the stash data without consulting the global mapping.



(a) 2D tiles in global address space (left), a 2D tile (center), and an object (right).



(b) 1D stash space with one of the fields of the AoS.

Figure 4.2: Mapping a global 2D AoS tile to a 1D stash address space.

4.4 Stash Software Interface

We envision the programmer or the compiler will map a (possibly non-contiguous) part of the global address space to the stash. For example, programmers writing applications for today's GPU scratchpads already effectively provide such a mapping. There has also been prior work on compiler methods to automatically infer this information [21, 88, 117]. The mapping of the global address space to stash requires strictly less work compared to that of a scratchpad as it avoids the need for explicit loads and stores between the global and stash address spaces.

4.4.1 Specifying Stash-to-Global Mapping

The mapping between global and stash address spaces is specified using two intrinsic functions. The first intrinsic, *AddMap*, is called to communicate a new mapping to the hardware. We need an *AddMap* call for every data structure (a linear array or a 2D tile of an AoS structure) that is mapped to the stash.

Figure 4.1b shows an example usage of *AddMap* along with its definition. Figures 4.2a and 4.2b respectively show an example 2D tiled data structure in the global address space and the mapping of one field of one of the 2D tiles in the 1D stash address space. The first two parameters of *AddMap* specify the stash and global virtual base addresses for the given tile, as shown in

Figure 4.2 (scratchpad base described in Section 4.5).

Figure 4.2a also shows the various parameters used to describe the object and the tile. The field size and the object size provide information about the global data structure (field size = object size for scalar arrays). The next three parameters specify information about the tile in the global address space: the row size of the tile, global stride between two rows of the tile, and number of strides. Finally, *isCoherent* specifies the operation mode of the stash (discussed in Section 4.4.3). Figure 4.2b shows the 1D mapping of the desired individual fields from the 2D global AoS data structure.

The second intrinsic function, $ChgMap$, is used when there is a change in mapping or the operation mode of a set of global addresses mapped to the stash. $ChgMap$ uses all of the $AddMap$ parameters and adds a field to identify the map entry it needs to change (an index in a hardware table, discussed in Section 4.5.1).

4.4.2 Stash Load and Store Instructions

The load and store instructions for a stash access are similar to those for a scratchpad. On a hit, the stash just needs to know the requested address. On a miss, in addition to the requested address, the stash also needs to know which stash-to-global mapping it needs to use (an index in a hardware table, discussed in Section 4.5.1, similar to that used by $ChgMap$ above). This information can be encoded in the instruction in at least two different ways without requiring extensions to the ISA. CUDA, for example, has multiple address modes for LD/ST instructions - register, register-plus-offset, and immediate addressing. The register-based addressing schemes hold the stash (or scratchpad) address in the register specified by the *register* field. We can use the higher bits of the register for storing the map index (since a stash address does not need all the bits of the register). Alternatively, we can use the register-plus-offset addressing scheme, where register holds the stash address and $offset$ holds the map index (in CUDA, offset is currently ignored when the local memory is configured as a scratchpad).

4.4.3 Usage Modes

Stash data can be used in four different modes:

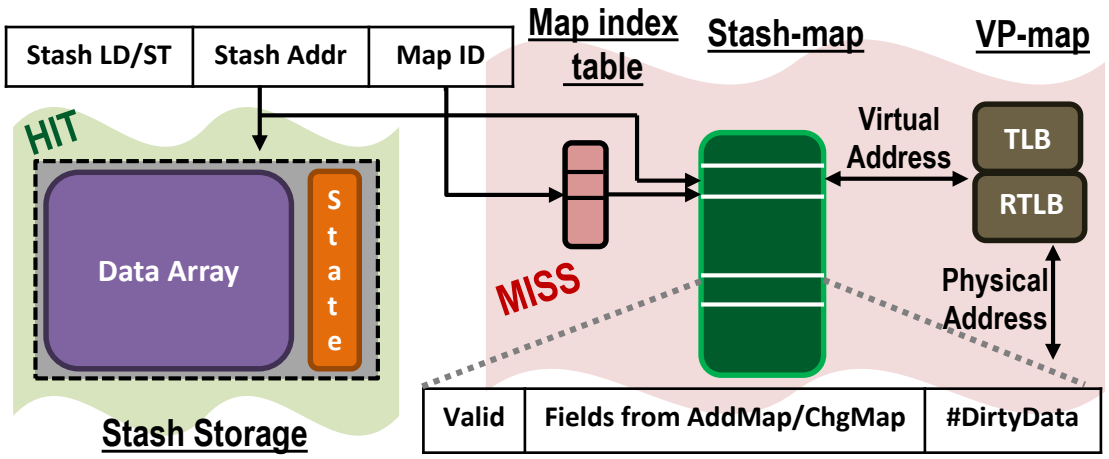


Figure 4.3: Stash hardware components.

- **Mapped Coherent**: This mode is based on Figure 4.1b – it provides a stash-to-global mapping and the stash data is globally addressable and visible.

- **Mapped Non-coherent**: This mode is similar to Mapped Coherent except that the stash data is not globally visible. As a result, any modifications to local stash data are not reflected in the global address space. We use the *isCoherent* bit to differentiate between the Mapped-Coherent and Mapped Non-Coherent usage modes.

- **Global-unmapped and Temporary**: In *Global-unmapped*, data is moved explicitly from/to the global space. Scratchpads can also be used in *temporary* mode for private, temporary values. Unlike Mapped Coherent and Mapped Non-coherent, these modes do not need an AddMap call, since they do not have global mappings. These modes allow programs to fall back, if necessary, to how scratchpads are currently used and ensure all current scratchpad code is supported in the system.

4.5 Stash Hardware Design

This section describes the design of the stash hardware, which provides stash-to-global and global-to-stash address translations for misses, writebacks, and remote requests. On current GPUs, every TB gets a separate scratchpad allocation. The runtime translates the program-specified scratchpad address to a physical location in the TB's allocated space. We assume a similar allocation and

runtime address mapping mechanism for the stash.

4.5.1 Stash Components

Figure 4.3 shows the stash's four hardware components: (1) *stash storage*, (2) *map index table*, (3) *stash-map*, and (4) *VP-map*. This section describes each component. Section 4.5.2 describes how they enable the different stash operations.

Stash Storage

The stash storage component provides data storage for the stash. It also contains state bits to identify hits and misses (depending on the coherence protocol) and to aid writebacks (explained in Section 4.5.2).

Map Index Table

The per TB map index table provides an index into the TB's stash-map entries. Each $AddMap$ allocates an entry into the map index table. Assuming a fixed ordering of $AddMap$ calls, the compiler can determine which table entry corresponds to a mapping – it includes this entry's ID in future stash instructions corresponding to this mapping (using the format from Section 4.4). The size of the table is the maximum number of $AddMap$'s allowed per TB (the design allocates up to four entries per TB). If the compiler runs out of entries, it cannot map any more data to the stash.

Stash-map

The stash-map contains an entry for each mapped stash data partition, shown in Figure 4.3. Each entry contains information to translate between the stash and global virtual address spaces, determined by the fields in $AddMap$ or $ChgMap$. We precompute most of the information required for the translations at the $AddMap$ and $ChgMap$ call and do not need to store all the fields from these calls. With the precomputed information, only six arithmetic operations are required per miss (details in [89]). In addition to information for address translation, the stash-map entry has a $Valid$ bit (denotes if the entry is valid) and a $\#DirtyData$ field used for writebacks.

We implemented the stash-map as a circular buffer with a tail pointer. We add and remove

entries to the stash-map in the same order for easy management of stash-map's fixed capacity. The number of entries should be at least the maximum number of TBs a core can execute in parallel multiplied by the maximum number of *AddMap* calls allowed per TB. We found that applications did not simultaneously use more than four map entries per TB. Assuming up to eight TBs in parallel, 32 map entries are sufficient but we use 64 entries to allow additional lazy writebacks.

VP-map

We need the virtual-to-physical translations for each page of a stash-to-global mapping because every mapping can span multiple virtual pages. VP-map uses two structures for this purpose. The first structure, TLB, provides a virtual to physical translation for every mapped page, required for stash misses and writebacks. We can leverage the core's TLB for this purpose. For remote requests which come with a physical address, we need a reverse translation. The second structure, $RTLB$, provides the reverse translation and is implemented as a CAM over physical pages.

Each entry in the VP-map has a pointer (not shown in Figure 4.3) to a stash-map entry that indicates the latest stash-map entry that requires the given translation. When a stash-map entry is replaced, any entries in the VP-map that have a pointer to that map entry are no longer needed. We remove these entries by walking the $RTLB$ or TLB. By keeping each $RTLB$ entry (and each TLB entry, if kept separate from system TLB) around until the last mapping that uses it is removed, no misses occur in the $RTLB$ (see Section 4.5.2). We assume that the number of virtual-to-physical translations required by all active mappings is less than the size of the VP-map (for the applications we studied, 64 entries were sufficient to support all the TBs running in parallel) and that the compiler or programmer is aware of this requirement.

4.5.2 Operations

Next we describe how the stash operations are implemented.

Hit: On a hit (determined by coherence bits as discussed in Section 4.5.4), the stash acts like a scratchpad, accessing only the storage component.

Miss: A miss needs to translate the stash address into a global physical address. Stash uses the index to the map index table provided by the instruction to determine its stash-map entry. Given

the stash address and the stash base from the stash-map entry, we can calculate the stash offset. Using the stash offset and the other fields of the stash-map entry, we can calculate the virtual offset (details in [89]). Once we have the virtual offset, we add it to the virtual base of the stash-map entry to obtain the missing global virtual address. Finally, using the VP-map we can determine the corresponding physical address which is used to handle the miss.

Additionally, a miss must consider if the data it replaces needs to be written back and a store miss must perform some bookkeeping to facilitate a future writeback. We describe these actions next.

Lazy Writebacks: Stash writebacks (only for *Mapped Coherent entries*) happen lazily; i.e., the writebacks are triggered only when the space is needed by a future stash allocation similar to cache evictions.

On a store, we need to maintain the index of the current stash-map entry for a future writeback. Naively we could store the stash-map entry's index per word and write back each word as needed but this is not efficient. Instead, we store the index at a larger, chunked granularity, say 64B, and perform writebacks at this granularity.[2] To know when to update this per chunk stash-map index, we have a dirty bit per stash chunk. On a store miss, if this dirty bit is not set, we set it and update the stash-map index. We also update the $\#DirtyData$ counter of the stash-map entry to track the number of dirty stash chunks in the corresponding stash space. The per chunk dirty bits are unset when the TB completes so that they are ready for use by a future TB using the same stash chunk.

Later, if a new mapping needs to use the same stash location, the old dirty data needs to be written back. We (conceptually) use a writeback bit per chunk to indicate the need for a writeback (Section 4.5.5 discusses using bits already present for coherence states for this purpose). This bit is set for all the dirty stash chunks at the end of a TB and checked on each access to trigger any needed writeback. To perform a writeback, we use the per chunk stash-map index to access the stash-map entry – similar to a miss, we determine the dirty data's physical address. we write back all dirty words in a chunk on a writeback (we leverage per word coherence state to determine the dirty words). On a writeback, the $\#DirtyData$ counter of the map entry is decremented and the

[2]This requires data structures in the memory to be aligned at the chosen chunk granularity.

writeback bit of the stash chunk is reset. When the $\#DirtyData$ counter reaches zero, the map entry is marked as invalid.

AddMap: An $AddMap$ call advances the stash-map's tail, and sets the next entry of its TB's map index table to point to this tail entry. It updates the stash-map tail entry with its parameters, does the needed precomputations for future address translations, and sets the $Valid$ bit (Section 4.5.1). It also invalidates any entries from the VP-map that have the new stash-map tail as the back pointer.

For every virtual page mapped, an entry is added to the VP-map's $RTLB$ (and the TLB, if maintained separately from the system's TLB). If the system TLB has the physical translation for this page, we populate the corresponding entries in VP-map (both in $RTLB$ and TLB). If the translation does not exist in the TLB, the physical translation is acquired at the subsequent stash miss. For every virtual page in a new map entry, the stash-map pointer in the corresponding entries in VP-map is updated to point to the new map entry. In the unlikely scenario where the VP-map becomes full and has no more space for new entries, we evict subsequent stash-map entries (using the procedure described here) until there are enough VP-map entries available. The VP-map is sized to ensure that this is always possible. This process guarantees that the RTLB never misses for remote requests.

If the stash-map entry being replaced was previously valid ($Valid$ bit set), then it indicates an old mapping has dirty data that has not yet been (lazily) written back. To ensure that the old mapping's data is written back before the entry is reused, DeNovoA initiates its writebacks and block the core until they are done. Alternately, a scout pointer can stay a few entries ahead of the tail, triggering non-blocking writebacks for valid stash-map entries. This case is rare because usually a new mapping has already reclaimed the stash space held by the old mapping, writing back the old dirty data on replacement.

ChgMap: $ChgMap$ updates an existing stash-map entry with new mapping information. If the mapping points to a new set of global addresses, we need to issue writebacks for any dirty data of the old mapping (only if *Mapped Coherent*) and mark all the remapped stash locations as invalid (using the coherence state bits in Section 4.5.4). A $ChgMap$ can also change the usage mode for the same chunk of global addresses. If an entry is changed from coherent to non-coherent, then

```
/* Values that can be precomputed */
//stashBytesPerRow = (rowSize / objectSize) * fieldSize
//virtualToStashRatio = strideSize / stashBytesPerRow
//objectToFieldRatio = objectSize / fieldSize

/* stashBase is obtained from the stash-map entry */
stashOffset = stashAddress - stashBase

fullRows = stashOffset * virtualToStashRatio
lastRow = (stashOffset % stashBytesPerRow) * objectToFieldRatio
virtualOffset = fullRows + lastRow

/* virtualBase is obtained from the stash-map entry */
virtualAddress = virtualBase + virtualOffset
```

Listing 4.1: Translating stash address to virtual address

we need to issue writebacks for the old mapping because the old mapping's stores are globally visible. However, if the entry is modified from non-coherent to coherent, then we need to issue ownership/registration requests for all dirty words in the new mapping according to the coherence protocol (Section 4.5.4).

4.5.3 Address Translation

Listing 4.1 shows the logic for translating a stash offset to its corresponding global virtual offset. The stash offset is obtained by subtracting a stash address (provided with the instruction) from the stash base found in the stash-map entry. As shown in the translation logic, we do not need to explicitly store all the parameters of an *AddMap* call in the hardware. We can pre-compute information required for the translations. For example, for stash to virtual translation, we can precompute three values. First, we need the number of bytes in stash that a given row in the global space corresponds to. This is equal to the number of bytes the shaded fields in a given row amount to in Figure 4.2a. This value is stored in *stashBytesPerRow*. Next to account for the gap between the two global rows, we need to know the global span of *stashBytesPerRow*. So we calculate the ratio of *strideSize* to *stashBytesPerRow*. This value is stored in *virtualToStashRatio*. *virtualToStashRatio* helps us find the corresponding global row for a given stash address. Finally, to calculate the relative position of the global address within the identified row, we need the ratio of object size to field size, stored in *objectToFieldRatio*. Using these pre-computed values, we can get the corresponding virtual span (virtual offset) for all the full rows of the tile the stash offset spans and for the partial last row (if any). This virtual offset is added to the virtual base found

```
/* Values that can be precomputed */
//stashBytesPerRow = (rowSize / objectSize) * fieldSize
//stashToVirtualRatio = stashBytesPerRow / strideSize
//fieldToObjectRatio = fieldSize / objectSize

/* virtualBase is obtained from the stash-map entry */
virtualOffset = virtualAddress - virtualBase

fullRows = virtualOffset * stashToVirtualRatio
lastRow = (virtualOffset % rowSize) * fieldToObjectRatio
stashOffset = fullRows + lastRow

/* stashBase is obtained from the stash-map entry */
stashAddress = stashBase + stashOffset
```

Listing 4.2: Translating virtual address to stash address

in the stash-map entry to get the corresponding virtual address. Overall, we need six arithmetic operations per miss.

The logic for the reverse translation of global address to stash address is similar and is shown in Listing 4.2.

4.5.4 Coherence Protocol Extensions for Stash

All *Mapped Coherent* stash data must be kept coherent. We can extend any coherence protocol to provide this support (e.g., a traditional hardware coherence protocol such as MESI or a software-driven hardware coherence protocol like DeNovoA from Section 2.3) as long as it supports the following three features:

1. Tracking at word granularity: Stash data must be tracked at word granularity because only useful words from a given cache line are brought into the stash.[3]

2. Merging partial cache lines: When the stash sends data to a cache (either as a writeback or a remote miss response), it may send only part of a cache line. Thus the cache must be able to merge partial cache lines.

3. Map index for physical-to-stash mapping: When data is modified by a stash, the directory needs to record the modifying core (as usual) and also the stash-map index for that data (so a remote request can determine where to obtain the data).

It is unclear which is the best underlying coherence protocol to extend for the stash since

[3]DeNovoA can support byte granularity accesses if all (stash-allocated) bytes in a word are accessed by the same CU at the same time; i.e., there are no word-level data races. None of the benchmarks we studied have byte granularity accesses.

protocols for tightly coupled heterogeneous systems are evolving rapidly and represent a moving target [71, 125]. Below we discuss how the above features can be incorporated within traditional directory-based hardware protocols and DeNovoA. For the evaluations, without loss of generality, we choose the latter since it incurs lower overhead, is simpler, and is closer to current GPU memory coherence strategies (e.g., it relies on cache self-invalidations rather than writer-initiated invalidations and it does not use directories).

Traditional protocols: We can support the above features in a traditional single-writer directory protocol (e.g., MESI) with minimal overhead by retaining coherence state at line granularity, but adding a bit per word to indicate whether its up-to-date copy is present in a cache or a stash. Assuming a shared last level cache (LLC), when a directory receives a stash store miss request, it transitions to modified state for that line, sets the above bit in the requested word, and stores the stash-map index (obtained with the miss request) in the data field for the word at the LLC. This straightforward extension, however, is susceptible to false-sharing (similar to single-writer protocols for caches) and the stash may lose the predictability of a guaranteed hit after an initial load. To avoid false-sharing, we could use a sector-based cache with word-sized sectors, but this incurs heavy overhead with conventional hardware protocols (state bits and sharers list per word at the directory).

