Reducing Synchronization and Communication Overheads in GPUs

by

Preyesh Dalmia

A dissertation for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

Academic Advisor:

Matthew D. Sinclair, Assistant Professor, Computer Sciences

Oral Examination Date:

07/19/2023

Oral Committee:

Joshua San Miguel, Assistant Professor, Electrical and Computer Engineering

Mikko Lipasti, Professor, Electrical and Computer Engineering

Dan Negrut, Professor, Mechanical Engineering

Michael Boyer, Member of Technical Staff, AMD Research

Dedicated to the people who always believed in me, Mom, Dad, and Didi

ACKNOWLEDGMENTS

My journey here at UW Madison has been life-changing. I have learned a lot of professional and personal lessons that are going to define how I live for the rest of my life. I owe a debt of gratitude to a lot of people, without whom this journey wouldn't have been possible. At the outset, I would like to express my sincere gratitude to my advisor, Prof. Matthew D. Sinclair, for his constant guidance, encouragement, constructive comments, and motivation, which gave concrete shape to my research endeavors. His breadth of knowledge and wide range of skills have always astonished me and continue to impress me. Deep in my heart, I have deep respect for him. To this day, he remains the most hardworking person I have ever met. He will always remain an inspiration to me. He has given me ample opportunities to learn, grow, and explore. Prof. Matt has been instrumental in molding me into what I am today. Much of the work in this thesis is a result of collaboration with Rohan Mahapatra, a teammate who has been a crucial partner in my academic and research journey at UW Madison. I deeply appreciate the hours we spent brainstorming, implementing, and debugging projects. I am thankful to all the members of the Heterogeneous Architecture Lab Research Group with whom I have either directly or indirectly collaborated. They have been especially excellent in providing feedback on presentations and in relinquishing their Linux jobs to provide me with extra slots during conference deadlines. I am also grateful to the faculty of both the ECE and CS departments at the University of Wisconsin-Madison for their all-out help and academic guidance, which helped me keep the momentum of my academic journey. Last but not least, a huge thanks goes to my family, especially my father, mother, and sister, who have always been a constant source of encouragement and boosted my morale with frequent calls despite the time zone difference across continents. Finally, a very special thanks to all my friends who have been instrumental in helping me get my PhD degree. On, Wisconsin!

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As programmable GPUs have become increasingly general-purpose, they are increasingly used by a wide variety of applications that leverage them for accelerated computing. This has resulted in them outgrowing their traditional use as accelerators attached to CPUs. Although GPUs continue to be used for streaming, data-parallel applications such as gaming, video editing, graphics rendering, and data mining they have now become the default platform for an ever-growing list of applications such as big data processing, deep learning, and graph analytics. However, workloads that traditionally ran on these systems have different characteristics than these newer workloads. For example, traditional workloads generally assume threads access independent data and synchronize infrequently. Accordingly, accelerators such as GPUs use simple, software-driven coherence protocols that tradeoff heavyweight synchronization operations for improved efficiency when synchronization is not required. To help reduce this overhead, accelerators also use scoped memory consistency models. Scopes enable cheap, local synchronization if the synchronizing threads are in the same thread block, but when this is not the case, heavyweight operations are performed to ensure correctness.

While this approach worked well for traditional applications, many modern applications that run on GPUs frequently share data across threads and utilize fine grained-synchronization. Thus, inefficient synchronization support is a significant bottleneck for running them on these accelerators. Thus, a holistic approach is required to tackle the inefficiencies in how synchronization is used in both single- and Multi-GPU systems. We propose hardware-software frameworks that use knowledge of the GPU memory hierarchy and algorithmic properties of applications to improve the efficiency of GPU global synchronization. First we target bottlenecks in explicit global synchronization resulting from the use of atomics for global memory updates. To resolve this bottleneck, we propose to cache commutative atomic updates locally using a novel buffering mechanism that exploits locality in atomics and reduces their serialization penalty, which reduces network traffic to the LLC and improves performance. Programmers also use global synchronization to ensure correctness with software synchronization primitives. However, existing GPU synchronization primitives either scale poorly or suffer from livelock or deadlock issues because of heavy contention between threads accessing shared synchronization objects. We overcome these inefficiencies by designing more efficient, scalable GPU synchronization primitives. Finally, we handle performance degradation that results from implicit global synchronization that takes place at kernel boundaries. GPU vendors are pivoting to chiplet-based designs where the global memory ordering point has moved from the L2 to the L3 or global memory. This necessitates bulk cache invalidations/writeback of the L2 caches in these chiplets, leading to a loss of potential inter-kernel data reuse. We propose an intelligent producer-consumer dependency tracking mechanism called CPCoh that reduces the number of bulk coherence maintenance operations required, thereby increasing inter-kernel reuse and improving performance. Overall we advance the state of the art for global synchronization in GPUs resulting in performance and energy improvements across a plethora of modern GPU applications.

1 INTRODUCTION

Traditionally general-purpose programmable accelerators such as GPUs were used for streaming, data parallel workloads with limited data reuse, or data sharing with coarse-grained synchronization that usually only happened at kernel boundaries. Recently, general-purpose GPUs (GPGPUs) have become the accelerator of choice for a wide range of applications including machine learning, graph analytics, computational biology and data mining. These newer GPGPU applications rely on fine-grained synchronization and data sharing for higher performance. However, GPU's massively parallel processing cores coupled with simple, software-driven coherence protocols and scoped consistency models provide challenges for efficient synchronization at various granularities. Unlike multi-core CPUs, which have significant OS support and complex coherence protocols that make synchronization relatively cheap, GPUs have limited OS support and simple, software-driven coherence protocols which make synchronization expensive. GPUs tradeoff inefficient synchronization support for more efficient regular data accesses. This tradeoff negatively affects performance when synchronization becomes more frequent, as it requires heavyweight actions at synchronization points to ensure correctness [6, 73, 81, 118, 203, 204, 206]. To partially reduce this overhead, GPU memory consistency models utilize scoped synchronization, which allows programmers to specify the scope at which threads must synchronize [67, 85–87, 136]. Scopes allow synchronization and coherence to be contained within a narrow subset of the memory hierarchy, where the level of the scope impacts the synchronization cost. For example in a scope-based memory consistency model, if the synchronizing threads belong to the same thread block (TB), then the synchronization occurs locally with reduced overhead. However, expensive global synchronization is still required if the synchronizing threads are in different TBs, which is often the case for these workloads. Global synchronization in GPUs has a variety of different use cases, which expose different challenges with its efficiency. This work takes a holistic look at global synchronization and proposes solutions to mitigate most of the important inefficiencies that result from it. In particular, we focus on three key types of synchronization: device-scoped relaxed atomics, software synchronization primitives, and implicit kernel-boundary synchronization.