Sectored software-driven protocols: DeNovo [46, 152, 153], which DeNovoA extends for heterogeneous systems, is a software-driven hardware coherence protocol that has word granularity sectors (coherence state is at word granularity, but tags are at conventional line granularity) and naturally does not suffer from false-sharing.

DeNovoA exploits software-inserted self-invalidations at synchronization points to eliminate directory overhead for tracking the sharers list. Moreover, DeNovoA requires fewer coherence state bits because it has no transient states. This combination allows DeNovoA's total state bits overhead to be competitive with (and more scalable than) line-based MESI [46].

Table 4.2 summarizes the additional storage overhead required to support the stash for variants of the MESI protocol (including a word-based MESI protocol with coherence and tag bits at the word granularity) and the DeNovoA protocol.

We extended the line-based DeNovoA protocol from [46] (with line granularity tags and word

Protocol	Tag Overhead	Coherence State Overhead	Sharer's List Overhead	Stash map index Overhead
MESI line	T	5	C	16
MESI word	16 * T	16 * 5 = 80	16 * C	0
MESI line with 16 sectors	T	16 * 5 = 80	16 * C	0
DeNovoA line	T	2 + 16 = 18 [46]	0	0

Table 4.2: Coherence storage overhead (in bits) required at the directory (per line) to support stash for various protocols. These calculations assume 64 byte cache lines with 4 byte words, C = number of cores, T = tag bits, and five state bits for MESI. We assume that the map information at the LLC can reuse the LLC data array (the first bit indicates if the data is a stash-map index or not, the rest of the bits hold the index).

granularity coherence, but does not use DeNovo's regions) to incorporate the stash. This protocol was originally proposed for multi-core CPUs and deterministic applications. Later versions of DeNovo support non-deterministic codes [152, 153], but the applications are deterministic. Although GPUs support non-determinism through operations such as atomics, these are typically resolved at the shared cache and are trivially coherent. We assume that software does not allow concurrent conflicting accesses to the same address in both cache and stash of a given core within a kernel. The protocol requires the following extensions to support stash operations:

Stores: Stores miss when in $Valid$ or $Invalid$ state and hit when in *Registered* state. In addition to registering the core ID at the directory, registration requests for words in the stash also need to include the corresponding stash-map index. In addition to storing the registered core ID in the data array of the LLC, the stash-map index can also be stored, and its presence indicated using a bit in the same LLC data word. Thus, the LLC continues to incur no additional storage overhead for keeping track of remotely modified data.

Self-invalidations: At the end of a kernel DeNovoA keeps the data that is registered by the core (specified by the coherence state) but self-invalidates the rest of the entries to make the stash space ready for any future new allocations. In contrast, a scratchpad invalidates all entries (after explicitly writing the data back to the global address space).

Remote requests: Remote requests for stash that are redirected via the directory come with a physical address and a stash-map index (stored at the directory during the request for registration). Using the physical address, VP-map provides the corresponding virtual address. Using the stash-map index, DeNovoA can obtain all the mapping information from the corresponding stash-map entry. We use the virtual base address from the entry and virtual address from the VP-map to

calculate the virtual offset. Once we have the virtual offset, we use the map-entry's other fields to calculate the stash offset and add it to the stash base to get the stash address.

4.5.5 State Bits Overhead for Stash Storage

With the above DeNovoA protocol, we compute the total state bits overhead in the stash storage component (Section 4.5.1) as follows. Even without the stash, each word (4B) needs 2 bits for the protocol state. Each stash chunk needs 6 additional bits for the stash-map index (assuming a 64 entry stash-map). Although Section 4.5.2 discussed a separate per-chunk writeback bit, since DeNovoA has only 3 states, we can use the extra state in its two coherence bits to indicate the need for a writeback.[4] Assuming 64B chunk granularity, all of the above sum to 39 bits per stash chunk, with a ∼8% overhead to the stash storage. Although this overhead might appear to be of the same magnitude as that of tags of conventional caches, only the two coherence state bits are accessed on hits (the common case).

4.5.6 Stash Optimization: Data Replication

It is possible for two allocations in stash space to be mapped to the same global address space. This can happen if the same read-only data is simultaneously mapped by several TBs in a CU, or if read-write data mapped in a previous kernel is mapped again in a later kernel on the same CU. By detecting this replication and copying replicated data between stash mappings, it is possible to avoid costly requests to the directory.

To detect data replication, we can make the map a CAM, searchable on the virtual base address. On an *AddMap* (an infrequent operation), the map is searched for the virtual base address of the entry being added to the map. If there is a match, we compare the tile specific parameters to confirm if the two mappings indeed perfectly match. If there is a match, we set a bit, *reuseBit*, and add a pointer to the old mapping in the new map entry. On a load miss, if the reuseBit is set, we first check the corresponding stash location of the old mapping and copy the value over if present. If not, we issue a miss to the registry.

If the new map entry is non-coherent and both the old and new map entries are for the same

[4]Stash does not utilize read-only region information, so we reuse this state bit.

CPU Parameters	
Frequency	2 GHz
Cores (microbenchmarks, apps)	15, 1
GPU Parameters	
Frequency	700 MHz
CUs (microbenchmarks, apps)	1, 15
Scratchpad/Stash Size	16 KB
Number of Banks in Stash/Scratchpad	32
Memory Hierarchy Parameters	
TLB & RTLB (VP-map)	64 entries each
Stash-map	64 entries
Stash address translation	10 cycles
L1 and Stash hit latency	1 cycle
Remote L1 and Stash hit latency	35−83 cycles
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
L2 hit latency	29−61 cycles
Memory latency	197−261 cycles

Table 4.3: Stash-specific parameters of the simulated heterogeneous system.

allocation in the stash, we need to writeback the old data. Instead, if the new map entry is coherent and both the old and new map entries are for different allocations in the stash, we need to send new registration requests for the new map entry.

4.6 Methodology

We extend the system described in Section 2.5 to add the stash. The stash is located at the same level as the GPU L1 caches and both the cache and stash write their data to the backing L2 cache bank. We also extended GPUWattch [97] to measure the energy of the GPU CUs and the memory hierarchy including all stash components. To model the stash storage we extended GPUWattch's scratchpad model by adding additional state bits. We model the stash-map as an SRAM structure and the VP-map as a CAM unit. Finally, we model the operations for the address translations by adding an ALU for each operation using an in-built ALU model in GPUWattch. We use 64 entry TLBs, RTLBs, and Stash-maps and 10 cycles for the stash address translation. The sizes for these hardware components are listed in Table 4.3.

4.6.1 Simulated Memory Configurations

The baseline architecture has scratchpads as described in Section 2.5.1. To evaluate the stash, we replaced the scratchpads with stashes, following the design in Section 4.5 (we used the DeNovoA

protocol and included the data replication optimization). We also compare stash to a cache-only configuration (i.e., all data allocated as global and accessed from the cache).

Additionally, we compare the stash to scratchpads enhanced with a DMA engine. Our DMA implementation is based on the D^{2}MA design [80]. D^{2}MA provides DMA capability for scratchpad loads on discrete GPUs and supports strided DMA mappings. D^{2}MA adds special load instructions and a hardware prefetch engine to preload all scratchpad words. Unlike D^{2}MA, our implementation blocks memory requests at a core granularity instead of a warp granularity, supports DMAs for stores in addition to loads, and runs on a tightly-coupled system. We conservatively do not charge additional energy for the DMA engine that issues the requests.

The DMA optimization enables scratchpads to prefetch their data. To evaluate the effect of software-guided hardware prefetching, we applied the same optimization to the stash. Unlike DMA for scratchpads, stash prefetching does not block the core because stash accesses are globally visible and duplicate requests are handled by the MSHR. We conservatively do not charge additional energy for the DMA or the prefetch engine that issues the requests.

Overall we evaluate the following configurations:

Scratch: 16 KB Scratchpad + 32 KB L1 Cache. All memory accesses use the type specified by the original application.

ScratchG: *Scratch* with all global accesses converted to scratchpad accesses.

ScratchGD: *ScratchG* configuration with DMA support.

Cache: 32 KB L1 Cache with all scratchpad accesses in the original application converted to global accesses.[5]

Stash: 16 KB Stash + 32 KB L1 Cache. The scratchpad accesses in the *Scratch* configuration were converted to stash accesses.

StashG: ***Stash*** with all global accesses converted to stash accesses.

StashGP: *StashG* configuration with prefetching support.

[5]Because the *Cache* configuration has 16 KB less SRAM than other configurations, we also examined a 64 KB cache (GEMS only allows power-of-2 caches). The 64 KB cache was better than the 32 KB cache but had worse execution time and energy consumption than *StashG* despite using more SRAM.

4.6.2 Conveying Information from Application

To convey information about when to perform self-invalidations, we used the same format as DeNovo [46]. Adding support for self-invalidating stash requests required adding an API call to convey the invalidation information from the software to the hardware at the end of a phase in the program.

Regions for self-invalidations: While a real system would assign regions with a compiler, for this study we created an API to manually instrument the program with this information for every allocated object.

Self-Invalidations: Self-invalidations are performed at the end of each phase for all data in the cache that is not *touched* or *registered*. Stash data is marked to be evicted when its TB completes execution; since all data in the stash must be touched or registered, we do not need to perform self-invalidations on it.

Communication space: To convey communication granularity information, we use another special API call that controls the communication region table of the simulator. On a load miss, the table is checked to determine the space of the requesting word, which we extended to take into account that desired words may be in a cache or stash

Adding and Changing Map Entries: We added API calls for $AddMap$ and $ChgMap$ and manually instrumented the programs to convey mappings between stash and global addresses from the software to the hardware. The call populates an entry in the map for the core. In a real system, we expect that the compiler would be able to determine the mapping information by determining which addresses would map to the scratchpad and send the mapping information to the hardware via a special assembly instruction.

Scratchpad Requests

We modified how stash requests are coalesced so that they are coalesced based on the cache line they would be accessing in global memory. [6] This potentially hurts performance compared to the current GPU coalescing scheme, because data that may lie in different banks in the scratchpad and thus be allowed to be coalesced on modern GPUs may not lie on the same cache line in

[6] In comparison, scratchpad requests do not belong to a cache line in global memory, since they are brought into the scratchpad via the core's resources.

global memory (and thus can't be coalesced). To help offset this, in a given cycle we continue to perform accesses to the stash as long as those requests hit. This mimics the behavior of the current scratchpad implementations and reduces the overhead of our scheme. Additionally, we combine stash requests from the same half-warp based on the cache line each word would access in global memory.[7]

4.6.3 Workloads

We present results for a set of benchmark applications to evaluate the effectiveness of the stash design on existing code. However, these existing applications are tuned for execution on a GPU with current scratchpad designs that do not efficiently support data reuse, control/data dependent memory accesses, and accessing specific fields from an AoS format. As a result, modern GPU applications typically do not use these features. However stash is a forward looking memory organization designed both to improve current applications and increase the use cases that can benefit from using scratchpads. Thus, to demonstrate the benefits of the stash, we also evaluate it for microbenchmarks designed to show future use cases.

Microbenchmarks

We evaluate four microbenchmarks: Implicit, Pollution, On-demand, and Reuse. Each microbenchmark is designed to emphasize a different benefit of the stash design. All four microbenchmarks use an array of elements in AoS format; each element in the array is a struct with multiple fields. The GPU kernels access a subset of the structure's fields; the same fields are subsequently accessed by the CPU to demonstrate how the CPU cores and GPU CUs communicate data that is mapped to the stash. We use a single GPU CU for all microbenchmarks. We parallelize the CPU code across 15 CPU cores to prevent the CPU accesses from dominating execution time. The details of each microbenchmark are discussed below and in Table 4.4.

Implicit highlights the benefits of the stash's implicit loads and lazy writebacks as highlighted in Table 4.1. In this microbenchmark, the stash maps one field from each element in an array of structures. The GPU kernel updates this field from each array element. The CPUs then access

[7] In the scratchpad how many requests in the half-warp can proceed simultaneously is done by checking for bank conflicts amongst the threads in the half-warp.

Application	Input Size	Stash Usage Modes	
		# Mapped Coherent	# Mapped Non-coherent
Backprop [43, 44]	32 KB	2	4
LUD [43, 44]	256x256 matrix	4	3
NW [43, 44]	512x512 matrix	2	2
Pathfinder [43, 44]	10 x 100K matrix	1	4
SGEMM [145]	A: 128x96, B: 96x160	1	2
Stencil [145]	128x128x4, 4 iterations	1	3
SURF [31]	66 KB image	1	1

Table 4.4: Input sizes and stash usage modes of applications.

this updated data.

Pollution highlights the ability of the stash to avoid cache pollution through its use of implicit loads that bypass the cache. Pollution's kernel reads and writes one field in two AoS arrays A and B; A is mapped to the stash or scratchpad while B uses the cache. A is sized to prevent reuse in the stash in order to demonstrate the benefits the stash obtains by not polluting the cache. B can fit inside the cache only without pollution from A. Both stash and DMA achieve reuse of B in the cache because they do not pollute the cache with explicit loads and stores.

On-demand highlights the on-demand nature of stash data transfer and is representative of an application with fine-grained sharing or irregular accesses. The On-demand kernel reads and writes only one element out of 32, based on a runtime condition. Scratchpad configurations (including ScratchGD) must conservatively load and store every element that may be accessed. Cache and stash, however, can identify a miss and generate a memory request only when necessary.

Reuse highlights the stash's data compaction and global visibility and addressability. This microbenchmark repeatedly invokes a kernel which accesses a single field from each element of a data array. The relevant fields of the data array can fit in the stash but not in the cache because it is compactly stored in the stash. Thus, each subsequent kernel can reuse data that has been loaded into the stash by a previous kernel and lazily written back. In contrast, the scratchpad configurations (including ScratchGD) are unable to exploit reuse because the scratchpad is not globally visible. Cache cannot reuse data because it is not able to compact data.

Hardware Unit	Hit Energy	Miss Energy
Scratchpad	55.3 pJ	–
Stash	55.4 pJ	86.8 pJ
L1 cache	177 pJ	197 pJ
TLB access	14.1 pJ	14.1 pJ[8]

Table 4.5: Per access energy for various hardware units.

Applications

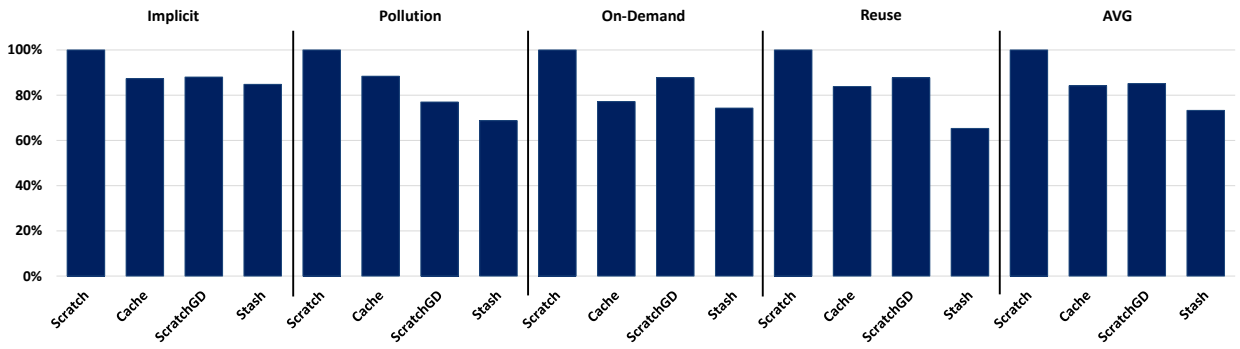
Table 4.4 lists the seven larger benchmark applications I use to evaluate the effectiveness of the stash. The applications are from Rodinia [43, 44], Parboil [145], and Computer Vision [31, 50]. All applications were selected for their use of the scratchpad. We manually modified the applications to use a unified shared memory address space (i.e., we removed all explicit copies between the CPU and GPU address spaces) and added the appropriate map calls based on the different stash modes of operation (from Section 4.4.3). The types of mappings used in each application (for all kernels combined) are listed in Table 4.4. We use only a single CPU core and do not parallelize these applications because they perform very little work on the CPU.

4.7 Results

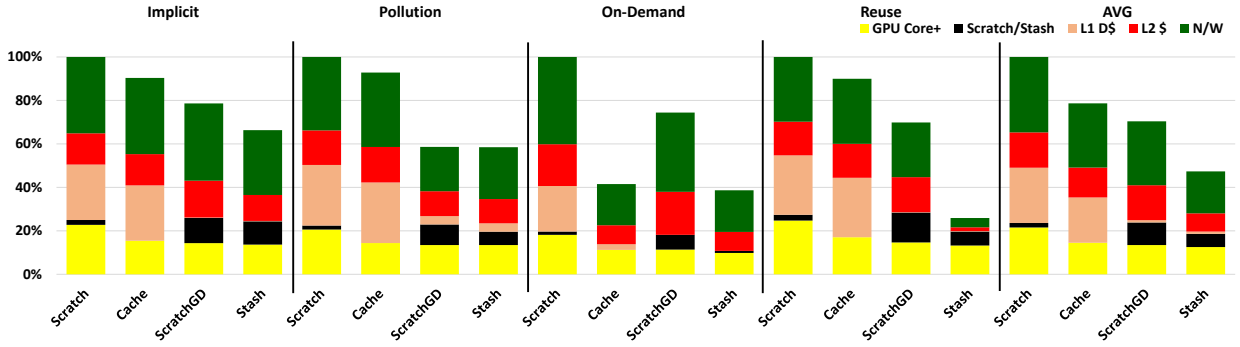
In this section, we compare the access energy for various hardware components, and examine how the stash performs compared to the other configurations, as discussed in Section 4.6.1.