- **Global synchronization through device-scoped atomics:** Many modern GPU applications utilize device-scope relaxed atomics – relaxed atomics imply no ordering on other memory accesses to update shared global variables. Device-scope accesses go to the first common ordering point, which for monolithic GPUs is the L2 cache. However, since L2 accesses in modern GPUs take over 100 cycles (Section 3.3), even relaxed, device-scope atomics are expensive which makes their use to update shared global variables inefficient. Recent work has shown that these atomic updates are a large source of inefficiency in ML training [125] and graph analytics [4, 226]. To overcome this inefficiency we use hardware-software co-design to reduce atomic latency, data movement, and energy [46]. At the software level, we exploit algorithmic properties; recent work identified these graph application use *commutative* atomics: the order of commutative atomics does not matter since the program does not read the updated values until all updates have completed [5, 27, 204, 242]. Exploiting this commutativity, at the hardware level we buffer partial device-scope atomic updates locally at each SM in a small *local atomic buffer* (*LAB*), before sending coalesced updates to the shared L2 later. This enables LAB to coalesce commutative atomic updates across all TBs on an SM, alleviating the impact of global atomic updates and improving performance by 28%, energy by 19%, and network traffic by 19% on average over the baseline GPU architecture while also outperforming state-of-the-art techniques such as hLRC [6] and PHI [146]. This work was published in the **International Symposium on High-Performance Computer Architecture (HPCA) 2022** [46].

- **Global synchronization through explicit software primitives:** Some GPU algorithms cannot rely on relaxed atomics to shared global variables and need stronger ordering guarantees at synchronization points. Thus, these applications rely on other forms of explicit synchronization, including barriers and semaphores, to avoid data races and ensure correctness when threads in different thread blocks or work groups are accessing the same data [78, 205, 208, 212] to avoid data races and ensure correctness. However, device-wide synchronization is prohibitively expensive and is often a bottleneck for emerging workloads that utilize it (discussed further in Section 4.4). Consequently, as GPU algorithms scale to thousands of threads, existing GPU synchronization primitives either scale poorly or suffer from livelock or deadlock issues because of heavy contention between threads accessing shared synchronization objects. We seek to overcome these inefficiencies by designing more efficient, scalable GPU barriers and semaphores. In particular, we show how multi-level sense reversing barriers and priority mechanisms for semaphores can be designed with the GPUs unique processing model in mind to improve the performance and scalability of GPU synchronization primitives. Overall, across three modern GPUs the proposed barrier algorithm improves performance by an average of 33% over a GPU tree barrier algorithm and improves performance by an average of 34% over CUDA Cooperative Groups [78] for five full-sized benchmarks at high contention levels; the new semaphore algorithm improves performance by an average of 83% compared to prior GPU semaphores. This work was published in **IEEE Transactions on Parallel and Distributed Systems (TPDS 2022)** [45].

- **Implicit global synchronization at kernel boundaries:** Some applications explicitly synchronize (e.g., with atomics) in application phases to ensure correctness when multiple threads are accessing shared data. However, all applications, even those that do not need explicit synchronization, must implicitly synchronize at kernel boundaries. During implicit synchronization, private caches must be invalidated and at the end of kernels, all dirty data must be written back. In monolithic GPUs, the L2 was a common synchronization point across all SMs because it was the LLC. Thus, implicit synchronization only needed to be performed on the L1 caches. However with a recent shift of commercial GPU architectures towards chiplet-based designs [189, 198], there is a need to bulk invalidate/writeback L2 caches that are private to a chiplet and are no longer the common synchronization point. This eliminates the potential for inter-kernel reuse from the L2 cache which GPU applications often rely on for higher performance [105, 239]. The reuse now can only be gained from the L3 cache, making it more expensive. To mitigate this performance impact there is a need to retain data in L2 caches across kernels. Thus, we seek to reduce the need for bulk L2 cache invalidations/writeback operations by proposing CPCoh. CPCoh splits the Command Processor, the interface between the host and the accelerator (Section 2.2.1) into local and global components to improve scalability, then augments them to track producer-consumer dependencies and kernel scheduling information. To do this, CPCoh adds a chiplet coherency table inside the Global Command Processor that has access to both the access modes and the kernel scheduling information that CPCoh requires to operate (discussed further in Chapter 5). This allows applications to retain more data in caches locally, improving average performance by 13% and 14%, energy consumption by 14% and 11%, and network traffic by 19% and 17%, respectively, over current approaches and HMG [189]. This work is in submission at this thesis was published [47, 48].

Next we provide background for our work and then go into detail about our solutions to each of the three pieces of global synchronization that have been discussed here. Overall, all our optimizations combined advance the state of the art for the implementation of global synchronization in general-purpose programmable accelerators.

2 BACKGROUND AND RELATED WORK

This chapter reviews the necessary background materials for the rest of this dissertation. Section 2.1 presents a high-level view of the modern GPU architecture, while Section 2.2 summarizes GPU consistency and coherence, and introduces the Command Processor.

2.1 GPU Architecture

A GPU application starts on the host CPU which launches work on the GPU in the form of kernels. Every time the kernel function is called, CPU will launch it onto the GPU through a software framework such as CUDA [158], HIP [13], or OpenCL [211]. GPU programs consist of one or more kernels of thousands of threads. The thread hierarchy is shown in Figure 2.2. A group of threads that are executed in lockstep and scheduled as a unit is referred to as a warp or a wavefront multiple warps combine to make a thread block or TB (AMD refers to them as work groups [1]). A grid of these thread blocks is used to execute a kernel. Figure 2.1 gives a simplified view of the GPU architecture it consists of Streaming Multiprocessors (SM) (also known as Compute Units) with their own L1 data caches and shared memories. The L2 cache serves as the common ordering point for all SMs which then interfaces with the main memory. Threads within a thread block can communicate via an on-chip scratchpad memory called the shared memory in CUDA (or local data store in OpenCL), and can synchronize via hardware barriers. The SM cores on a GPU access a shared last-level cache and off-chip DRAM memory through an on-chip interconnection network. From a programming perspective, GPU memory is divided into several spaces, each with its own characteristics in terms of performance and semantics. For example, data that is shared only by the threads of a thread block can be stored in the shared memory, while data that is shared by multiple thread blocks must be stored in the global memory.

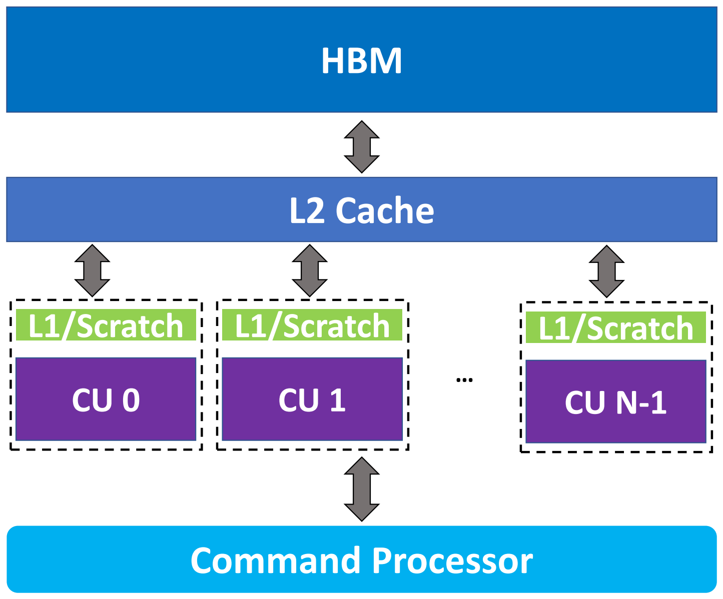


Figure 2.1: Simplified monolithic GPU architecture

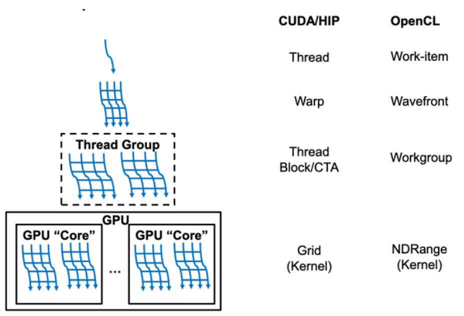


Figure 2.2: GPU thread hierarchy

2.2 GPU Coherence and Consistency

Since GPUs traditionally ran massively data-parallel, streaming applications with coarse-grained synchronization and little to no data reuse, GPU architectures used simple, software-driven Valid-Invalid (VI)-style coherence protocols [118, 189, 203, 204]. Unlike CPU coherence protocols, these protocols did not have ownership requests, downgrade requests, writer-initiated invalidations, state bits, snoopy buses or directories [206].

Typically, GPU L1 caches either use a write through or write no-allocate approach for global memory writes [104, 206]. To improve performance, these writes may be buffered and coalesced until the next store release [81]. When a store release occurs (the end of the kernel or a release synchronization operation), all prior stores must complete and the data is written to the next level of the memory hierarchy, which is usually shared between SMs. Thus, fine-grained synchronization that uses load acquires and/or store releases provides ordering between data and atomic requests from TBs on multiple SMs. For example, sequentially consistent (SC) atomics orders both data and atomic accesses. As GPU coherence protocols do not obtain ownership for written data or atomics, they must perform synchronization accesses (atomics) at the LLC (usually L2), they must flush all dirty data from the store buffer on releases, and they must self-invalidate the entire cache on acquires to ensure a consistent view of memory for variables being accessed by multiple TBs. Atomics are sometimes used as *relaxed* atomics. Relaxed atomics act neither as an acquire nor a release, they imply no ordering on other memory accesses, and they can be reordered with other data and atomic accesses. As a result, relaxed atomics are cheaper. However, since L2 accesses in modern GPUs take over 100 cycles (Section 3.3), even relaxed, device-scope atomics are expensive. In contrast, CPUs often obtain ownership for written data and atomics (e.g., in MOESI-style coherence), which makes synchronization points cheap; however, these protocols are a poor fit for GPUs [81, 206].

Instead GPU consistency models utilize scopes to reduce synchronization overhead, as part of sequentially consistent for heterogeneous-race-free (SC-for-HRF) based consistency models [67, 85–87, 136]. Programs which properly identify both memory accesses as data or synchronization and each synchronization accesses' scope are guaranteed to be SC-for-HRF. Although SC-for-HRF has multiple scopes, we focus on the two most widely used variants: local and device. Locally scoped atomics are only guaranteed to be visible to other threads in the

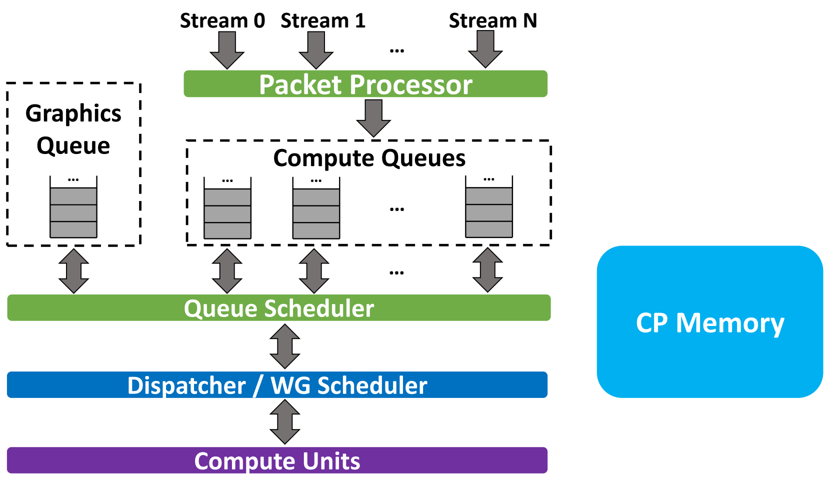


Figure 2.3: GPU Command Processor

same TB, while device-scoped atomics are visible to all threads across the GPU. Thus, locally scoped synchronization is significantly cheaper since it does not invalidate all valid L1 data on acquires nor writes through dirty data on releases.