4.7.1 Access Energy Comparisons

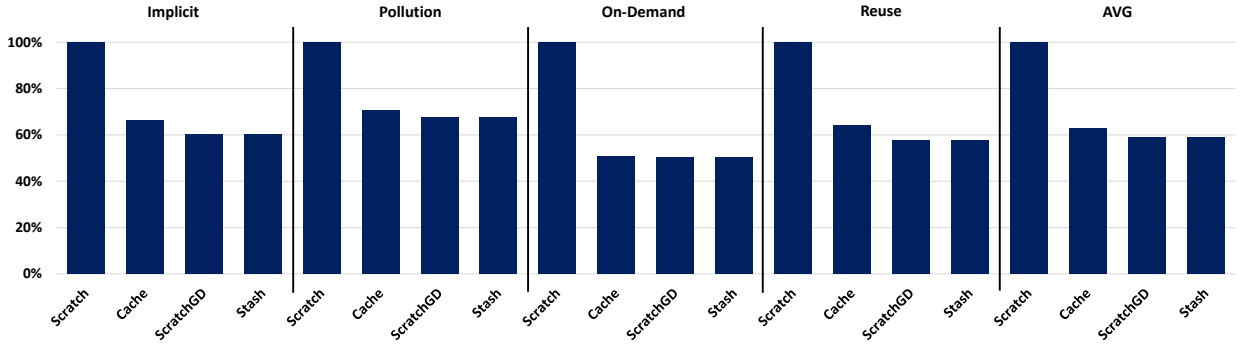
Table 4.5 shows per access energy for various hardware components used in our simulations. The table shows that scratchpad access energy (no misses for scratchpad accesses) is 29% of the L1 cache hit energy. Stash's hit energy is comparable to that of scratchpad and its miss energy is 41% of the L1 cache miss energy. Thus accessing the stash is more energy-efficient than a cache and the stash's hit energy is comparable to that of a scratchpad. In these estimates, we assumed an 8-way set associative cache with no way prediction [89].



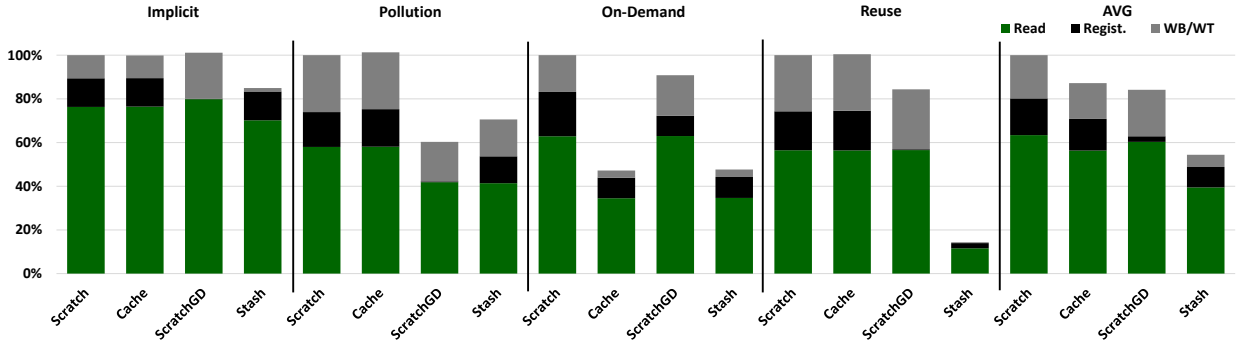
(a) Execution time



(b) Dynamic energy



(c) GPU instruction count



(d) Network traffic (flit-crossings)

Figure 4.4: Comparison of microbenchmarks. The bars are normalized to the *Scratch* configuration.

4.7.2 Microbenchmarks

Figure 4.4 shows the execution time, energy, GPU instruction count, and network traffic for the microbenchmarks using scratchpad (*Scratch*), cache (*Cache*), scratchpad with DMA (*ScratchGD*), and stash (*Stash*). We omit the remaining configurations (*ScratchG*, *StashG*, and *StashGP*) because the microbenchmarks (except Pollution) do not have any global memory accesses, so *ScratchG* is identical to *Scratch* and *StashG* is identical to *Stash*. Similar to Sections 2.7 and 3.6, the energy bars are again subdivided by where energy is consumed: GPU core, L1 cache, scratchpad/stash, L2 cache, or network. Similarly, the network traffic bars are again subdivided by message type: read, write, or writeback.

Our results show that, on average, the stash reduces execution time and consumes less energy than the scratchpad, cache, and DMA configurations – 13%, 27%, and 14% lower execution time, respectively and 35%, 53%, and 32% less energy, respectively. Overall, the microbenchmark results show that (a) the stash performs better than scratchpad, caches, and scratchpad with DMA; and (b) data structures and access patterns that are currently unsuitable for scratchpad storage can be efficiently mapped to stash. Next we discuss the sources of these benefits for each configuration.

Scratchpad vs. Stash

Compared with the scratchpad configuration, stash provides the following global addressing and global visibility benefits:

Implicit data movement: By implicitly transferring data to local memory, *Stash* executes 40% fewer instructions than *Scratch* for the Implicit benchmark and decreases execution time by 15% and energy consumption by 34%.

No cache pollution: Unlike scratchpads, stash does not access the cache when transferring data to or from local memory. By avoiding cache pollution, *Stash* consumes 42% less energy and reduces execution time by 31% in the Pollution benchmark.

On-demand loads into structures: The On-demand microbenchmark results show the advantages of on-demand loads. Since stash only transfers the data it accesses into local memory, *Stash* reduces energy consumption by 61% and execution time by 26% relative to *Scratch*, which must transfer

[8] We do not model a TLB miss, so all TLB accesses are charged as if they are hits.

the entire data array to and from the local memory.

Lazy writebacks/Reuse: The Reuse microbenchmark demonstrates how the the stash's lazy write-backs, in addition to not requiring invalidations at the end of each kernel, enable data reuse across kernels. By avoiding repeated transfers, *Stash* consumes 74% less energy and executes in 35% less time than *Scratch*.

The primary benefit of scratchpad is its energy efficient access. Scratchpad has less hardware overhead than stash and does not require a state check on each access. However, the software overhead required to load and write out data limits the use cases of scratchpad to regular data that is accessed frequently within a kernel. By adding global visibility and global addressability, stash memory eliminates this software overhead and can attain the energy efficiency of scratchpad (and higher) on a much larger class of programs.

Cache vs. Stash

Compared to cache, stash benefits from direct addressability and compact storage. With direct addressability, stash accesses do not need a tag lookup, do not incur conflict misses, and only need to perform address translation on a miss. Thus a stash access consumes *less energy* than a cache access for both hits and misses and the stash reduces energy by 35% on average.

In addition to the benefits from direct addressing, the Pollution and Reuse microbenchmarks also demonstrate the benefits of *compact storage*. In these microbenchmarks the cache configuration repeatedly evicts and reloads data because it is limited by associativity and cache line storage granularity. Thus it cannot efficiently store a strided array. Because the stash provides compact storage and direct addressability, it outperforms the cache for these microbenchmarks: up to 71% in energy and up to 22% in execution time.

Cache is able to store much more irregular structures and is able to address a much larger global data space than stash. However, when a data structure is linearizable in memory and can fit compactly in the stash space, stash can provide much more efficient access than cache with significantly less overhead than scratchpad.

ScratchGD vs. Stash

Applying DMA to a scratchpad configuration mitigates many of the scratchpad's inefficiencies by preloading the data directly into the scratchpad. Even so, such a configuration still lacks many of the benefits of global addressability and visibility present in stash. First, since scratchpads are not globally addressable, DMA must explicitly transfer all data to and from the scratchpad before and after each kernel. All threads must wait for the entire DMA load to complete before accessing the array, which can stall threads unnecessarily and create bursty traffic in the network. Second, DMA must transfer all data in the mapped array whether or not it is accessed by the program. The *On-demand* microbenchmark highlights this problem: when accesses are sparse and unpredictable stash achieves 48% lower energy and 48% less network traffic. Third, since scratchpad is not globally visible, DMA is unable to take advantage of *reuse* across kernels; therefore, stash sees 83% traffic reduction, 63% energy reduction, and 26% execution time reduction in the Reuse microbenchmark. DMA also incurs additional local memory accesses compared with stash because it accesses the scratchpad at the DMA load, the program access, and the DMA store. These downsides cause DMA to consume additional energy than the stash: Stash's stash/scratchpad energy component is 46% lower than DMA's on average.

Pollution's network traffic is 17% lower with DMA compared to stash. In Pollution, the stash's registration requests increase traffic when data is evicted from the stash before its next use because stash issues both registration and writeback requests while DMA only issues writeback requests. In general though, global visibility and addressability improve performance and energy and make stash feasible for a wider range of data access patterns.

These results validate our claim that the stash combines the advantages of scratchpads and caches into a single efficient memory organization. Compared to a scratchpad, the stash is globally addressable and visible; compared to a cache, the stash is directly addressable with more efficient lookups and provides compact storage. Compared with a non-blocking DMA engine, the stash is globally addressable and visible and can transfer data on-demand. Overall, the stash configurations always outperform the scratchpad and cache configurations, even in situations where a scratchpad or DMA engine would not traditionally be used, while also providing decreased network traffic and energy consumption.

4.7.3 Applications

In this section we evaluate the seven benchmarks for all configurations. First, we compare the *Scratch*, *ScratchG*, and *Cache* configurations to evaluate each application's original access pattern (part scratchpad and part global accesses). Next, we compare *Scratch* against *Stash* and *StashG* to examine the benefits of stash against both scratchpads (*Stash*) and caches (*StashG*). Finally, we compare the DMA configuration (*ScratchGD*) to the stash with prefetching (*StashGP*).

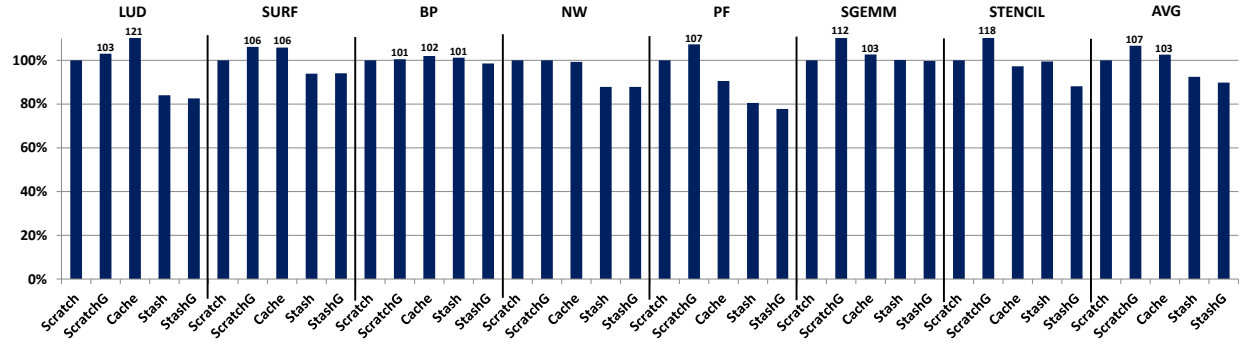
Figure 4.5 shows execution time, energy consumption, GPU instruction count, and network traffic for the seven applications, normalized to *Scratch*. Compared to other configurations in Figure 4.5, stash improves both performance and energy: compared to *Scratch* (the best scratchpad version) and *Cache*, on average *StashG* (the best stash version) reduces execution time by 10% and 12% (max 22% and 31%), respectively, while decreasing energy by 16% and 32% (max 30% and 51%), respectively. Next we analyze these results in more detail.

Scratch vs. ScratchG vs. Cache

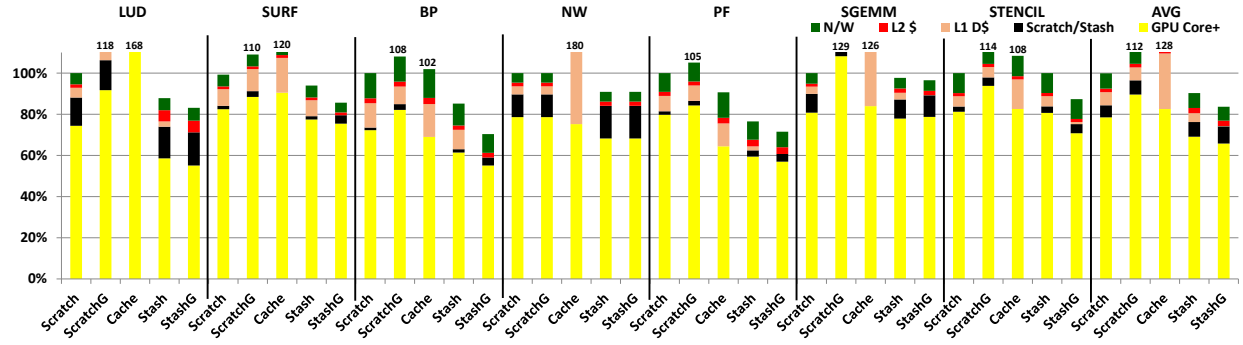
Figure 4.5a shows that *ScratchG* performs worse than *Scratch* for all applications except NW, which has no global accesses in *Scratch*. On average, *ScratchG* is 7% worse than *Scratch*, because the global accesses that are converted to scratchpad accesses increase the overall instruction count (the global accesses are better off being global as there is no temporal locality for these accesses). The increased instruction count also increases the number of scratchpad accesses in *ScratchG*. Since there is no temporal reuse for this data, each converted global access still incurs a cache access but also adds a scratchpad access. As a result, *ScratchG* also consumes more energy than *Scratch* (12% on average in Figure 4.5b).

Unlike *Scratch*, *Cache* does not need extra instructions to explicitly transfer data to local memory. However, *Cache* requires instructions to calculate global addresses. In general, converting all scratchpad accesses to global accesses does not improve performance – the global address calculations increase execution time for applications like LUD, SURF, and BP (PF is one notable exception). As a result, on average *Cache* increases execution time by 3% over *Scratch*.

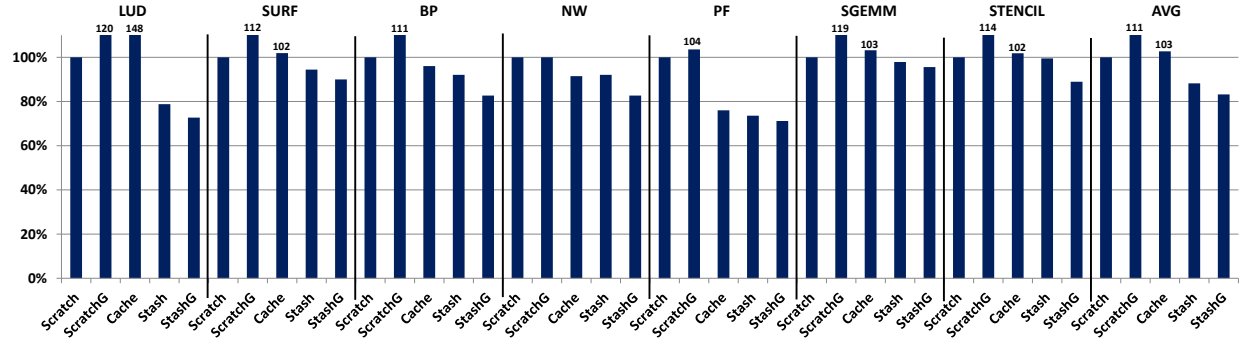
Moreover, as Section 4.7.1 showed, cache accesses consume significantly more energy than scratchpad accesses. Caches are also unable to compact data and have conflict misses which further



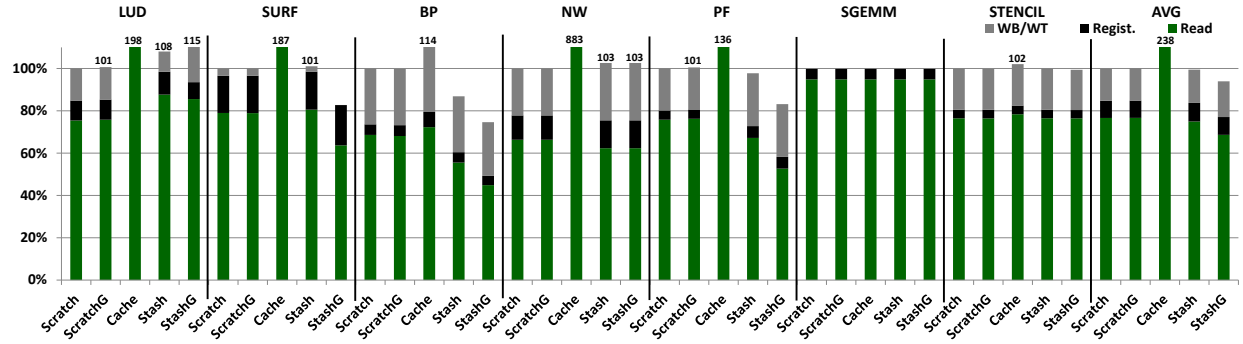
(a) Execution time



(b) Dynamic energy



(c) GPU instruction count



(d) Network traffic (flit-crossings)

Figure 4.5: Comparison of configurations for the seven benchmarks. The bars are normalized to the *Scratch* configuration.

increase both energy consumption and network traffic. As a result, *Cache* consumes an average of 28% energy more than *Scratch*.

As expected, *ScratchG* increases instruction count compared to *Scratch* for all applications (except NW which only uses the cache to load the scratchpad) as more data is accessed via scratchpad (Figure 4.5c). Although *Cache* does not need instructions to explicitly transfer data to local memory as *Scratch* does, converting local accesses to global accesses can introduce additional instructions for index computation in applications like LUD and SURF.

Finally, Figure 4.5d compares the effect of the configurations on the network traffic. *ScratchG* performs comparable or slightly worse against *Scratch*. Compared to *ScratchG*, *Cache* always performs worse due to conflict misses and inability to compact the data (on average more than 3X increase and for NW more than 8X increase in traffic).

These results show that overall, the allocation of memory locations to the scratchpad in the original applications performs better and is more energy-efficient. We therefore compare only the *Scratch* configuration to *Stash* and *StashG* below.

Scratch vs. Stash vs. StashG

Next we compare the stash configurations to *Scratch* to demonstrate the benefits of stash over a scratchpad, including the benefits of converting cache requests to stash requests in *StashG*.

Figure 4.5a shows that both stash configurations reduce execution time compared to *Scratch*. *Stash* reduces execution time compared to *Scratch* (up to 20%, 8% on average) by exploiting the stash's global addressability and visibility. The improvements are especially good for LUD, NW, and PF, which exploit the stash's global visibility to reuse data across kernels.

StashG, which converts global accesses to stash accesses, shows a further modest improvement in execution time by exploiting the stash's ability to remove index computations (index computations performed by the core for a global access are now performed more efficiently by the $Stash-map$ in hardware). As discussed below, *StashG* further reduces the instruction count compared to *Stash*. As a result, using the stash organization for all accesses (*StashG*) reduces execution time by up to 22% (10% on average). This shows that more data access patterns can take advantage of the stash.

Figure 4.5b shows that the stash configurations also reduce energy compared to *Scratch*. Com-

pared to *Scratch*, *Stash* uses the stash's global addressability to remove explicit copies and reduce both GPU core energy and L1 cache energy. By converting global accesses to stash accesses, *StashG* reduces energy even further. The stash's global addressing also removes cache pollution, which affects the performance for applications like BP, and decreases the L1 cache energy compared to the scratchpad. There are two positive energy implications when the global accesses are converted to stash accesses: (i) a stash access is more energy-efficient compared to a cache access; and (ii) the index computations performed by the $Stash - map$ mentioned above consume less energy (which reduces 'GPU core+' portion for *StashG* compared to *Stash*). These advantages help *StashG* to reduce energy by 16% (max 30%) compared to *Scratch*.

Figure 4.5c shows that many applications also benefit from the stash's global addressing, which improves execution time by reducing instruction count by an average of 12% (26% max) compared to *Scratch*. This reduction is more significant for applications that use the scratchpad/stash heavily, such as LUD, NW, PF, SURF, and BP. *StashG* further reduces instructions (17% on average compared to *Scratch*). Stencil's instruction count sees the most benefit from converting global accesses to stash accesses (*Stash* vs. *StashG*). This happens because Stencil has several data structures that are perform global accesses in the original application. By converting these accesses to stash accesses, many index computations can be efficiently done in hardware by the $Stash-map$.

As Figure 4.5d shows, in general neither *Stash* nor *StashG* significantly affects network traffic compared to *Scratch*. On average, *Stash* and *StashG* reduce network traffic by 1% and 6%, respectively, compared to *Scratch*. For PF, both stash configurations show increased writeback traffic. This occurs because the cache size (32KB) is larger than the stash size (16KB), which helps *Scratch* avoid evicting in-use data to the L2. Since the stash bypasses the cache, the larger cache space goes unused in the stash configurations. As a result, when the stash space fills up, data must be evicted to the L2. Increasing the stash size to 48KB (and a 32KB cache, which is largely unused in this configuration) completely eliminates the increased writeback traffic. Moreover, this demonstrates how a system with support for dynamic reconfiguration of on-chip memory can be useful. The stash configurations also have increased read traffic for LUD. The reason for this behavior is similar to that of the Implicit microbenchmark: as the stash keeps data around, future accesses sometimes result in indirection through the directory. As mentioned previously, a prediction mechanism to

determine the remote location of data could help resolve this issue.

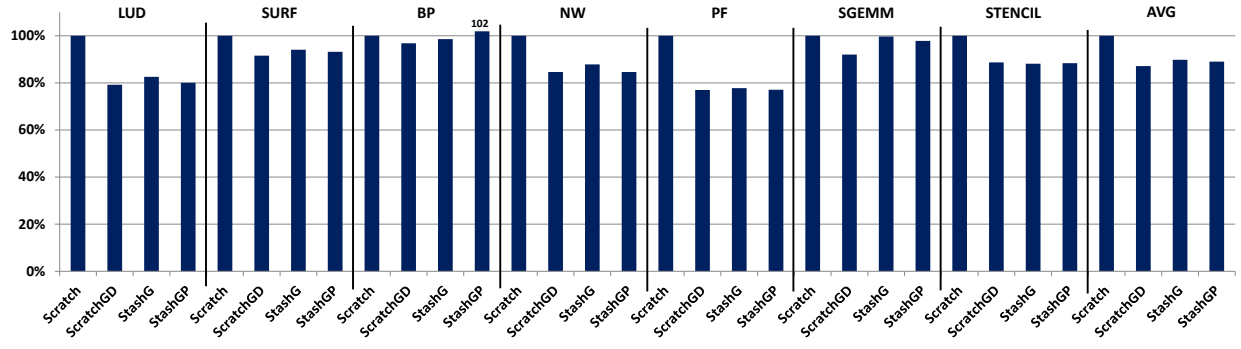
Overall, these results show that the stash effectively combines the benefits of scratchpads and caches to provide higher performance and lower energy even for current applications that are not designed to exploit the stash's new use cases. When both scratchpad and cache accesses are converted to stash accesses, across all seven applications, on average execution time is reduced by 10% and energy is reduced by 14%.

StashG vs. ScratchGD vs. StashGP

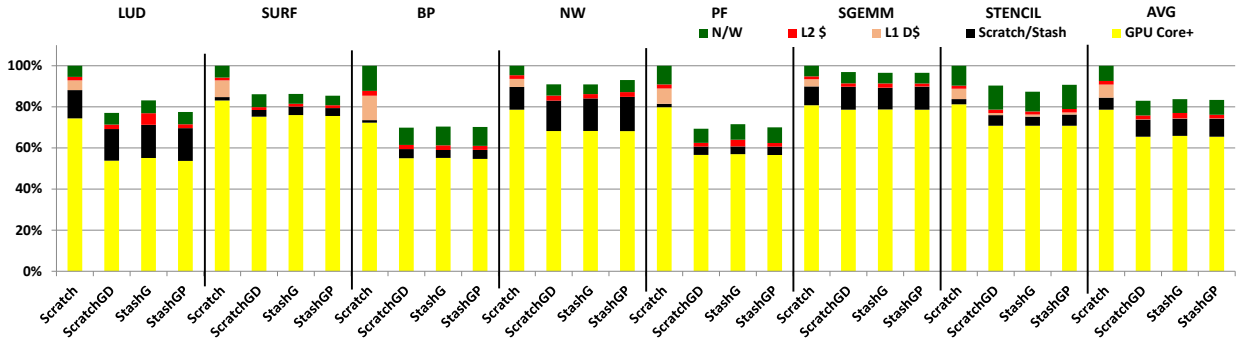
As discussed earlier, in addition to avoiding instruction count overhead, compared to *StashG*, *ScratchGD* also has a potential advantage of prefetching the data. *StashGP* applies a similar prefetching optimization to *StashG*. As the applications are not written to exploit the other benefits the stash provides over *ScratchGD* (e.g., on-demand accesses and reuse of data), the (small) performance and energy differences between *StashG*, *ScratchGD*, and *StashGP* primarily come from prefetching. However, as Figure 4.6 shows,[9] prefetching does not significantly improve on *StashG*: on average *ScratchGD* and *StashG* enhanced with a similar prefetching optimization reduce execution time by $< 3\%$ vs. *StashG*.

Nevertheless, prefetching provides some benefits for all of the applications except Stencil, which accesses the scratchpad/stash data immediately after loading it. For this subset of applications, on average *ScratchGD* and *StashGP* reduces execution time by 4% and 5%, respectively, compared to *StashG*. The slight improvement of *StashGP* over *ScratchGD* occurs because *StashGP* does not block the core for pending prefetch requests. The difference in energy consumption across these three configurations is negligible ($< 0.5\%$ on average). None of the three configurations incur any instruction count overhead and see the same instruction count: *ScratchGD* employs DMA to mitigate extraneous, explicit instructions while *StashG* and *StashGP* implicitly move data into the stash. However, prefetching sometimes hurts network traffic by fetching unnecessary data. Overall, prefetching data for these applications only provides a small benefit.

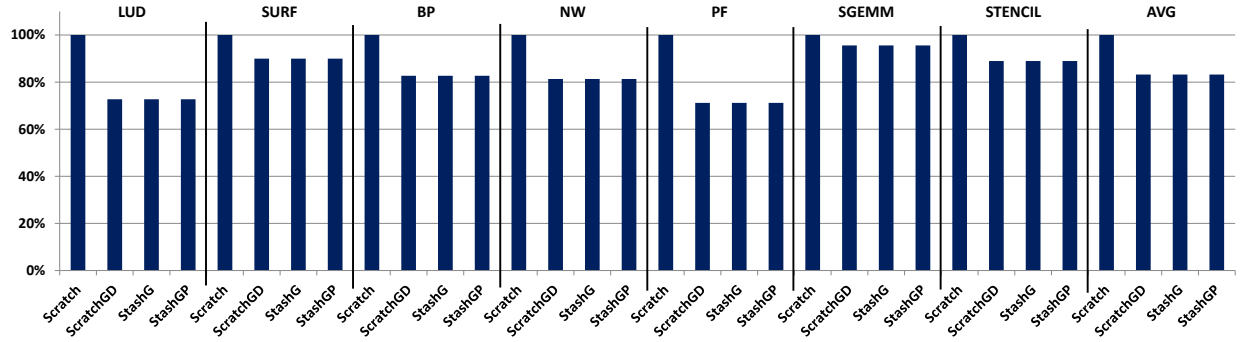
[9]For continuity, we include *Scratch* in Figure 4.6.



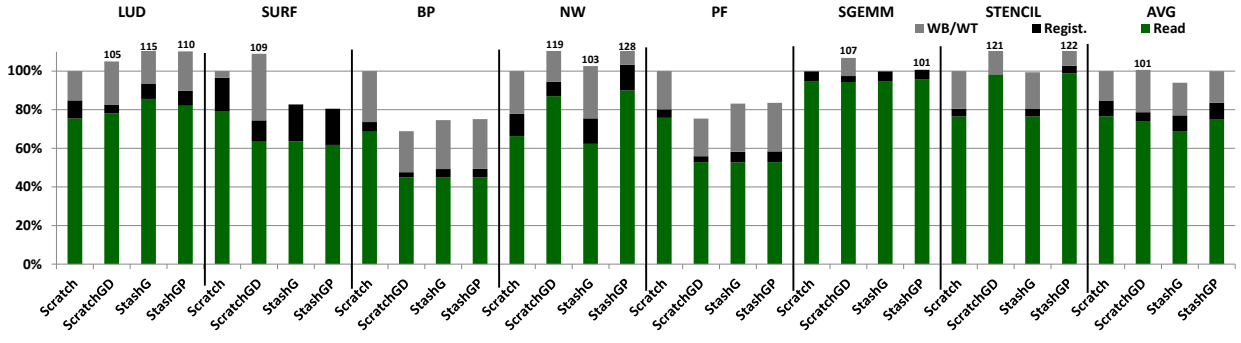
(a) Execution time



(b) Dynamic energy



(c) GPU instruction count



(d) Network traffic (flit-crossings)

Figure 4.6: Comparison of the prefetching configurations for the seven benchmarks, normalized to *Scratch*.

4.8 Summary

Efficient data movement is essential for designing energy efficient heterogeneous systems. To obtain high performance and energy efficiency, modern heterogeneous systems often use specialized memories, such as scratchpads. Software-managed scratchpads are directly addressable and provide compact storage, but are not globally addressable or visible because they they exist in a private, incoherent address space. As a result, despite the tremendous potential gains scratchpads can provide, they are not as widely used as caches. In comparison, hardware-managed caches are widely used because they can transparently exploit spatial and temporal locality without information from the software. However, caches have indirect, hardware-managed addressing and do not provide compact storage. As a result, programmers and hardware designers currently have to choose either scratchpads or caches.

In this chapter, we introduce a new memory organization, **stash**. With the stash, programmers and hardware designers no longer need to choose either scratchpads or caches – the stash provides the benefits of **both** caches and scratchpads. The stash makes specialized memories globally addressable and coherent while retaining the benefits of direct addressing and compact storage. Although there are some situations, such as irregular accesses, where caches should still be used, the results show that the stash provides better performance and energy efficiency than either caches or scratchpads. Thus, the stash represents an important step forward in efficiently moving data throughout the memory hierarchy of heterogeneous systems.

The stash also introduces new use cases, such as inter-kernel reuse of stash data. The stash can also be used to optimize storage for a more diverse set of data structures and applications. Since the stash is kept coherent and the stash is more energy efficient than the cache, the stash can achieve the benefits of direct addressability and compact storage even for data that is not amenable to traditional scratchpad storage (e.g., data that is accessed only once). Coherence also allows stash to exploit application properties such as inter-kernel reuse that are not captured with a software-managed memory. As a result, specialized memory can be used for a broader class of applications than is currently feasible.

In addition to creating new use cases, our work on the stash potentially enables many new research opportunities. Caches and scratchpads are only two of the memory organizations used in

today's heterogeneous systems. Other specialized private memory structures such as FIFOs, stream buffers, and vector registers suffer from the same inefficiencies as scratchpads. Like scratchpads, these memory organizations require explicit data movement from the coherent global address space. Thus, the ideas underlying the stash architecture can be applied to integrate other specialized, private memories into the unified, coherent, global address space. By making these specialized, private memories globally addressable and coherent, data can be moved more efficiently throughout the memory hierarchy.

In addition to showing how other private memories can be made globally visible and coherent, the stash also opens up several other exciting research directions. First, the stash can be made even more efficient by applying optimizations such as providing a flexible (vs. cache line based) communication granularity. Second, the stash's ability to reuse data provides opportunities for new stash-aware scheduling algorithms. Third, using dynamically reconfigurable SRAMs will allow the stash and cache to use different sizes for different applications. Finally, the stash can also be used on other compute units such as CPUs. We further discuss the future directions enabled by the stash in Section 7.2.3.

Chapter 5

HeteroSync: A Benchmark Suite for Fine-Grained Synchronization

5.1 Motivation

As the amount of research on optimizing the memory hierarchy of heterogeneous systems increases, it is important to be able to compare the various approaches. For example, in addition to our work, there has been several recent papers on coherence and consistency for heterogeneous CPU-GPU systems [15, 92, 71, 121]. Many of these papers introduce new microbenchmarks to measure the efficiency of their proposed changes, but often use different benchmarks that make it difficult to compare the proposed schemes.

To help alleviate this problem, we create a new microbenchmark suite, HeteroSync, that contains synchronization microbenchmarks. Specifically, HeteroSync combines the SyncPrims microbenchmarks from Section 2.6 and the relaxed atomic microbenchmarks from Section 3.3. We will release HeteroSync shortly to provide a standard set of microbenchmarks for future work. Additionally, HeteroSync allows researchers to explore the differences between various synchronization algorithms, coherence protocols, and consistency models for CPU-GPU systems.

Our results in this chapter (Section 5.3) demonstrate how HeteroSync can be used in this manner: we analyze the scalability of the HeteroSync algorithms, the GPU and DeNovoA coherence protocols, and the DRF0, DRFrlx, and HRF memory consistency models. We show that HeteroSync highlights the differences between different algorithms, coherence protocols, and consistency models as the number of CUs varies. For locally scoped microbenchmarks, DeNovoA with DRF and GPU coherence with HRF scale much better than the GPU coherence with DRF. For the hybrid and globally scoped SyncPrims, DeNovoA with DRF scales better than all other configurations. The relaxed atomics microbenchmarks show mixed scalability results: for some microbenchmarks relaxed atomics improve scalability, while for others they increase execution time, only provide

small benefits, or are unaffected by scaling the number of CUs.

Other prior work designed more efficient algorithms with fine-grained synchronization for discrete GPUs [41, 61, 62, 98, 115, 162] or collaborative CPU-GPU computing [69, 150]. But unlike our work they either do not consider tightly-coupled systems or mostly focus on applications that do not use fine-grained synchronization. We further address related work in Section 6.4.

5.2 Methodology

The simulations use a setup similar to Sections 2.5 and 3.5. The only difference from Table 2.3 is that the number of CUs varies from 1 to 15.

5.2.1 Configurations

We evaluate the following configurations, using the same implementations as Chapters 2 and 3:

GPU-DRF0 (GD0): *GD0* combines the baseline DRF memory model (no scopes) with GPU coherence and performs all synchronization accesses at the L2 cache.

GPU-HRF (GH): *GH* uses GPU coherence and HRF's HRF-Indirect memory model. GH performs locally scoped synchronization accesses at the L1s and globally scoped synchronization accesses at the L2.

DeNovoA-DRF0 (DD0): *DD0* uses the DeNovoSync0 coherence protocol (without regions), a DRF0 memory model, and performs all synchronization accesses at the L1 (after registration).[1]

GPU-DRFrlx (GDR): *GDR* uses GPU coherence with the DRFrlx memory model, which allows it to overlap relaxed atomics.

DeNovoA-DRFrlx (DDR): *DDR* uses DeNovoA coherence and the DRFrlx memory model.

Some of these configurations are only useful for analyzing either the SyncPrims or relaxed atomics microbenchmarks. Thus, in our evaluation we do not use GDR or DDR for the SyncPrims microbenchmarks (since they do not use relaxed atomics) and do not use GH for the relaxed atomics microbenchmarks (since only Flags would benefit from scoped synchronization). We use the remainder of the configurations for all of HeteroSync's microbenchmarks. Since our focus is the GPU, for all configurations, the CPU always uses the DeNovoA coherence protocol. We also

[1]To avoid cluttering the graphs, we do not include results for *DD+RO*, but we observed similar trends.

Benchmark	Input
Global Synchronization	
FA Mutex (FAM_G) Sleep Mutex (SLM_G) Spin Mutex (SPM_G) Spin Mutex with backoff (SPMBO_G)	4 TBs/CU, 10 Ld&St/thr/iter
Spin Semaphore (SS_G) Spin Semaphore with backoff (SSBO_G)	4 TBs/CU, readers: 10 Ld/thr/iter writers: 30 St/thr/iter
Local or Hybrid Synchronization	
FA Mutex (FAM_L) Sleep Mutex (SLM_L) Spin Mutex (SPM_L) Spin Mutex+backoff (SPMBO_L) Tree Barrier with local data exchange (TRBEX_LG) Tree Barrier (TRB_LG) Lock-Free Tree Barr with local data exchange (LFTRBEX_LG) Lock-Free Tree Barr (LFTRB_LG)	4 TBs/CU, 10 Ld&St/thr/iter
Spin Semaphore (SS_L) Spin Semaphore with backoff (SSBO_L)	4 TBs/CU, readers: 10 Ld/thr/iter writers: 30 St/thr/iter

Table 5.1: SyncPrims microbenchmarks with input sizes used for scaling study. The version of each microbenchmark with local and global scope are again denoted with a '_L' and '_G', respectively.