2.2.1 GPU Command Processors

Figure 2.3 shows a simplified diagram of the overall Command Processor (CP) [10, 73].[1] For GPUs, the programmable CP interfaces between the software, via the driver and runtime (e.g., AMD's ROCm [15] stack), and the hardware. Since CPs are programmable, vendors can adjust their functionality without changing hardware. Once a user has written their GPU program [14, 158, 211], the underlying GPU driver and runtime create software queues and enqueue the program's GPU kernels, along with any memory management and inter-kernel synchronization, as packet(s).

Next, the CP's *packet processor* maps each packet onto one of its hardware *compute queues* using the *queue scheduler*. The CP's *queue scheduler's* goal is to map t kernels to M processing units while maximizing resource utilization. To

[1]Without loss of generality, we use AMD terminology when discussing CPs. NVIDIA utilizes embedded RISC cores for similar purposes [1, 159].

meet this goal, GPUs support multiple hardware queues [10, 131, 156, 182] to manage independent work submitted asynchronously with GPU streams [14, 135, 163], which allow programmers to enqueue work that may be executed in parallel with another stream on the GPU. Normally, each stream is mapped to a queue and each queue holds multiple kernels from a single stream. The CP maintains inter-kernel dependencies between kernels in the same stream but allows kernels from different streams to execute asynchronously. Within a queue, a queue entry describes a separate kernel launch and includes details such as the dimension of threads, register usage, scratchpad size, and pointers to arrays (or other kernel arguments) being accessed. The CP's *Work-Group (WG) scheduler* reads these fields to dispatch WGs to CUs. Generally, WG schedulers issue all WGs from one kernel before switching to issue WGs from a different kernel. Moreover, WG schedulers will do this in round-robin fashion [182] across the available CUs.

3 ONLY BUFFER WHEN YOU NEED TO: REDUCING ON-CHIP GPU TRAFFIC WITH RECONFIGURABLE LOCAL ATOMIC BUFFERS

3.1 Motivation

Applications such as machine learning and graph analytics leverage massive parallel computation available on GPUs to improve their performance. Unlike tradition GPGPU applications, many of these applications rely on data sharing across TB, and in some cases they require fine-grained synchronization. Thus synchronization is necessary to ensure correctness and avoid data races. In GPUs threads in different TBs must use *global*-scoped atomics that are performed at the shared last level cache (LLC, usually the L2) to perform these updates (Section 2.2). However, since GPUs do not efficiently support atomics, this limits scalability (discussed further in Section 2.2). Recent work has shown that these atomic updates are a large source of inefficiency in ML training [125] and graph analytics [4, 226].

To validate these claims, we profiled histograms, graph analytics, and ML training applications (described further in Section 3.3). Figure 3.1 shows that device-scoped atomics make up a significant fraction (29% on average) of their global memory accesses. Thus they can become a bottleneck for the overall system as atomics are relatively expensive operations. However, atomics used in these programs do not imply ordering on surrounding accesses because they are commutatively updating shared variables. Thus, they can safely use lower overhead relaxed atomics (discussed further in Section 2.2). Nevertheless, relaxed atomics are still expensive. Moreover, although applications like BC (Betweenness Centrality) have fewer atomics, these accesses are often a bottleneck because they are serialized [58] – multiple threads from the same TB concurrently, atomically update the same address, which serializes the accesses (discussed further in

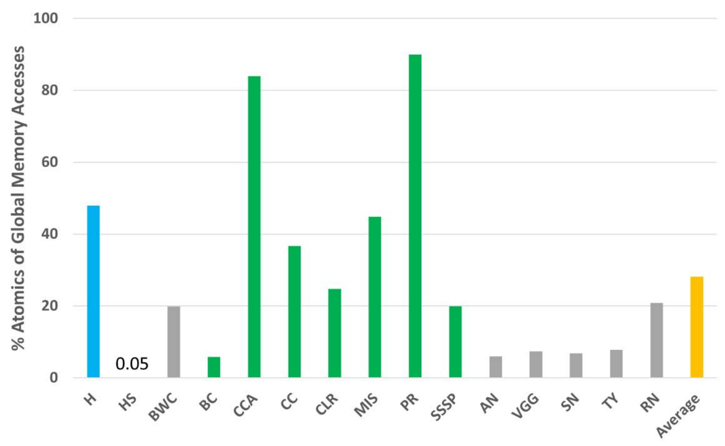


Figure 3.1: Percent of device-scope commutative atomics for histograms (blue), graph analytics (green), and ML training (gray) on a Titan V GPU.

Section 3.2.1). Given the importance of ML Training and graph analytics, prior work has proposed customized solutions for graph analytics [34, 75, 236, 244] and ML training [38, 96, 126, 155, 184, 235]. However, GPUs still remain the most widely used computing platform for these applications due to their availability and ease of programming. Recent work on optimizing ML training on GPUs includes reducing the width and/or number of memory accesses [90, 225, 248], utilizing compression [192], or rewriting code to frequently perform memory accesses in the register file or shared memory [54, 59, 108, 164, 246]. However, Figure 3.1 shows that many of the remaining memory requests are atomics that update shared locations. Accordingly, these atomics represent a significant overhead.

To overcome this inefficiency we propose a hardware-software co-design approach that reduces atomic latency, data movement, and energy. At the software level, we exploit algorithmic properties; recent work identified that graph analytics algorithms often use *commutative* relaxed atomics – i.e., although the accesses must be performed atomically to ensure correctness, the order of the atomics does not matter since the program does not view the updated values until all updates have completed [5, 27, 204, 242]. We find that this property also holds for ML training weight updates: the updated weights are not used until subsequent layers.

Moreover, other, non-commutative relaxed atomics with similar properties can benefit from our approach. At the hardware level, we exploit the commutativity of these atomic updates to buffer partial device-scope atomic updates locally at each SM in a small local atomic buffer (LAB), before sending coalesced updates to the shared L2 later. We propose to extend the partitioning of the unified local memory [68] to include the LAB. This enables LAB to coalesce commutative atomic updates across all TBs on an SM, and improves performance, energy, and network traffic by reducing both the latency for atomic accesses and the number of commutative atomic accesses sent to the L2.

Prior work (discussed in more detail in Section 3.6) also exploits commutativity to improve performance and reduce energy [61]. In particular, recent research like DeNovo and hLRC cache device-scoped atomics in GPU L1 caches [6, 203, 204]. However, they require significant coherence protocol or consistency model changes. Similarly, in multi-core CPUs AIM, CCache, Coup, and PHI exploit commutativity by adding an additional coherence state or in-cache buffers [5, 24, 146, 242]. Although these approaches exploit similar insights, since GPUs use lightweight, software-driven coherence protocols [118, 203, 206] and have high L1 cache contention, our results show these solutions are not ideal fits for GPUs. Instead, LAB shows that using the existing reconfigurable SRAM to separately buffer atomics improves performance and energy efficiency relative to PHI and hLRC (Section 3.4), requires minor software changes (annotating commutative atomic accesses), and does not require coherence protocol or consistency model changes. Thus, LAB provides similar or better benefits than hLRC and PHI, without their downsides.

Overall, across 15 graph analytics and ML training workloads, a small, reconfigurable, per SM LAB improves performance by 28%, reduces energy by 19%, and reduces on-chip traffic by 19% on average, respectively. Moreover, LAB improves on state-of-the-art solutions like hLRC and PHI. Additionally, our results show that reconfiguring 8 KB or less of the SM's local SRAM into a LAB is often sufficient. Finally, LAB does not affect applications that do not use commutative atomics, unlike other state-of-the-art solutions.

Listing 3.1: Pseudo-code of GPU histogram kernel [141].

```
if (tid <= (N - 1)) {
    loc = arr[tid];
    //atomicAdd(&hist[loc], 1, mem_order_comm);
    atomicAdd(&hist[loc], 1); // commutative
}
```

Listing 3.2: Pseudo-code from key PageRank kernel [37].

```
end = ((tid+1) < numNodes ? row[tid+1] : numEdges);
for (edge = row[tid]; edge < end; ++edge) {
    nodeID = col[edge];
    inc = pR1[tid]/(float)(end-start);
    //atomicAdd(&pR2[nodeID], inc, mem_order_comm);
    atomicAdd(&pR2[nodeID], inc); // commutative
}
```

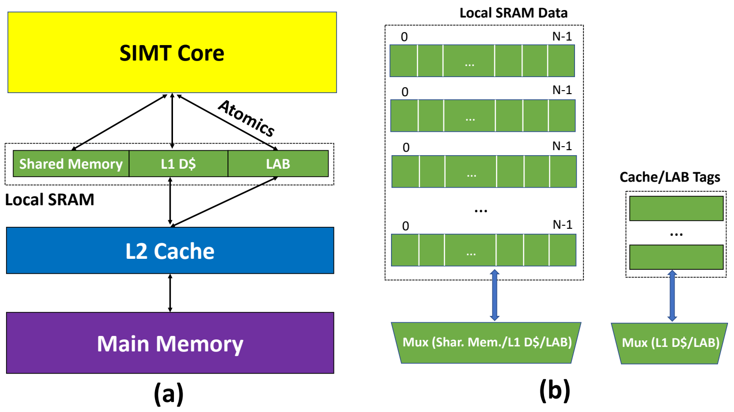


Figure 3.2: Proposed design (a) including LAB (in green) and (b) local SRAM.

3.2 Design and Implementation of the LAB

3.2.1 LAB Hardware Support

Figure 3.2a shows our proposed addition of the LAB to the GPU memory hierarchy. Physically, the LAB is similar to the L1 data cache; as shown in Figure 3.2b we exploit the reconfigurability of the unified local memory [68] to partition it into L1 data cache, shared memory, and LAB. This requires an additional 2:1 mux for the tag array to avoid duplication and determine if a tag belongs to the cache or LAB; the data array already has a 4:1 mux which previously had one unused input [68].[1] Like some GPU L1 caches, the LAB has 128 byte lines, broken into four 32-byte sectors.

Since LAB is intended to be small, we make it associative. LAB also utilizes a portion of the local SRAM's data and tag arrays (via the muxes in Figure 3.2b). The data array holds partial values for a given address, while the metadata holds address, replacement, and atomic. Since we are only storing an update to the overall global variable, we use an allocate on fill write miss policy, where the variable is allocated the initialized with the value of the first update to that address. Next, we discuss LAB's operation.

3.2.1.1 Evictions

When the LAB is full, we use an LRU replacement policy to determine which entry to evict. However, evictions can be done off the critical path, since the commutative atomics do not imply any ordering on other memory accesses. Accordingly, as soon as the message is sent to the L2, we reuse the entry. When the evicted atomic request reaches the L2, it updates the appropriate addresses' value.[2]

[1]It is also possible to implement LAB inside the L1 cache using techniques such as way partitioning [43]. However, way partitioning also requires additional muxes and may increase conflict misses. Thus, we focus on partitioning the SRAM, which also decouples data and atomic accesses.

[2]Like prior work [6, 118, 203], we assume the GPU has an ALU co-located with the L1 cache. If this is not the case, then Figure 3.2 and the overall area (Section 3.4.6) would need to include this.