Microbenchmark	Input
Flags[154]	60 TBs
Histogram (H)[124]	256 KB, 256 bins
Histogram_global (HG)[124]	64 TBs, 256 KB, 256 bins
Histogram_global (HG-2K)[124]	64 TBs, 256 KB, 2K bins
Histogram_global-Non-Order (HG-NO)	64 TBs, 256 KB, 256 bins
Multiple Locks (ML)[64]	512 TBs
RefCounter (RC)[154]	64 TBs
Seqlocks (SEQ)[33]	512 TBs
SplitCounter (SPC)[111]	112 TBs

Table 5.2: Relaxed atomic microbenchmarks with input sizes used for scaling study.

assume support for performing synchronization accesses (using atomics) at the L1 and L2. Finally, we use self-relative speedups to compare the execution time of the microbenchmarks.

5.2.2 Benchmarks

We examine how HeteroSync's performance scales as the number of CUs vary from 1 to 15, using the algorithms we described in detail in Sections 2.6 and 3.3. Tables 5.1 and 5.2 summarize the microbenchmarks and the input sizes we use.[2] All codes use a single CPU core. For all relaxed atomic microbenchmarks, we use strong scaling: as the number of CUs increase, the total amount of work stays the same but is divided across more CUs. For the relaxed atomics scaling study, we

[2]The input sizes differ slightly from Chapters 2 and 3 to explore what happens for larger inputs.

focus on the difference between using fully relaxed atomics (GDR, DDR) and SC atomics (GD0, DD0).

For all SyncPrims, we perform 100 iterations of the critical section for two versions described in Section 2.6: a locally scoped version that shares data locally on an CU (denoted with "_L") and a globally scoped version that shares data globally (denoted with "_G"). The tree barriers, which use both local and global scope, are denoted with "_LG." We use weak scaling for all SyncPrims: as the number of CUs increase, the amount of work per thread (and per CU) remains constant. Thus, as the numbers of CUs increase contention also increases for the hybrid and globally scoped microbenchmarks; contention stays the same for the locally scoped microbenchmarks because they only compete with other threads on the same CU. Although we would have preferred to use strong scaling, the SyncPrims microbenchmarks was designed to do weak scaling. For the SyncPrims scaling study, we focus on the difference between *GD*, *GH*, and *DD*.

5.3 Results

Respectively, Figures 5.1, 5.2, and 5.3 show how the local/hybrid scoped SyncPrims, globally scoped SyncPrims, and relaxed atomics microbenchmarks in HeteroSync scale as the number of CUs varies. Overall, our results show the benefits to emerging coherence and consistency techniques for heterogeneous systems and how the microbenchmarks are able to pinpoint differences in these approaches. For the locally scoped microbenchmarks, *DD* and *GH* provide near perfect scaling, due to low contention (recall we analyze weak scaling for these benchmarks). However, *GD0* scales poorly due to increased contention. For the hybrid and globally scoped SyncPrims, increasing the number of CUs increases contention and execution time for *GH* and *DD*, but *DD* scales better than *GH*. The relaxed atomics microbenchmarks show mixed results: for some microbenchmarks relaxed atomics improve (strong) scalability, while for others they increase execution time, only provide small benefits, or are unaffected by scaling the number of CUs. Below we describe all of these results and their implications in more detail.[3]

[3]Regardless of the number of CUs, we keep the the size and access latencies for each level of the memory hierarchy constant.

5.3.1 Local/Hybrid SyncPrims

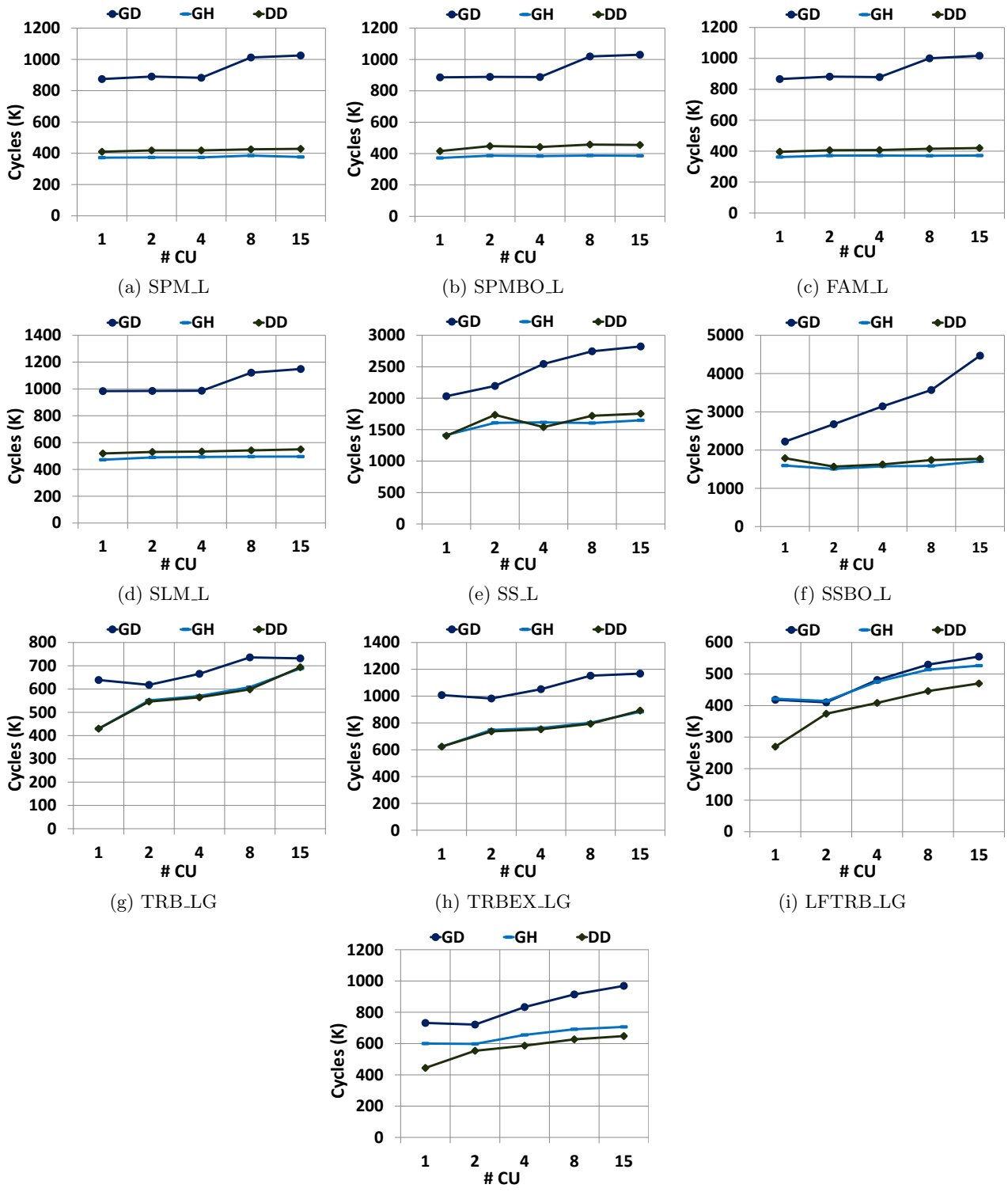
Mutexes: The locally scoped mutex algorithms (Figure 5.1a-5.1d) all show similar scaling trends: with weak scaling, *GH* and *DD* provide near perfect scaling while *GD* suffers from increased contention and scales poorly. *GH* and *DD* scale well because they can keep the data local and perform the atomics at the L1 cache; *GD* scales poorly because it writes through all dirty data on releases, invalidates all valid data on acquires, and performs atomics at the L2. For all locally scoped mutexes, as shown in Section 2.7.2, as the number of CUs increase, *GH* and *DD* clearly outperform *GD*, while *GH* slightly outperforms *DD*. Furthermore, as the number of CUs increases, as expected, *GD*'s execution time increases due to increased contention at the L2 for the atomic variables.

As discussed in Section 2.6, the locally scoped mutex algorithms have per-CU mutexes. As a result, contention for a given mutex is low.[4] Thus, we see that all of the mutex algorithms obtain similar execution times. The one exception is SLM_L, where the decentralized ticket lock increases execution time (e.g., 24% for *DD* with 15 CUs) over the centralized ticket lock. Thus, as expected, decentralizing the ticket lock does not help in cases of low contention.

Semaphores: The locally scoped semaphores (Figure 5.1e and 5.1f) have very different scaling trends: *GH* and *DD* do not have perfect scaling and *GD* scales much worse than it did in the mutex algorithms. *GD* scales poorly because it has even more synchronization accesses than the mutexes and these accesses must be performed at the L2 – whereas *GH* (since scope is local) and *DD* can access the variables locally. The semaphore's reader-writer format also impacts whether *DD* or *GH* provides the best performance. When the writer enters the semaphore first (as happens to be the case for SS_L, 4 CUs), *DD* slightly outperforms *GH* for SS_L because it immediately obtains ownership for read-write data and can reuse it across subsequent acquires and releases. However, when the readers enter the semaphore first, *DeNovoA* must invalidate this data; in comparison, *GH* exploits the scope information to retain the data. Adding backoff (SSBO_L) reduces *DD*'s overheads such that *GH* and *DD* provide similar performance (*GH* is 9% better than *DD* for 8 CUs, and 4% better for 15 CUs).

Barriers: As the number of CUs increase, the barrier algorithms (Figure 5.1g-5.1j) execution time

[4] Although contention for a given mutex is low, since *GD* sends all atomics to the L2, L2 contention increases as the number of CUs increase, which hurts *GD*'s scalability.



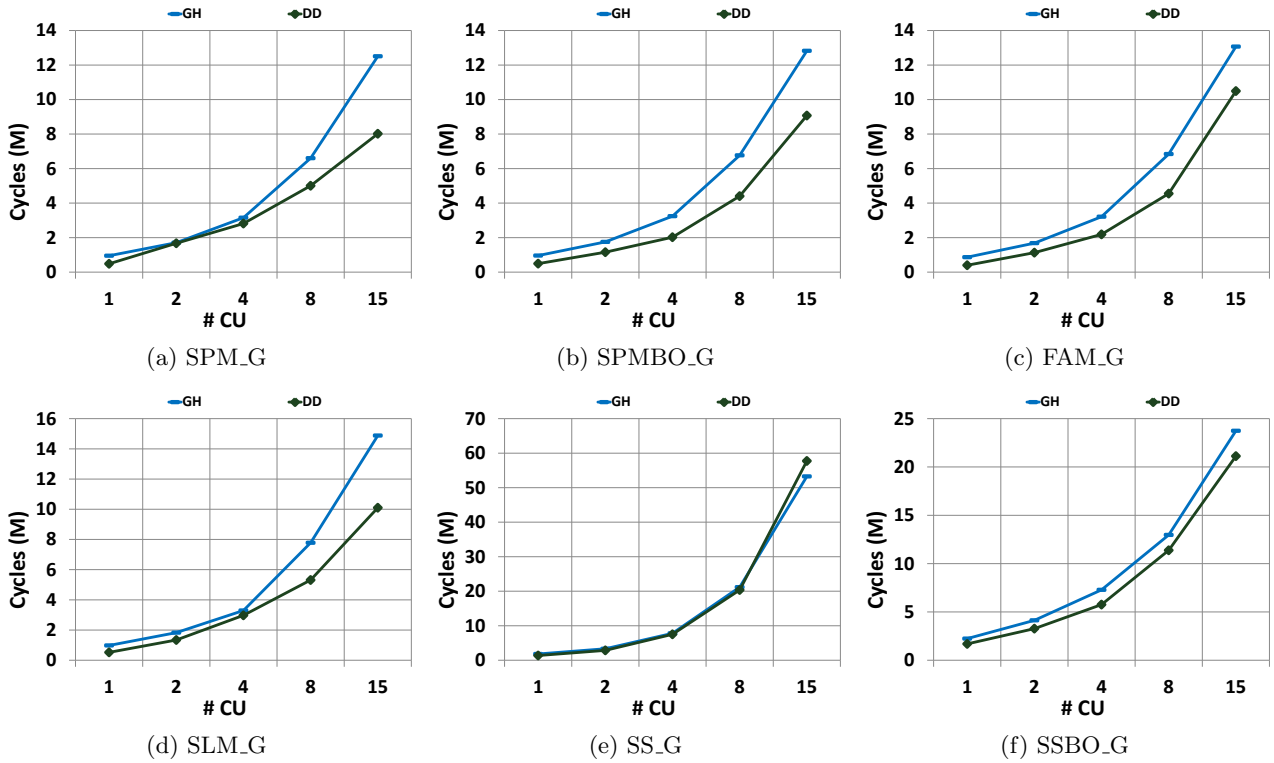
(a) SPM_L (b) SPMBO_L (c) FAM_L

(d) SLM_L (e) SS_L (f) SSBO_L

(g) TRB_LG (h) TRBEX_LG (i) LFTRB_LG

(j) LFTRBEX_LG

Figure 5.1: Weak scaling results for local and hybrid scoped synchronization benchmarks.



(a) SPM_G (b) SPMBO_G (c) FAM_G

(d) SLM_G (e) SS_G (f) SSBO_G

Figure 5.2: Weak scaling results for globally scoped synchronization benchmarks.

increases for *GD*, *GH* and *DD*. Although the local barriers stays the same, more TBs join the global barrier as the number of CUs increase. Nevertheless, *DD* and *GH*'s ability to retain some of the data locally reduces execution time compared to *GD* (e.g., *DD* is 12% better than *GD* for 2 CU's). Moreover, adding local data exchange (TRBEX_LG, LFTRBEX_LG) further increases *DD*'s benefits over *GD*, because they can reuse this data. These results confirm our finding that *DD* outperforms *GH* when hybrid scope is used (Figure 2.3), but also show that the amount *DD* and *GH* benefit over *GD* varies with the number of CUs. Moreover, as expected, compared to the atomic tree barrier the lock-free tree barrier reduces execution time (e.g., *GD* is 28% faster for 8 CUs) and scales better.

5.3.2 Global SyncPrims

Mutexes: Since *GD* and *GH* provide the same performance when scope is global, we only show *GH* in Figure 5.2. As the number of CUs increase, as expected both *GH* and *DD* scale poorly.

Nevertheless, *DD* always scales better than *GH* for all globally scoped mutexes (Figure 5.2a-5.2d) because it is better able to exploit reuse of written data and atomics, which we previously showed only for 15 CUs (Figure 2.4). For example, for 8 CUs, *DD* outperforms *GH* by 24%, 29%, 33%, and 32%, respectively, for SPM_G, SPMBO_G, FAM_G, and SLM_G.

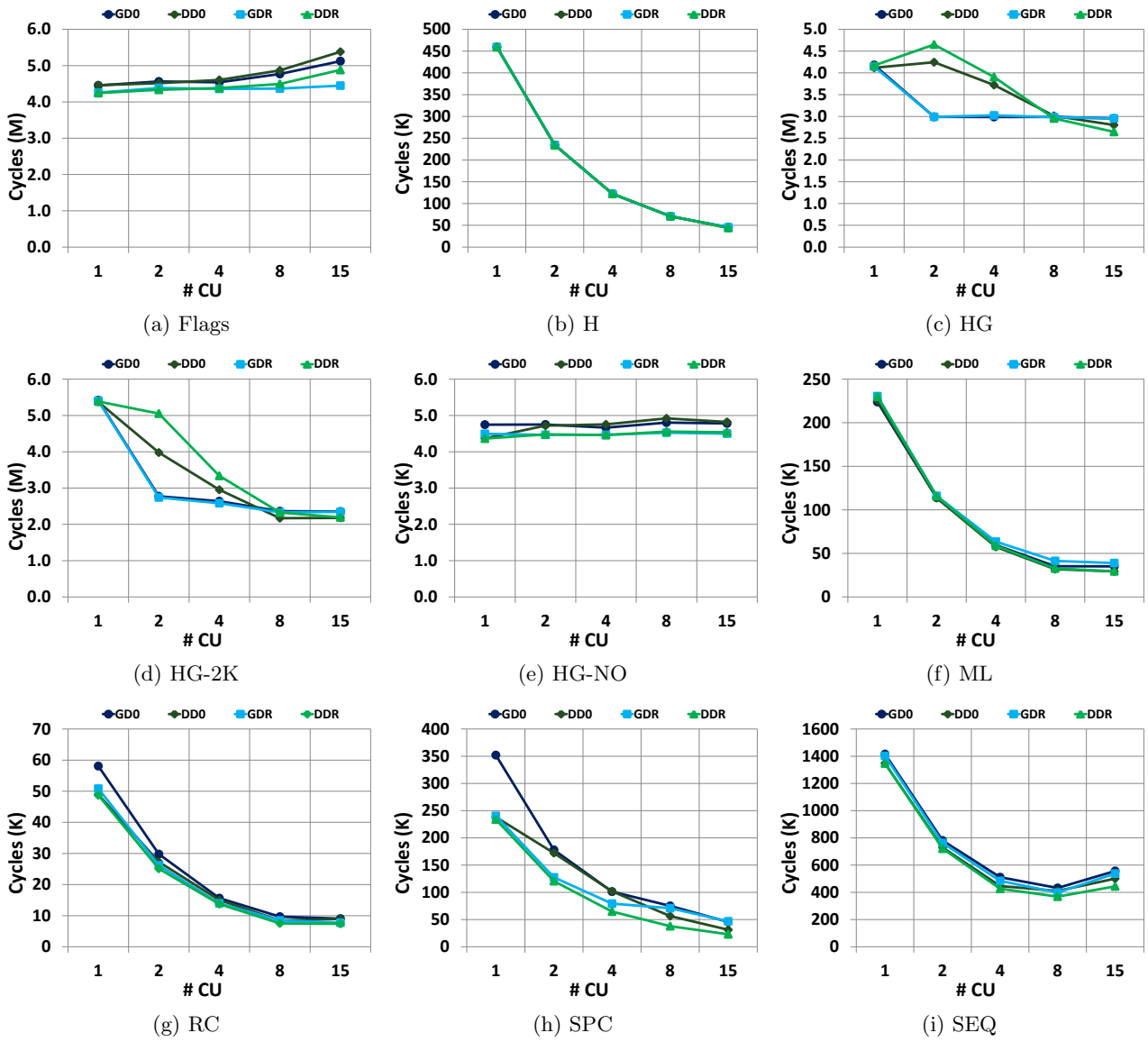
SPMBO_G again does not reduce execution time compared to SPM_G, despite increased contention for the globally shared mutexes. SPMBO_G performs backoff by executing instructions (NO-OPs). Thus, even though SPMBO_G reduces contention for the shared mutex, it does not reduce execution time. However, for higher levels of contention (e.g., 15 CUs) SLM_G's decentralized ticket lock is more scalable than FAM_G for *DD* (but not *GH*) because *DeNovoA* allows threads to spin locally in their L1 caches.