3.2.1.2 Coalescing

Like L1 cache and shared memory accesses, the GPU coalescer coalesces accesses before sending them to the LAB. Thus, threads within the same warp are coalesced and LAB can use the same number of read and write ports as the L1 cache.

3.2.1.3 Handling Uncoalesced Accesses

Some programs have divergent, uncoalesced memory accesses where every thread may access a unique cache line. Although we only observed up to 38% divergence (12 unique cache lines/warp), LAB still supports these accesses. On an uncoalesced access, the GPU (and LAB) treats each requested line as a unique request. Consequently, LAB handles them as they arrive. If the number of requests exceed the LAB's size, then requests evict entries from the same uncoalesced access.

3.2.1.4 Behavior at Ordering Points

Although atomics using the LAB are relaxed and thus do not imply ordering on other memory requests, at ordering points we flush all LAB entries to the L2. Thus, at all kernel boundaries or at software enforced ordering points (e.g., CUDA's **threadfence** or barriers, mutexes, and semaphores) all LAB entries are evicted. None of our benchmarks had software ordering points, so flushes only happened at kernel boundaries.

3.2.1.5 Atomics with Ordering Requirements

Since LAB stores partial atomic updates, it cannot be used for atomics with ordering requirements (e.g., SC atomics) without causing significant delays. Thus, our software requirements (Section 3.2.2) ensure that only commutative atomics use the LAB. Non-commutative atomics must be performed at the appropriate level defined by their scope. We discuss LAB's implications on GPU coherence and consistency further in Section 3.2.3.

3.2.1.6 Serialization of Atomics

Atomic serialization occurs due to two primary factors: atomic collisions and centralized resources. Intra-warp atomic collisions occur when multiple threads in a warp attempt to update the same memory location concurrently, while inter-warp and inter-TB atomic collisions occur when multiple threads from different warps (or different TBs) attempt to update the same address concurrently. When this happens, one request must be issued before the other. Atomic serialization can also occur at a centralized destination such as the L2 cache. Since device-scoped atomics are performed at the L2 (which is $\approx$6X more expensive than GPU L1 accesses, Section 3.3), this serialization can be very expensive, especially when combined with L2 queuing delay and potentially backpressure from interconnect stalls. Since LAB performs all commutative, device-scope atomics locally, LAB reduces the serialization penalty due to atomic collisions. Moreover, LAB's decentralized updates (one LAB per SM) also reduces atomic serialization from centralized resources, since only the combined requests are sent to the L2.

3.2.1.7 Identifying Atomic Function

CUDA supports multiple types of atomics, some of which are not commutative with each other. Thus, the LAB must track what atomic function is being performed for each line. Since CUDA has fewer than 16 atomic functions [165], we use 4 bits per LAB line to identify the atomic operation. If a LAB hit occurs but the atomic function does not match, we flush the entry.

3.2.1.8 Benefits Over Software Solutions

Since the atomic operations are commutative and are not read before all updates complete, programmers could either use per-TB shared memory to accumulate updates locally or use L1 data cache with private variables. In this situation, a per-TB global atomic performs the final global memory update. However, for most of our applications this would require large, sparsely accessed shared memory allocations that limit the number of TBs per SM. In contrast, LAB only holds frequently accessed atomic addresses and their values, without requiring large allocations.

Overall, LAB effectively enables per-SM intra-warp, inter-warp, and inter-TB reuse in the same kernel invocation. For example, when threads in Listings 3.1 and 3.2 on the same SM (inter- or intra-TB) access the same address atomically, they can be reused in the LAB. Moreover, when atomic reuse is minimal, LAB's decentralized design still reduces the serialization penalty (3.2.1.6).

Although the LAB may increase the burstiness of the atomic traffic in the worst case, we have not observed significant increases in queuing delay due to LAB's coalescing benefits and because SMs usually flush at different times (as shown in Section 3.4). Moreover, the atomics sent to the L2 by the LAB will be performed off the critical path, except at kernel boundaries where the LAB is flushed in parallel with the caches, since instructions complete upon reaching the LAB. This lessens the observed impact of burstiness (Figure 3.8).

3.2.2 Software Support

Distinguishing Atomic Operations: LAB relies on identifying which atomic accesses can be buffered locally (e.g., commutative atomics). To identify which accesses are commutative atomics, we leverage recent work that proposed additional memory orderings: SC-for-Data Race Free Relaxed (or SC-for-DRFrlx) introduced a new memory ordering, memory_order_comm, to identify commutative accesses [204]. However, since we do not need the additional complexity for other, non-commutative relaxed atomics that SC-for-DRFrlx proposes, we use a SC-for-HRF consistency model with the additional commutative memory ordering. Programmers or compilers can instrument software to use this new memory ordering to indicate commutative atomics to the hardware, analogous to how C, C++, HSA, and OpenCL specify other memory orderings [27, 86, 87]. For example, Listing 3.1 and 3.2, show how the programmer would label the `atomicAdd`'s (the commented-out versions) as commutative atomics. Non-commutative atomics bypass LAB.

3.2.3 Impact on Consistency and Coherence

The key ideas that allow the LAB to batch atomic updates without affecting consistency and coherence guarantees are:

3.2.3.1 Commutativity

The order of commutative Read-Modify-Write updates to a shared variable can be arbitrary without affecting correctness. Since these updates race, they must use atomics to conform to the SC-for-HRF consistency model (Section 2.2). However, since reordering updates still produces correct results, programs often use relaxed atomics for the updates [204]. Thus, caching relaxed commutative atomics in the LAB should not affect the final results. By perturbing the order of atomics, LAB may cause small rounding errors for floating point commutative atomics, but this is already a problem on real GPUs, where the atomic order is not deterministic [40, 42, 52]. Although we only observed minor impacts on the final results (less than 1% for all the micro-benchmarks and full-sized benchmarks studied in this chapter), if complete determinism is desired we could adopt DAB [40] or Reproducible FP [52] at the cost of additional area.

3.2.3.2 Interaction with Data Accesses

Like C++ [27], HSA [87], and OpenCL [109], by default we assume that atomically updated shared variables are always accessed atomically. Thus, if a program uses properly labeled and synchronized commutative atomics, there will never be data accesses to the same variable and SC results are guaranteed (which also ensures that the program does not view the updated values until all updates have completed) [204]. As a result, data accesses and non-commutative atomics do not need to check the LAB, since the commutative atomics do not order other accesses and all accesses to a commutative address are atomic. However, although CUDA is moving towards similar requirements [166], currently it allows a variable to be accessed by both atomic and data accesses in the same kernel. If this happens for accesses using `memory_order_comm`, a commutative race may occur [204] and, like other DRF-based consistency models, threads may access a stale value and SC results are not guaranteed (e.g., since data accesses do not check the LAB). However, if the data accesses occur in separate kernels, since the LAB is flushed at the end of each kernel, the data accesses will see any previously buffered LAB updates. If strong atomicity is to be violated or the assumption that a value is not read before the next kernel is broken, the LAB design does not provide any correctness guarantees. In such cases, the programmer will need to insert appropriate fences to ensure that the output produced is functionally correct. The LAB will evict all its entries whenever a fence instruction is issued by the program (3.2.1.4).

3.2.3.3 Coherence

As discussed in Section 2.2, GPU coherence relies on data-race-freedom and software invalidations to ensure that there is no stale data in the local caches. Adding the LAB does not impact the GPU coherence protocol, since it only aggregates updates for commutative, atomically accessed global addresses. Accordingly, LAB does not require any changes to the existing coherence protocol and additional fences are not required because the commutative updates do not need to be ordered with one another.

GPU Feature	**Configuration (Size, Access Latency)**
SMs	80
# Registers / SM	64 KB
LI Instruction Cache / SM	128 KB
LI Data Cache / SM	32 KB (max 128 KB), 28 cycles
L2 Cache	4.6 MB, 148 cycles
MSHR	256 (L1) and 192 (L2) Entries
Shared Memory Size / SM	96 KB (max 128 KB), 19 cycles
Memory	16 GB HBM2, 248 cycles

Table 3.1: Simulated baseline GPU parameters

Operation	**Energy (pJ)**
Non-Memory Operation	3.7
L1D (32 KB) Read/Write	1.4097, 1.7044
L1I (132 KB) Read/Write	5.6387, 6.8177
L2 (4.6 MB) Read/Write	193.59, 234.0675
LAB (Size 8) Read/Write	0.0881, 0.1065
LAB (Size 16) Read/Write	0.1762, 0.2131
LAB (Size 64) Read/Write	0.3524, 0.4261
LAB (Size 128) Read/Write	0.7048, 0.8522
LAB (Size 256) Read/Write	1.4097, 1.7044
LAB (Size Inf) Read/Write	45.1097, 54.5417
NOC	254
Main Memory	501

Table 3.2: Per-access energies used [44, 77, 172].

3.3 Methodology

3.3.1 Simulation Environment & Parameters

To evaluate LAB's impact, we added LAB to GPGPU-Sim v4.0.0 [23, 104, 129, 130, 186], which has been shown in previous work to provide high accuracy for modern NVIDIA GPUs, including when running ML workloads [104, 129, 130].

Table 3.1 summarizes the key system parameters, which is based on a NVIDIA Titan V GPU [168]. Additionally, we assume support for performing atomics at the LAB. We use CUDA 8 and cuDNN v7.1.3 for the ML training benchmarks, because these are the latest versions of CUDA and cuDNN that embed the PTX in the libraries – which is necessary to run cuDNN in GPGPU-Sim [130][3] For all other benchmarks (Table 3.3), we use CUDA 11.2. Although GPGPU-Sim has an integrated energy model [128], it has not been validated for post-Fermi architectures and is not representative for modern GPUs.[4] Thus, we use a per-access energy model (Table 3.2) based on recent work [44, 77, 172].

To label commutative atomics (Section 3.2.2), like prior work [204] we use software flags to find and simulate these accesses.

3.3.2 Configurations

Since the LAB is physically located in the unified local SRAM, configuring part of the SRAM to be LAB reduces the size of the L1 data cache or shared memory. Hence, we examine varying the size of the L1 data cache and shared memory. Overall, we use the following configurations:

- **Baseline**: The baseline GPU configuration without an LAB, with a 32 KB L1 data cache and 96 KB shared memory, and which performs all device-scoped atomics at the shared L2.

- **Cache-8KB**: Models the *Baseline* configuration with 8 KB less cache, which is representative of a 64-entry LAB.

- **Cache+8KB**: *Baseline* configuration with 8 KB more cache instead of using LAB.

- **Cache*2**: Like *Cache+8KB*, except doubles the cache size.

- **hLRC**: hLRC [6] obtains ownership for atomics, enabling it to cache them locally. Since hLRC has not been publicly released, we implemented and validated it in GPGPU-Sim.

[3]Accel-Sim [106] introduced the capability of directly simulating SASS code, enabling the use of newer versions of CUDA and cuDNN but it was not available until after this work was published.

[4]Accel-Wattch now provides a validated GPU energy model for modern GPUs, but was not available until after this work was published [99].

- **PHI**: PHI [146] buffers atomics in write allocate L1 caches (fetch on write); we implemented and extended PHI for GPUs (only cache lines with commutative atomics are buffered updates). We also extended PHI to use a lazy fetch on read scheme, which has a 6% difference but did not affect the takeaways. Although PHI was designed for MESI-like CPU coherence, we optimistically ignore invalidation and downgrade overheads – otherwise PHI is worse.

- **LAB** `i`: We vary LAB's size to examine its sensitivity: LAB `i` represents `i` LAB entries per SM: 8, 16, 32, 64, 256, and Infinite. Each statically reconfigures the cache, shared memory, and LAB proportions based on LAB size before kernel launch, similar to CUDA's existing cache/shared memory flag. Although some configurations require significant SRAM, we include them to examine larger LAB performance. For context, a 64-entry LAB uses approximately 8 KB of local SRAM. For all LABs except infinite, we take space from the cache since the applications were less sensitive to cache size and changing shared memory size affected utilization.