Semaphores: Execution time also increases for the globally scoped semaphores (Figure 5.2e and 5.2f) as contention increases. For SS_G, increasing contention hurts *DD* more than *GH* because more ownership requests must be sent to remote L1s. For 15 CUs, *DD* is 8% worse than *GH*. Introducing backoff significantly reduces execution time (63% for *DD*, 55% for *GH* with 15 CUs) and *DD* has 11% less execution time than *GH*.

5.3.3 Relaxed Atomics

We use strong scaling for the relaxed atomics microbenchmarks. ML, RC, SPC, and SEQ (Figure 5.3f-5.3i) see significant reductions in execution time when the number of CUs increase. This shows that they are able to effectively partition the work across the CUs, especially through 8 CUs. Using relaxed atomics also reduces execution time for RC, SPC, and SEQ, although the gains are sometimes small. In the best case, for SPC with 4 CUs, *DDR* (*GDR*) reduces execution time by 36% (22%) over *DD0* (*GD0*). However, ML does not see the same effect – relaxed atomics always increase execution time due to increased contention. Although *DeNovoA*'s ability to reuse data lessens the impact (in the worst case, with 8 CUs, *DDR* increases execution time by 3% over *DD0*), with GPU coherence the increased contention is especially harmful (in the worst case, with 8 CUs, *GDR* increases execution time by 17% over *GD0*).

Moreover, *DeNovoA* reduces execution time compared to GPU coherence regardless of the number of CUs used, by reusing written data and atomics. For example, for SPC with 15 CUs,



(a) Flags (b) H (c) HG

(d) HG-2K (e) HG-NO (f) ML

(g) RC (h) SPC (i) SEQ

Figure 5.3: Strong scaling results for relaxed atomic benchmarks.

DDR reduces execution time by 51% over *GDR*. Similarly, for ML with 15 CUs, *DDR* reduces execution time by 24% over *GDR*. In general, relaxed atomics improve the scalability of these algorithms, although there are some exceptions (ML as discussed above, SPC, with 8 and 15 CUs with *GDR*, and SEQ, with 15 CUs for *GDR* and *DDR*) where the extra contention caused hurts scalability.

Flags (Figure 5.3a) execution time scales similarly to LFTRB_LG, because the lock-free tree barrier makes up a significant portion of Flags' execution time. Nevertheless, as the number of

CUs increases, the benefits from using relaxed atomics increase (for 15 CUs, 9% less execution time for *DDR* vs. *DD0* and *GDR* vs. *GD0*) and GPU coherence reduces execution time more than *DeNovoA* because of the overhead of remote L1 ownership requests.

Different histograms show different scaling trends. H (Figure 5.3b) scales well because it uses the scratchpad to keep a private, local count of the counter updates before it updates the global counters. This reduces contention for the shared counters by reducing the number of updates). As a result, as expected, H performs much better than the histograms that perform all updates globally (HG and HG-2K)[5] As the number of CUs increases, contention increases because there is less work done per TB. However, there are still relatively few shared counter updates, so all configurations provide roughly equivalent performance (with 15 CUs, *DD0* reduces execution time by 3% over *GD0*, and *DDR* reduces execution time by 1% over *DD0*). If the number of CUs were increased even further, the differences between configurations would potentially grow.

Unlike H, HG-NO (Figure 5.3e) is relatively unaffected by increasing the number of CUs. This is not surprising, because HG-NO only uses a single TB. Thus, increasing the number of CUs only affects which CU may own the data from the previous phase of the application (HG). Relaxed atomics reduce execution time, but the gains are small ($\leq$ 6% in all cases) and are unaffected by varying the number of CUs.

Increasing the number of CUs shows mixed results for HG and HG-2K (Figure 5.3c and 5.3d). For example, for HG, increasing the number of CUs from 1 to 2 increases execution time by 3% for *DD0* and 10% for *DDR*. Increasing the number of histogram bins (HG-2K) reduces the gap between *DeNovoA* and GPU coherence (with 2 and 4 CUs), but does not eliminate it. Similarly, using relaxed atomics with *DeNovoA* does not help for 2 and 4 CUs due to MSHR saturation. However, as the number of CUs increases, *DeNovoA* is able to overcome this overhead and reduce execution time compared to GPU coherence. Similarly, *DDR* reduces execution time by 6% over *DD0* when there are more CUs by enabling more overlap of requests in the memory system. *GDR* does not significantly improve on *GD0* because of increased contention. Thus, HG and HG-2K show mixed scalability results, which earlier results in this section did not show, and relaxed atomics do not significantly improve scalability.

[5]To stress the memory system, HG and HG-2K do 4X more binning than H. This exaggerates the gap between HG/HG-2K and H. However, the general trends still hold when HG and HG-2K perform the same binning as H.

5.4 Summary

In Chapters 2 and 3, we have demonstrated to design a more efficient memory hierarchy for heterogeneous systems by adjusting the division of complexity between coherence and consistency in modern heterogeneous systems and providing better support for relaxed atomics in heterogeneous systems. Here, we examine how these solutions perform as the system for different sized systems using HeteroSync, a new benchmark suite that contains various synchronization microbenchmarks.

HeteroSync allows us to compare algorithm scalability, coherence protocols, and consistency models for heterogeneous systems. Our results show that many of the trends we observed in Chapters 2 and 3 hold as the number of CUs vary. In general, DeNovoA scales better than GPU coherence, especially as contention increases, although using an HRF consistency model improves GPU coherence's scalability for microbenchmarks that can take advantage of HRF's locally scoped synchronizations. However, the relaxed atomics microbenchmarks do not always scale well due to increased contention in the memory system. These results show how HeteroSync can be used as a standard set of benchmarks by researchers to compare the efficiency of their schemes to prior work.

Chapter 6

Related Work

6.1 Coherence & Consistency for Heterogeneous Systems

Recently, there has been significant work on optimizing coherence and consistency for heterogeneous systems. Here, we compare these approaches our work.

6.1.1 Memory Consistency Models

As discussed in Chapter 2, the HRF memory model provides a scoped, relaxed memory model for heterogeneous CPU-GPU systems with write-through GPU L1 caches and software-managed coherence [77]. HRF uses scopes to improve performance by keeping invalidations local as much as possible. Unfortunately scoped synchronization is difficult for programmers to reason about and significantly complicates the consistency model. Scoped synchronizations require the programmer/compiler to identify at which level in the hierarchy data is shared at. When this information is unavailable or uncertain, software must be overly conservative and may invalidate useful data. Our work shows that we can provide equivalent or better performance, without using scopes, across a wide range of workloads.

The HRF-Relaxed memory model addresses many of HRF's shortcomings by generalizing it to allow formalization of and reasoning about more complicated memory models and relaxed atomics [63]. However, HRF-Relaxed is even more complex than HRF. Batty, et al. have also developed a similar memory model [28] that formally proves that a partial order over the memory operations is sufficient for SC-for-DRF. Unlike our work, none of this work examines how to provide SC semantics for relaxed atomics.

Other prior work on memory consistency models for GPUs found that the TSO and relaxed memory models did not significantly outperform SC in a system with MOESI coherence and write-

Feature	Benefit	HSC [125]	FC[93], TC[141]	Quick Release[71]	Remote Scopes[121, 159]	DD
Reuse Written Data	Reuse written data across synchs	✓	✗	✓	✓	✓
Reuse Valid Data	Reuse cached valid data across synchs	✓	✗	✓	✗	✗
No Bursty Traffic	Avoid bursts of writes	✓	✓	✗	✗	✓
No Invalidations/ACKs	Decreased network traffic	✗	✓	✗	✗	✓
Decoupled Granularity	Only transfer useful data	✗	✓	✓ (for STs)	✓ (for STs)	✓
Reuse Synchronization	Efficient support for fine-grained synch	✓	✗	✓	✓	✓
Dynamic Sharing	Efficient support for work stealing	✓	✓	✗	✓	✓

Table 6.1: Comparison of DeNovoA to other GPU coherence schemes. The read-only region enhancement to DeNovoA also allows valid data reuse for read-only data.

back caches [72, 73]. However, the work does not measure the coherence overhead of the studied configurations or evaluate alternative coherence protocols. Furthermore, DeNovoA also has several advantages over an ownership-based MOESI protocol, as discussed in Section 2.2.

6.1.2 Coherence Protocols

There has also been significant prior work on optimizing coherence protocols for standalone GPUs or CPU-GPU systems. Although conventional hardware protocols such as MESI support fine-grained synchronization with DRF, prior research has shown that they are a poor fit for conventional GPU applications [71, 141]. Additionally, the DeNovo project has shown that for CPUs, DeNovo provides comparable or better performance than MESI at much less complexity [46, 152, 153].

Other recent work has also improved coherence for GPUs [141] or heterogeneous systems [93]. These protocols provide some of the same benefits as DeNovoA, but do not consider fine-grained synchronization or impact on consistency models [138]. Table 6.1 compares *DD* to the most closely related prior work across the key features from Table 2.2:

HSC[125]: Heterogeneous System Coherence (HSC) is a hierarchical, ownership-based CPU-GPU cache coherence protocol. HSC provides the same advantages as the ownership-based protocols we discussed in Section 2.2. By adding coarse-grained hardware regions[1] to MOESI, HSC aggregates coherence traffic and reduces MOESI's network traffic overheads when used with GPUs. However, HSC's coarse regions restrict data layout and the types of communication that can effectively occur. For example, if the communication frequency does not conform to the hierarchical assumptions of the hardware, region coherence could add to network overhead and may harm performance. Moreover, HSC's coherence protocol is significantly more complex than DeNovoA.

[1]HSC's regions aggregate coherence traffic for consecutive cache lines while DeNovoA's regions pass information from the software to the hardware to perform selective self-invalidations

TemporalCoherence[141] & **FusionCoherence**[93]: FusionCoherence and TemporalCoherence use timestamp-based protocols for fixed function accelerators that utilize self-invalidations and self-downgrades and thus provide many of the same benefits as *DD*. However, this work does not consider fine-grained synchronization or impact on consistency models.

QuickRelease[71]: QuickRelease reduces the overhead of synchronization operations in conventional GPU coherence protocols and allows data to be reused across synchronization points. QuickRelease adds store FIFOs next to the caches to track what data is dirty and needs to be written through to the LLC at the next synchronization point, which reduces overhead. However, QuickRelease requires broadcast invalidations to ensure that no stale data can be accessed. Additionally, QuickRelease does not have efficient support for algorithms with dynamic sharing: all shared data must use the LLC in algorithms with dynamic sharing.

RemoteScopes[121, 159]: RemoteScopes improves on QuickRelease by providing better support for algorithms with dynamic sharing. In the common case, dynamically shared data synchronizes with a local scope and when data is shared, RemoteScopes "promotes" the scope of the synchronization access to a larger common scope to synchronize properly. Although RemoteScopes improves performance for applications with dynamic sharing, because it does not obtain ownership, it must use acknowledgments and other heavyweight hardware mechanisms to ensure that no stale data is accessed. For example, RemoteScopes flushes the entire cache on acquires and uses broadcast invalidations and acknowledgments to ensure data is flushed. In comparison, DeNovoA provides the same support but is much simpler, does not cause extra cache flushes, and does not require the programmer to add an additional "remote scope" to each synchronization access.

Overall, while each of the coherence protocols in previous work provides some of the same benefits as *DD*, none of them provide all of the benefits of *DD*. Furthermore, none of the above work explores the impact of ownership on consistency models.

6.1.3 Subsequent Coherence Protocols

Some subsequent work adopted a similar approach to ours. For example, UMH applies similar ideas to ours to systems with multiple GPUs [165]. Koukos, et al. also examine how to design efficient, tightly coupled CPU-GPU systems with a unified address space and coherent caches [92].

To do this, they extend the VIPS coherence protocol [3, 52, 84, 85, 131, 132, 133, 134] (which is similar to DeNovo, as prior work has discussed [151]). The CPU cores use VIPS and the GPU CUs use GPU-style coherence, and the system uses the HRF consistency model. Thus, their system is similar to *GH*. One key difference is that their system is hierarchical and uses a private-shared classifier to avoid flushing and invalidating private data to lower levels of the system.

Heterogeneous Lazy Release Consistency (hLRC) also builds on DeNovoA. Unlike DeNovoA, hLRC only obtains ownership for synchronization variables, and writes through dirty data to the LLC [15]. hLRC avoids conservatively invalidating data and reduces LLC pressure for written data. The authors show that this provides a 7% improvement on DeNovoA. In Section 7.2.1, we discuss how future work could combine the hLRC and DeNovoA's approaches.

Other subsequent work focuses on integrating discrete GPUs with CPUs from other vendors [9]. In these systems it is more difficult to implement full HW cache coherence since the CPU and GPU vendors are different. Their solution is do selective caching: GPU caches only cache data that do not require coherence updates from the CPU (a cuckoo filter is used to identify uncacheable data). As a result, only data that is accessed privately by the GPU is cached on the GPU, which reduces the overhead of coherence transactions. This approach works well for the streaming applications the authors focus on. However, for applications with fine-grained synchronization and communication across the CPU and GPU, the overheads are likely to be higher. Other work takes a similar approach for streaming workloads when the vendor designs both the CPU and GPU [27].

6.2 Relaxed Atomics

As relaxed atomics have been a long-standing open problem in the concurrency community, many others have also explored how to make it easier to use relaxed atomics. Here we discuss how this work relates to ours.

6.2.1 Memory Consistency Models

As discussed in Chapter 2, the HSA, HRF, and OpenCL memory models seek to mitigate the overhead of atomics with another construct: scoped synchronization [28, 63, 77, 78, 96]. These models allow the programmer to distinguish some atomics as having local scope (vs. global scope)

while retaining SC semantics. The HSA memory model adds relaxed atomics to HRF's memory model but, unlike DRFrlx, does not provide SC semantics in the presence of relaxed atomics. However, scoped synchronization based models do not address the overheads for globally scoped synchronization. Additionally, in Chapter 2 we showed that with an appropriate coherence protocol (e.g., the DeNovoA protocol), scopes are not worth the added complexity to the memory model.

Other work has tried to improve support for relaxed atomics in C, C++, Java, HSA, HRF, and OpenCL [28, 63, 78, 81, 83, 94, 123]. we take a different approach, motivated by how developers use relaxed atomics in heterogeneous systems, and extend the existing DRF memory models to incorporate these use cases with SC-centric guarantees. Previous work has also examined how applications with relaxed atomics behave on various multi-core CPUs with weak memory models [67, 130] and GPUs [143, 144]. In addition to exploring the performance benefits of certain fencing schemes, this work also demonstrates the difficulty in correctly synchronizing applications on architectures that do not use SC-centric consistency models, which further motivates designing simpler, SC-centric consistency models.

Others have examined how GPUs perform for the SC, TSO, and SPARC RMO memory models [72, 73]. They found that the TSO and relaxed memory models did not significantly outperform the SC memory model. However, they do not consider the impact of relaxed atomics. Additionally, recent work has examined graph analytics workloads on GPUs [12, 161]. These papers motivate our work, because they show that graph analytics workloads suffer from poor GPU support for atomics, load imbalance, inefficient utilization of L1 caches, and cache misses – issues that our work helps resolve.

Finally, recent work has been examining how to properly specify, verify, and translate memory consistency models in existing systems [101, 102, 103, 105, 108, 107, 155]. The authors have found numerous deficiencies, especially in weak memory consistency models. These findings demonstrate the fragility of existing memory consistency models and further motivate our work to use the simpler, easier-to-understand DRF memory consistency model. Furthermore, if DRFrlx is adopted, we view their work as being complementary, because it should make it easier to ensure that DRFrlx is specified, verified, and translated correctly.

6.2.2 Memory Orderings

This work focuses on `memory_order_relaxed`. However, some applications use other relaxed memory orderings such as the release and acquire memory orderings. For example, Seqlocks' reader-side `seq` accesses can use release-acquire ordering [33]. Since these memory orderings are not our focus, we do not explore them in this work. However, PL_{pc}'s [5, 64] unessential operations and loop reads/writes could be used to ensure SC for some of these applications.

`Memory_order_consume` can improve performance compared to `memory_order_acquire` by relaxing the ordering of subsequent memory accesses with respect to the consume operation [112]. Specifically, when there is dependency ordering between a paired write is dependency ordered before a load consume and the load consume carries a dependency to some later memory reference, then any memory reference(s) before the paired write will happen before it. Consume provides some similar relaxations to quantum, but allows less reorderings because it relies on dependencies for ordering, whereas quantum does not have any such ordering constraints and thus can be reordered more. Moreover, it is hard for compilers to correctly identify dependence ordering [112] and the C++17 standard advises against its use, as it does not appreciably improve performance over acquire [142].

6.2.3 Other Related Work

Coup also exploits commutativity to improve performance of updates to shared data [163]. By adding an 'update' state to the coherence protocol, multiple caches may perform commutative updates in parallel to a given cache line. Although our work also exploits commutativity, Coup focuses on how to efficiently support commutative operations in the coherence protocol, whereas we created a new memory model that provides more robust semantics for several classes of atomic operations, not just commutative atomics. Coup also requires adding new states to the MESI coherence protocol, which is already very complicated; our work does not require any changes to the coherence protocol.

Although conventional hardware protocols such as MESI also reduce the benefits of relaxed atomics, we do not compare to them in this work because prior research has observed that they incur significant complexity (e.g., writer-initiated invalidations, directory overhead, and many transient

states leading to cache state overhead) and are a poor fit for conventional GPU applications [71, 141]. Instead we use the DeNovoA coherence protocol, which has been shown to provide provides comparable or better performance than MESI at much less complexity for multi-core CPUs [46, 90, 152, 153] and heterogeneous systems (Chapter 2).

Recently domain specific languages (DSLs) like Delite [76, 147, 148, 149], Halide [127, 128, 129], and TensorFlow [1, 2, 82] have emerged and made it easier to write portable, high performance programs for heterogeneous systems. As it relates to relaxed atomics (and memory consistency models in general), a key property of these DSLs is that the only expert programmers (or compilers) writing the DSLs need to use relaxed atomics – the programmers who use the DSLs can effectively ignore these issues in favor of higher level constructs that the DSLs provide. However, relaxed atomics are extremely important for obtaining high performance in heterogeneous systems, and DSLs like Delite, Halide, and TensorFlow are often based on languages like C++ and CUDA, whose semantics for using relaxed atomics correctly are broken (as discussed in Chapter 3). Thus, although DSLs may make it easier for programmers to ignore relaxed atomics, having better semantics for relaxed atomics will make it easier for the DSL developers to create DSLs that are portable, correct, and high performance. Additionally, DSLs that use relaxed atomics are similar to libraries using relaxed atomics – a relaxed atomic used within the higher level DSL constructs does not require the user to understand the construct's implementation as long as the DSL developer properly conveys the expected pre- and post-conditions for SC executions of the (quantum-equivalent) program.

6.3 Private Memories

The most relevant work to the stash was published concurrently with the stash [17]. Like stash, it also examines how to make scratchpads globally visible and coherent but focuses on how to provide software and compiler support for the co-existence of caches and scratchpads on a given CPU. In comparison, we focus on providing efficient hardware support for globally visible and coherent scratchpads in GPUs.

There is also much prior work on improving private memories for CPUs and GPUs. Table 6.2 compares the most closely related work to stash (other than [17]) using the benefits from Table 4.1:
Bypassing L1: (MUBUF [19]): L1 bypass does not pollute the L1 when transferring data between

Feature	Benefit	Bypass L1 [19]	Change Layout [40, 45]	Elide Tag [136, 164]	Virtual Private Memories [47, 48, 49, 106]	DMAs [30, 80]	Stash
Directly addressed	No HW translation access	✓	✗	✗, ✓	✓	✓	✓ (on hit)
	No tag access	✓	✗	✓ (on hit)	✗	✓	✓
	No conflict misses	✓	✗	✗	✓	✓	✓
Compact storage	Efficiently use SRAM storage	✓	✓	✗	✓	✓	✓
Global addressing	Implicit data movement	✗	✓	✓	✗	✗	✓
	Don't pollute other memories	✓	✓	✓	✓	✓	✓
	On-demand loads	✗	✓	✓	✗	✗	✓
Global visibility	Lazy writebacks to global AS	✗	✓	✓	✗	✗	✓
	Reuse across kernels or phases	✗	✓	✓	Partial	✗	✓
Applied to GPU		✓	✗, ✓	✗	✗, ✗, ✗, ✓	✓	✓

Table 6.2: Comparison of stash and prior work.

global memory and the scratchpad, but does not offer any other benefits of the stash.

Change Data Layout: (Impulse [40], Dymaxion [45]): By compacting data that will be accessed together, this technique provides an advantage over conventional caches, but does not explicitly provide other benefits of scratchpads.

Elide Tag: (TLC [136], TCE [164]): This technique optimizes conventional caches by removing the need for tag accesses (and TLB accesses for TCE) on hits. Thus, this technique provides some of the benefits scratchpads provide in addition to the benefits of caches. However, it relies on high cache hit rates (which are not common for GPUs) and does not remove conflict misses or provide compact storage of the stash.

Virtual Private Memories (VLS [49], Hybrid Cache [47], BiN [48], Accelerator Store [106]): Virtualizing private memories like scratchpads provides many of the benefits of scratchpads and caches. Like scratchpads, these techniques do not require address translation in HW and do not have conflict misses; like a cache, they also reuse data across kernels while avoiding polluting other memories. However, it requires tag checks and has explicit data movement which prevents lazily writing back data to the global address space. Furthermore, these techniques do not support on-demand loads [2] and only partially support reuse.[3]

DMAs: (CudaDMA [30], D^{2}MA [80]): A DMA engine on the GPU can efficiently move data into the scratchpad without incurring excessive instruction overhead and polluting other memories. However, as discussed earlier, it does not provide the benefits of on-demand loads (beneficial with control divergence), lazy writebacks, and reuse across kernels.

[2]VLS does on-demand accesses after thread migration on a context switch, but the initial loads into the virtual private store are through DMA. Although we do not discuss context switches, the stash's global visibility and coherence means that the stash can lazily write back stash data on context switches.

[3]In VLS, if two cores want to read/write the same global data conditionally in alternate phases in their VLS's, then the cores have to write back the data at the end of the phase even if the conditional writes don't happen.

In summary, while these techniques provides some of the same benefits as the stash, *none of them provide all of the benefits.*

6.4 Synchronization Benchmarks

There have been many studies on synchronization benchmarks for CPUs (e.g., Synchrobench [67]), but we focus on those for GPUs. Previous work has created suites of GPU microbenchmarks that use various synchronization primitives for discrete GPUs: several papers have explored the design of concurrent queues [41] and lock-free data structures [115]. Others have benchmarks that use fine-grained locking in a similar manner to the SyncPrims mutex locks [61, 62], but use them to explore transactional memory in discrete GPUs. More recent work has explored optimizing performance for lock-free applications on GPUs through optimized assembly code and rollback for cases where deadlock occurs [98, 162]. None of these papers examine how their algorithms perform on tightly coupled systems with DRF or HRF consistency models though. Chai and Hetero-Mark introduce benchmarks for collaborative CPU-GPU computing, but they focus on applications that do not use fine-grained synchronization [69, 150]. Some notable exceptions are Chai's *Image Histogram - Input Partitioning* and Hetero-Mark's *Color Histogram*, which are similar to H and HG, and Chai's *Padding*, which is similar to Flags.

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

Traditionally, Moore's Law implicitly improved efficiency. As Moore's Law has started to slow down in recent years, heterogeneous systems with specialized compute units have become increasingly popular due to their high performance and energy efficiency for specific workloads. Unfortunately, heterogeneous systems are inefficient and hard to program. especially for emerging applications that use specialized memories and graph analytics workloads with fine-grained synchronization, relaxed atomics, and more general sharing patterns.

To help address these issues, heterogeneous systems have recently begun providing a unified, global address space across CPUs and accelerators (primarily GPUs) to improve programmability. A global address space makes it easier to program heterogeneous systems because explicit copies are no longer needed; however it is often inefficient, especially for emerging applications with fine-grained synchronization or relaxed atomics. Heterogeneous systems also use specialized memories that improve efficiency but are difficult to program because they are not part of the global address space. Subsequent work improves efficiency, but often hurts programmability. This thesis redesigns the memory hierarchy of heterogeneous systems to make it more efficient and easier to use by addressing the following issues:

- **Efficient Coherence and Consistency for Heterogeneous Systems**: Our work is the first to identify simple cache coherence protocols (e.g., GPU-style coherence) and complex memory consistency models (e.g., HRF) as contributors to the tension between efficiency and programmability in heterogeneous systems, especially for emerging applications with fine-grained synchronization such as graph analytics workloads. We show that HRF's complexity is not necessary to obtain high performance in heterogeneous systems. Instead, we demon-

strate that the DeNovoA cache coherence protocol, combined with the simpler, traditional DRF memory consistency model, provides equivalent or better performance to GPU coherence with an HRF memory consistency model across a wide range of workloads. DeNovoA is a hardware-driven software cache coherence protocol that self-invalidates valid data and obtains ownership for dirty data and atomics. Moreover, DeNovoA is close in simplicity to conventional accelerator coherence protocols.

- **Efficient Support for and Evaluation of Relaxed Atomics**: DRF0 provides high performance and programmability for most applications. However, DRF0's overheads are too stringent for some applications. This lead to the introduction of relaxed atomics, which can be reordered with all other memory operations to improve performance – at the cost of potentially violating DRF0's SC semantics. Unfortunately, providing acceptable formal semantics for relaxed atomics is a long-standing open problem that has plagued the concurrency community. As a result, it is extremely difficult for programmers to use relaxed atomics correctly. We propose a new memory consistency model, DRFrlx, that provides SC-centric semantics for all common uses of relaxed atomics in heterogeneous systems while retaining their efficiency benefits.

- **Integrating Specialized Memories Into the Unified Address Space**: Heterogeneous systems use specialized memory like scratchpads to improve data organization and movement. Scratchpads are software-managed, directly addressable and provide compact storage. However, because scratchpads exist in a private address space, they are difficult to program and inefficient for some applications, such as those with inter-kernel reuse, fine-grained sharing, or irregular access patterns. In comparison, hardware-managed caches are easy to use and transparent to the programmer, but power-inefficient. We propose a new memory organization, **stash**, that integrates scratchpads into the unified global address space and makes them coherent by extending the low overhead,efficient DeNovoA protocol. Stash retains the benefits of conventional scratchpads, improves their efficiency, and makes them applicable to a wider range of applications.

- **HeteroSync: Benchmark Suite for Fine-Grained Synchronization**: Recent work on

coherence and consistency for heterogeneous CPU-GPU systems has explored how to provide better efficiency for these emerging applications. However, there is a lack of standardization across these papers, which makes it hard to compare their proposed schemes. To resolve this issue, we created HeteroSync. HeteroSync combines includes microbenchmarks implementing various synchronization primitives, annotations for locally and globally scoped atomics, and relaxed atomics. Researchers can use HeteroSync to easily compare algorithm scalability, coherence protocols, and consistency models for heterogeneous systems.

Overall, this thesis makes fundamental contributions to the design of the memory hierarchy of future heterogeneous systems by showing how to provide both efficiency and programmability for emerging workloads *without affecting the efficiency of traditional heterogeneous workloads*. Furthermore, this work makes it easier for programmers to use heterogeneous systems and take advantage of the benefits specialization provides.

7.2 Future Directions

This thesis also enables several interesting future directions for research. Some of these directions are:

7.2.1 Coherence

Although DeNovoA combined with a DRF consistency model improves performance and reduces energy for many applications, there are certain applications where DeNovoA hurts performance or energy. For example, in Section 3.6, DeNovoA hurts performance or energy for applications with low synchronization reuse (BC-1) or high contention (Flags, HG-NO). When contention is high, DeNovoA may suffer from LLC pressure, since its LLC is inclusive for registered variables. Similarly, in Section 2.7, DeNovoA hurts performance slightly for applications with streaming access patterns: BP, LUD, NN, SRAD, and Lava. DeNovoA with a DRF memory model is also outperformed when applications have data that is read multiple times (across synchronization points) before it is written: SS_L, SSBO_L, and UTS. Although HRF's locally scoped synchronization allow this data to remain in the cache longer, DeNovoA with a DRF consistency model must conservatively

invalidate this data at each acquire to avoid reading stale values.

As discussed in Section 6.1.2, hLRC partially addresses these shortcomings by only obtaining ownership for synchronization variables and writing written data through to the LLC [15]. However, hLRC does not allow written data to be reused locally. Since applications like BC and PageRank see significant benefits from obtaining ownership for written data, retaining this feature is important. By optimizing DeNovoA, one could avoid conservative invalidations and reuse written data by combining the hLRC scheme for avoiding conservative invalidations with additional optimizations to DeNovoA:

Adaptively Perform Synchronization Accesses at Different Levels of the Memory Hierarchy: Obtaining ownership for synchronization accesses is usually beneficial. When synchronization reuse is low or contention is high, the additional latency of obtaining ownership from a remote L1 hurts performance and energy compared to performing all synchronization accesses at the LLC. To resolve this issue, one could adaptively perform synchronization accesses at either the L1 or the LLC in DeNovoA. This scheme avoids caching data that is unlikely to be reused. To determine which level of the memory hierarchy synchronization accesses should be performed at, the scheme could exploit information from the software and hardware. At the software level, the programmer can provide a hint to the hardware about which level of the memory hierarchy to perform the synchronization accesses at.

Exploiting software information incurs little hardware overhead, but requires the programmer to understand the applications' access pattern. Also, it is possible that the software information could be too coarse-grained to express which variables see synchronization reuse. Hardware can provide a more fine-grained approach by keeping track of the number of times a given synchronization variable has been reused (or, alternatively, evicted before it could be reused). If a hardware counter indicates that this variable is often not reused before being evicted, then DeNovoA should perform the synchronization access at the LLC instead of obtaining ownership. Although the hardware approach offers more fine-grained control, it also incurs more area and energy overhead.

Write Through Written, Streaming Data Accesses: Obtaining ownership for written data is also inefficient when data reuse is low. One could remove this inefficiency by using the above approach for synchronization variables to identify data that is unlikely to benefit from reuse and

write through this data to the LLC instead of obtaining ownership for it. Avoiding caching this data will improve performance for streaming applications by avoiding the additional latency of accessing this data from a remote L1 cache and allowing the L1 caches to cache other data that is more likely to be reused. Moreover, unlike the recent work discussed above, this scheme allows written data that benefits from reuse to remain in the L1 cache.

Non-Inclusive LLC: When lots of data is owned by the local L1 caches, DeNovoA may prematurely evict some of this data due to LLC pressure. Making DeNovoA's LLC non-inclusive for written data will remove this overhead. To use a non-inclusive LLC, DeNovoA must keep track of the owner of given cache lines that are currently owned by an L1. One way to do this would be to keep a directory next to the non-inclusive LLC but this will require careful analysis to identify the appropriate tradeoff between area, energy, and performance.

7.2.2 Consistency

DRFrlx represents a major step forward for consistency models in heterogeneous systems because it provides SC-centric, formalized semantics for relaxed atomics that retain the efficiency benefits of relaxed atomics.

CPU Memory Models: In this thesis, we focused on how relaxed atomics are used in heterogeneous systems because atomics in heterogeneous systems are far more expensive than in multi-core CPUs. Although multi-core CPUs use sophisticated coherence protocols like MESI to help hide the benefits of relaxed atomics, some CPU applications still see benefits from them. Thus, an important future direction would be to further analyze how relaxed atomics are used in multi-core CPU applications to make sure that DRFrlx covers all common uses of relaxed atomics in CPUs.

Adoption in Programming Languages: To integrate DRFrlx into major programming languages like C, C++, OpenCL, and HSA will require some additional cross-cutting research. For example, the new relaxed atomics categories DRFrlx introduces require changing the memory ordering tags provided by the language, as discussed in Section 3.3.6. Moreover, adoption of DRFrlx will also require discussion with the standards committees.

7.2.3 Specialized Memories

Applying Stash Concepts to Additional Specialized Memories: Caches and scratchpads are only two of the memory organizations used in today's heterogeneous systems. Other specialized private memory structures such as FIFOs, stream buffers, and vector registers suffer from similar inefficiencies as scratchpads. Like scratchpads, these memory organizations require inefficient, explicit data movement from the coherent global address space. For example, the Neural Processing Unit (NPU) uses FIFOs to explicitly transfer data between the global address space and its private NPU address space [59]. This explicit data movement is inefficient and requires significant programmer involvement.

To resolve this inefficiency, one could apply the ideas underlying the stash architecture to integrate other specialized, private memories into the unified, coherent, global address space without losing the benefits of specialization. By making these specialized, private memories globally addressable and coherent, data can be moved efficiently throughout the entire memory hierarchy without explicit data movement. Since DeNovoA only transfers needed data, it can also support specialized memories with different bit widths by taking advantage of the stash's ability to scatter and gather data. Moreover, making these specialized memories part of the global address space opens up additional opportunities for optimizing the movement of data through the memory hierarchy. For example, DeNovoA can exploit the regular data access patterns that many accelerator applications have to perform direct transfers between CUs (producer-consumer style) at the appropriate transfer granularity instead of suffering directory indirections.

Stash-Aware Scheduling Algorithms: One new use case the stash enables is inter-kernel reuse of stash data. Since many GPU applications have highly regular access patterns and exhibit temporal and spatial locality, this represents an opportunity to improve performance and reduce energy consumption by reducing the number of misses an application has. However, schedulers for modern heterogeneous systems are unable to explicitly take advantage of this because they are not aware of where data is located at the end of a kernel. Thus, another interesting future direction is to optimize scheduling algorithms to make them aware of the stash.

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 Mané, 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 Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Whitepaper.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, pages 265–283. USENIX, November 2016.

[3] Parosh Aziza Abdulla, Mohamed Faouzi Atig, Stefanos Kaxiras, Carl Leonardsson, Alberto Ros, and Yunyun Zhu. *Fencing Programs with Self-Invalidation and Self-Downgrade*, pages 19–35. Formal Techniques for Distributed Objects, Components, and Systems: 36th IFIP WG 6.1 International Conference. Springer International Publishing, Cham, 2016.

[4] Sarita Adve and Mark Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA, 1990.

[5] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin, Madison, December 1993.

[6] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, pages 90–101, August 2010.

[7] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, December 1996.

[8] Sarita V. Adve and Mark D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, pages 613–624, June 1993.

[9] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F. Wenisch, John Danskin, and Stephen W. Keckler. Selective GPU caches to eliminate CPU-GPU HW Cache Coherence. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA, pages 494–506, March 2016.

[10] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 33–42, 2009.

[11] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA, New York, NY, USA, 2000. ACM.

[12] Masab Ahmad and Omer Khan. GPU Concurrency Choices in Graph Analytics. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 1–10, September 2016.

[13] Jade Alglave, Mark Batty, Alastair F Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency Weak behaviours and programming assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, New York, NY, USA, 2015. ACM.

[14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, July 2014.

[15] Johnathan Alsop, Bradford Beckmann, Marc Orr, and David Wood. Lazy Release Consistency for GPUs. In *Proceedings of the 49th International Symposium on Microarchitecture*, MICRO, 2016.

[16] Johnathan Alsop, Matthew D. Sinclair, Rakesh Komuravelli, and Sarita V. Adve. GSI: A GPU Stall Inspector to Characterize the Sources of Memory Stalls for Tightly Coupled GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 172–182, April 2016.

[17] Lluc Alvarez, Luis Vilanova, Miquel Moreto, Marc Casas, Marc Gonzalez, Xavier Martorell, Nacho Navarro, Eduard Ayguade, and Mateo Valero. Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, 2015.

[18] AMD. Compute Cores. `https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf`.

[19] AMD. Sea Islands Series Instruction Set Architecture. `http://developer.amd.com/wordpress/media/2013/07/AMD_Sea_Islands_Instruction_Set_Architecture.pdf`, February 2013.

[20] Aaron Ariel, Wilson W.L. Fung, Andrew E. Turner, and Tor M. Aamodt. Visualizing Complex Dynamics in Many-Core Accelerator Architectures. In *IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS, pages 164–174, March 2010.

[21] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratchpad-based Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

[22] David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. *Benchmarking for Graph Clustering and Partitioning*, pages 73–82. Springer New York, New York, NY, 2014.

[23] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge Workshop. *American Mathematical Society*, 7:210–223, 2013.

[24] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 163–174, April 2009.

[25] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES, pages 73–78, New York, NY, USA, 2002. ACM.

[26] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Arkaprava Basu, Sooraj Puthoor, Shuai Che, and Bradford M. Beckmann. Software Assisted Hardware Cache Coherence for Heterogeneous Processors. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS, pages 279–288, New York, NY, USA, 2016. ACM.

[28] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC Atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 634–648, New York, NY, USA, 2016. ACM.

[29] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The Problem of Programming Language Concurrency Semantics. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Berlin Heidelberg, 2015.

[30] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[31] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded Up Robust Features. In Alex Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV*, volume 3951 of *Lecture Notes in Computer Science*, pages 404–417. Springer Berlin Heidelberg, 2006.

[32] Hans-J. Boehm. How to Miscompile Programs with "Benign" Data Races. In *3rd USENIX Workshop on Hot Topics in Parallelism*, HotPar, pages 1–6, June 2011.

[33] Hans-J. Boehm. Can Seqlocks Get Along with Programming Language Memory Models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC, pages 12–20, New York, NY, USA, 2012. ACM.

[34] Hans-J. Boehm. N3710: Specifying the absence of "out of thin air" results (LWG2265). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3710.html, 2013.

[35] Hans-J. Boehm. N3786: Prohibiting "out of thin air" results in C++14. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm, 2013.

[36] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 68–78, New York, NY, USA, 2008. ACM.

[37] Hans-J. Boehm and Brian Demsky. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC, pages 7:1–7:6, New York, NY, USA, 2014. ACM.

[38] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 141–151, Nov 2012.

[39] C++. C++ Reference: Memory Order. http://en.cppreference.com/w/cpp/atomic/memory_order, 2015.

[40] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA, pages 70–79, Washington, DC, USA, 1999. IEEE Computer Society.

[41] Daniel Cederman, Bapi Chatterjee, and Philippas Tsigas. Understanding the Performance of Concurrent Data Structures on Graphics Processors. In *Proceeedings of 18th International Euro-Par Conference on Parallel Processing*, pages 883–894, 2012.

[42] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 185–195, Sept 2013.

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

[44] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 1–11, 2010.

[45] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 13:1–13:11, New York, NY, USA, 2011. ACM.

[46] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 155–166, 2011.

[47] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. BiN: A buffer-in-NUCA Scheme for Accelerator-rich CMPs. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED, pages 225–230, New York, NY, USA, 2012. ACM.

[48] Jason Cong, Karthik Gururaj, Hui Huang, Chunyue Liu, Glenn Reinman, and Yi Zou. An Energy-efficient Adaptive Hybrid Cache. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED, pages 67–72, Piscataway, NJ, USA, 2011. IEEE Press.

[49] Henry Cook, Krste Asanovic, and David A. Patterson. Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments. Technical report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2009.

[50] Michael Cowgill. Speeded Up Robustness Features (SURF), December 2009.

[51] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU, pages 63–74, New York, NY, USA, 2010. ACM.

[52] Mahdad Davari, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. The Effects of Granularity and Adaptivity on Private/Shared Classification for Coherence. *ACM Transactions on Architecture and Code Optimization*, 12(3):26:1–26:21, August 2015.

[53] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011.

[54] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, Feb 2012.

[55] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of The 33rd International Conference on Machine Learning*, ICML, pages 2024–2033, 2016.

[56] DoE. Top Ten Exascale Research Challenges. `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf`, February 2014.

[57] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[58] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA, pages 365–376, New York, NY, USA, 2011. ACM.

[59] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[60] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K. Govindaraju. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):608–620, April 2011.

[61] Wilson W. L. Fung and Tor M. Aamodt. Energy Efficient GPU Transactional Memory via Space-time Optimizations. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 408–420, New York, NY, USA, 2013. ACM.

[62] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 296–307, New York, NY, USA, 2011. ACM.

[63] Benedict R. Gaster, Derek Hower, and Lee Howes. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Transactions on Architecture and Code Optimizations*, 12(1):7:1–7:26, April 2015.

[64] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[65] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA, pages 15–26, New York, NY, USA, 1990. ACM.

[66] James R Goodman. Cache Consistency and Sequential Consistency. Technical report, SCI Committee, March 1989.

[67] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 1–10, New York, NY, USA, 2015. ACM.

[68] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a High-level Language Targeted to GPU Codes. In *Innovative Parallel Computing*, InPar, pages 1–10, May 2012.

[69] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Victor García-Flores, Simon G. de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 43–54, April 2017.

[70] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 260–269, New York, NY, USA, 2008. ACM.

[71] 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 *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA, pages 189–200, Feb 2014.

[72] Blake A Hechtman and Daniel J. Sorin. Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 118–119, April 2013.

[73] Blake A. Hechtman and Daniel J. Sorin. Exploring Memory Consistency for Massively-threaded Throughput-oriented Processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA, pages 201–212, New York, NY, USA, 2013. ACM.

[74] Joel Hestness, Stephen W. Keckler, and David A. Wood. GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 87–97, October 2015.

[75] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[76] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 349–362, New York, NY, USA, 2012. ACM.

[77] 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 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 427–440, New York, NY, USA, 2014. ACM.

[78] HSA Foundation. HSA Platform System Architecture Specification. `http://www.hsafoundation.com/?ddownload=4944`, 2015.

[79] IntelPR. Intel Discloses Newest Microarchitecture and 14 Nanometer Manufacturing Process Technical Details. *Intel Newsroom*, 2014.

[80] D. Anoushe Jamshidi, Mehrzad Samadi, and Scott Mahlke. D2MA: Accelerating Coarse-grained Data Transfer for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT, pages 431–442, New York, NY, USA, 2014. ACM.

[81] Alan Jeffrey and James Riely. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS, pages 759–767, New York, NY, USA, 2016. ACM.

[82] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, pages 1–12, New York, NY, USA, 2017. ACM.

[83] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 175–189, New York, NY, USA, 2017. ACM.

[84] Stefanos Kaxiras and Georgios Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54–65, September 2010.

[85] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-cache Coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA, pages 535–546, New York, NY, USA, 2013. ACM.

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

[87] Ji Yun Kim and Christopher Batten. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 75–87, Dec 2014.

[88] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. Automatic Datatype Generation and Optimization. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 327–328, New York, NY, USA, 2012. ACM.

[89] Rakesh Komuravelli. *Exploiting Software Information for an Efficient Memory Hierarchy*. PhD thesis, University of Illinois at Urbana-Champaign, December 2014.

[90] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. Revisiting the Complexity of Hardware Cache Coherence and Some Implications. *ACM Transactions on Architecture and Code Optimization*, 11(4):37:1–37:22, December 2014.

[91] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Prakalp Srivastava, Maria Kotsifakou, Sarita V. Adve, and Vikram S. Adve. Stash: Have Your Scratchpad and Cache it Too. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, pages 707–719, 2015.

[92] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead. *ACM Transactions on Architecture and Code Optimization*, 13(1):1:1–1:22, March 2016.

[93] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. Fusion: Design Tradeoffs in Coherence Cache Hierarchies for Accelerators. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, pages 733–745, 2015.

[94] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming Release-Acquire Consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 649–662, New York, NY, USA, 2016. ACM.

[95] Shin-Ying Lee and Carole-Jean Wu. CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 175–186. ACM, 2014.

[96] Lee Howes and Aaftab Munshi. The OpenCL Specification, Version 2.0. Khronos Group, 2015.

[97] 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 40th Annual International Symposium on Computer Architecture*, ISCA, pages 487–498, New York, NY, USA, 2013. ACM.

[98] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS, pages 109–118, New York, NY, USA, 2015. ACM.

[99] Chao Li, Yi Yang, Dai Hongwen, Yan Shengen, Frank Mueller, and Huiyang Zhou. Understanding the Tradeoffs Between Software-Managed vs. Hardware-Managed Caches in GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 231–242, 2014.

[100] 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 *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 469–480, December 2009.

[101] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 635–646, December 2014.

[102] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 233–247, New York, NY, USA, 2016. ACM.

[103] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA, pages 388–400, New York, NY, USA, 2015. ACM.

[104] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 661–675, New York, NY, USA, 2017. ACM.

[105] Daniel J. Lustig. *Specifying, Verifying, and Translating Between Memory Consistency Models*. PhD thesis, Princeton University, November 2015.

[106] Michael J. Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. The Accelerator Store: A Shared Memory Framework for Accelerator-Based Systems. *ACM Transactions on Architecture and Code Optimization*, 8(4):48:1–48:22, January 2012.

[107] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLcheck: Verifying the Memory Consistency of RTL Designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 463–476, New York, NY, USA, 2017. ACM.

[108] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. In *48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 26–37, Dec 2015.

[109] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32nd Symposium on Principles of Programming Languages*, POPL, pages 378–391, New York, NY, USA, 2005. ACM.

[110] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[111] Paul McKenney. Some Examples of Kernel-Hacker Informal Correctness Reasoning. In *Proceedings of the Dagstuhl Workshop on Compositional Verification Methods for Next-Generation Concurrency*, 2015.

[112] Paul E. McKenney, Torvald Riegel, and Jeff Preshing. N4036: Towards Implementation and Use of memory_order_consume. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4036.pdf`, 2014.

[113] Ugljesa Milic, Issac Gelado, Nikola Puzovic, Alex Ramirez, and Milo Tomasevic. Parallelizing General Histogram Application for CUDA Architectures. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS, pages 11–18, July 2013.

[114] David S. Miller. Semantics and Behavior of Atomic and Bitmask Operations. `https://01.org/linuxgraphics/gfx-docs/drm/core-api/atomic_ops.html`, 2016.

[115] Prabhakar Misra and Mainak Chaudhuri. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In *IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS, pages 53–60, December 2012.

[116] Andrew Nere, Atif Hashmi, and Mikko Lipasti. Profiling Heterogeneous Multi-GPU Systems to Accelerate Cortically Inspired Learning Algorithms. In *IEEE International Parallel Distributed Processing Symposium*, IPDPS, pages 906–920, May 2011.

[117] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Memory Allocation for Embedded Systems with a Compile-time-unknown Scratchpad Size. *ACM Transactions on Embedded Computing Systems*, 8(3):1–32, April 2009.

[118] NVIDIA. CUDA SDK 3.1. `http://developer.nvidia.com/object/cuda_3_1_downloads.html`, 2010.

[119] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In George Almasi, Caalin Cascaval, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 235–250. Springer Berlin Heidelberg, 2007.

[120] Molly A O'Neil and Martin Burtscher. Microarchitectural Performance Characterization of Irregular GPU kernels. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 130–139, Oct 2014.

[121] Marc S Orr, Shuai Che, Ayse Yilmazer, Bradford M Beckmann, Mark D Hill, and David A Wood. Synchronization Using Remote-Scope Promotion. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, New York, NY, USA, 2015. ACM.

[122] 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 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 743–758, New York, NY, USA, 2014. ACM.

[123] Jean Pichon-Pharabod and Peter Sewell. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 622–633, New York, NY, USA, 2016. ACM.

[124] Victor Podlozhnyuk. Histogram calculation in CUDA. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf`, 2007.

[125] 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 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 457–467, New York, NY, USA, 2013. ACM.

[126] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[127] Jing Pu, Steven Bell, Yang Xuan, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *CoRR*, abs/1610.09405, 2016.

[128] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Transactions on Graphics*, 31(4):32:1–32:12, July 2012.

[129] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 519–530, New York, NY, USA, 2013. ACM.

[130] Carl G. Ritson and Scott Owens. Benchmarking Weak Memory Models. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 24:1–24:11, New York, NY, USA, 2016. ACM.

[131] Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. Hierarchical Private/Shared Classification: The Key to Simple and Efficient Coherence for Clustered Cache Hierarchies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA, pages 186–197, Feb 2015.

[132] Alberto Ros and Stefanos Kaxiras. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 241–252, New York, NY, USA, 2012. ACM.

[133] Alberto Ros and Stefanos Kaxiras. Callback: Efficient Synchronization Without Invalidation with a Directory Just for Spin-waiting. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA, pages 427–438, New York, NY, USA, 2015. ACM.

[134] Alberto Ros, Carl Leonardsson, Christos Sakalis, and Stefanos Kaxiras. POSTER: Efficient Self-Invalidation/Self-Downgrade for Critical Sections with Relaxed Semantics. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT, pages 433–434, New York, NY, USA, 2016. ACM.

[135] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming Model for a Heterogeneous x86 Platform. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, pages 431–440, New York, NY, USA, 2009. ACM.

[136] Andreas Sembrant, Erik Hagersten, and David Black-Shaffer. TLC: A Tag-less Cache for Reducing Dynamic First Level Cache Energy. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 49–61, New York, NY, USA, 2013. ACM.

[137] Matthew Sinclair, Henry Duwe, and Karthikeyan Sankaralingam. Porting CMP Benchmarks to GPUs. Technical report, Department of Computer Sciences, The University of Wisconsin-Madison, 2011.

[138] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 647–659, December 2015.

[139] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, pages 161–174, New York, NY, USA, 2017. ACM.

[140] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs. In *IEEE International Symposium on Workload Characterization*, IISWC, October 2017.

[141] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache Coherence for GPU Architectures. In *19th International Symposium on High Performance Computer Architecture*, HPCA, pages 578–590, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[142] Richard Smith. N4659: Working Draft, Standard for Programming Language C++. `https://github.com/cplusplus/draft/blob/master/papers/n4659.pdf`, 2017.

[143] Tyler Sorensen, Jade Alglave, Ganesh Gopalakrishnan, and Vinod Grover. ICS: U: Towards Shared Memory Consistency Models for GPUs. In *Towards Shared Memory Consistency Models for GPUs*, ICS, pages 489–490, 2013.

[144] Tyler Sorensen and Alastair F. Donaldson. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 100–113, New York, NY, USA, 2016. ACM.

[145] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, Department of ECE and CS, University of Illinois at Urbana-Champaign, March 2012.

[146] Jeff A. Stuart and John D. Owens. Efficient Synchronization Primitives for GPUs. *CoRR*, abs/1110.4623, 2011.

[147] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems*, 13(4s):134:1–134:25, April 2014.

[148] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, pages 145–154, New York, NY, USA, 2013. ACM.

[149] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and Reuse with Compiled Domain-specific Languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP, pages 52–78, Berlin, Heidelberg, 2013. Springer-Verlag.

[150] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter Mccardwell, Alejandro Villegas, and David Kaeli. Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 1–10, September 2016.

[151] Hyojin Sung. *DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism.* PhD thesis, University of Illinois at Urbana-Champaign, July 2015.

[152] Hyojin Sung and Sarita V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 545–559, 2015.

[153] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-determinism. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 13–26, New York, NY, USA, 2013. ACM.

[154] Herb Sutter. Atomic Weapons: The C++ Memory Model and Modern Hardware. In *C++ and Beyond*, 2012.

[155] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 119–133, New York, NY, USA, 2017. ACM.

[156] Sumesh Udayakumaran and Rajeev Barua. Compiler-decided Dynamic Memory Allocation for Scratchpad Based Embedded Systems. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES, pages 276–286, New York, NY, USA, 2003. ACM.

[157] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic Allocation for Scratchpad Memory Using Compile-time Decisions. *ACM Transactions on Embedded Computing Systems*, 5(2):472–511, May 2006.

[158] Viktor Vafeiadis and Chinmay Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, pages 867–884, New York, NY, USA, 2013. ACM.

[159] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. Remote-Scope Promotion: Clarified, Rectified, and Verified. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 731–747, New York, NY, USA, 2015. ACM.

[160] Shucai Xiao and Wu-chun Feng. Inter-block GPU Communication via Fast Barrier Synchronization. In *IEEE International Symposium on Parallel Distributed Processing*, IPDPS, pages 1–12, April 2010.

[161] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph Processing on GPUs: Where are the Bottlenecks? In *IEEE International Symposium on Workload Characterization*, IISWC, pages 140–149, October 2014.

[162] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. Lock-based Synchronization for GPU Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF, pages 205–213, New York, NY, USA, 2016. ACM.

[163] Guowei Zhang, Webb Horn, and Daniel Sanchez. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-coherent Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO, pages 13–25, New York, NY, USA, 2015. ACM.

[164] Zhong Zheng, Zhiying Wang, and Mikko Lipasti. Tag Check Elision. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED, pages 351–356, New York, NY, USA, 2014. ACM.

[165] Amir Kavyan Ziabari, Yifan Sun, Yenai Ma, Dana Schaa, José L. Abellán, Rafael Ubal, John Kim, Ajay Joshi, and David Kaeli. UMH: A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs. *ACM Transactions on Architecture and Code Optimization*, 13(4):35:1–35:25, December 2016.