We also tried studying additional configurations of the local SRAM to determine if existing caching mechanisms could be used to achieve some of LAB's gains. Increasing shared memory size per SM had no effect. Moreover, although weights currently do not use shared memory, cuDNN uses shared memory for other arrays (from inspecting disassembled binaries [222]). However, since the TBs per SM are limited by register file size [180], increasing shared memory per SM did not increase the TBs per SM. Since cuDNN is closed source, we tried modifying cuTLASS [103] instead. However, cuTLASS lacked the corresponding kernels.

Finally, to isolate LAB's serialization and coalescing benefits, we implemented a LAB variant that performs atomics locally but does not store data in the LAB, preventing reuse and separating the coalescing and serialization benefits. Since it

Benchmark	**Input**
Microbenchmarks	
Histogram[141] (H)	256K (30720x17280 pixels)
Histogram_Shared[141] (HS)	256K (30720x17280 pixels)
Backward Conv[56] (BWC)	NCHW = 128,3,256,256
Graph Analytics ('_' denotes different utilization levels)	
BC[_100, _75, _50, _25][37]	CT, NH, VT, AK[53]
CCA[_100, _75, _50, _25][199]	amazon, olesnik0, wing, emailEnron[199]
CC[199, 223]	flower.txt[199]
CLR[199], MIS[199], PR[37], SSSP[199] [_100, _75, _50, _25]	or2010, nd2010, nv2010, nh2010[22]
ML Training	
AlexNet[56] (AN)	NCHW = 16,3,227,227
VGG-19[56] (VGG)	NCHW = 16,3,112,112
SqueezeNet[56] (SN)	NCHW = 16,3,224,224
Tiny YOLO[188] (TY)	NCHW = 16,3,416,416
ResNet[80] (RN)	NCHW = 16,3,256,256
GPGPU	
Backprop[35]	64K
B+Tree[35]	mil.txt
BFS[35]	graph1MW_6.txt
DWT2D[35]	1Kx1K
gaussian[35]	1Kx1K
Heartwall[35]	test.avi
Hotspot[35]	512x512
huffman[35]	1024
HybridSort[35]	2^{18}
KMeans[35]	kdd_cup
LavaMD[35]	boxes1D
Leukocyte[35]	testfile.avi
LUD[35]	512
MummerGPU[35]	NC_003997
Myocyte[35]	100
NN[35]	lat 30, long 90
NW[35]	8Kx8K
Pathfinder[35]	1Mx100x20
ParticleFilter[35]	128x128, 4K particles
SRAD[35]	2Kx2K
Streamcluster[35]	8K

Table 3.3: Benchmarks and inputs.

is difficult to isolate serialization from reducing backpressure and interconnect stalls, this provides a minimum bound on LAB's serialization reductions. We also isolated the burstiness overheads by turning off the end-of-kernel LAB flushes and measuring its impact (shown using a separate bar in Figure 3.8).

3.3.3 Benchmarks

Table 3.3 summarizes the workloads we use. To study performance for GPGPU applications, we analyze Rodinia [35, 36] with inputs sized to fully utilize the GPU.[5] We also analyze two histogram [141] variants each with 256 bins: an ideal use case for LAB where most accesses are device-scoped atomics, similar to Listing 3.1, and another that uses shared memory to bin updates locally before sending a single atomic update per bin to global memory [204].

We also use popular graph analytics and ML training workloads: BC, CCA, CC, CLR, MIS, SSSP, PR, AlexNet, VGG-19, SqueezeNet, Tiny YOLO, and ResNet. We selected these benchmarks because they cover a wide variety of use cases, and have been shown to be high performance in prior work [199, 209]. The larger benchmarks are unable use similar software optimizations to the histograms. Since the graph analytics algorithms are input dependent, we study multiple input graphs that utilize varying amounts of the GPU.[6]

For all DNNs we extend DNNMark [56] to model the networks and run them for one iteration since CNN iterations have similar behavior [243, 247]. Moreover, to study the training kernels that use atomics in isolation, we also run microbenchmarks such as BWC and ResNet (1 Layer).

[5]We do not use CFD because it has issues with GPGPU-Sim 4.0 [106].

[6]Except for CC, which only has one publicly available input size.

3.4 Results

Figures 3.3-3.6 show the performance improvement, interconnect traffic reduction, miss rate, and energy consumption, respectively, for all microbenchmarks and benchmarks, across the configurations described in Section 3.3. We divide energy into multiple components based on source: ALU, shared memory, L1, L2, interconnect, LAB, and main memory. Broadly, the smaller and larger cache configurations show no appreciable change in performance due to the mostly streaming, read only nature of the application's data loads. Thus, we do not show the additional cache configurations in Figure 3.6, as the energy impact follows a similar trend. In comparison, LAB yields significant benefits. With LAB, an application's performance is closely tied to the ratio of global atomic requests to the total number of global memory requests (ATGR), the application's spatial and temporal locality for atomic transactions, and the application's ratio of LAB size to atomic's working set size. Since these properties vary per application, LAB's benefits vary. LAB's coalescing benefits tap into the locality that exists within atomic transactions, while the serialization benefits (Section 3.2) result from performing atomics locally at the LAB rather than at the L2. Without LAB, overlapping, device-scoped relaxed atomics are sent to the L2, increasing queuing delay and interconnect buffer stalls. Coalescing these atomics in the LAB reduces these overheads, and helps LAB rival Figure 3.1 (since Figure 3.1 profiles real GPUs, it cannot include LAB reuse). Overall, across all non-infinite LAB sizes, on average LAB improves performance by 28%, reduces energy by 19%, and reduces interconnect traffic by 19%, while also improving on state-of-the-art techniques like PHI and hLRC.

3.4.1 Graph Analytics Workloads

GPU Utilization Study: Figure 3.7 shows how the graph analytics algorithms perform for different input graphs that utilize 25%, 50%, 75%, and 100% of the GPU, respectively, averaged across LAB sizes 8-256. As utilization increased, the graphs provide additional reuse opportunities, but also increased contention that

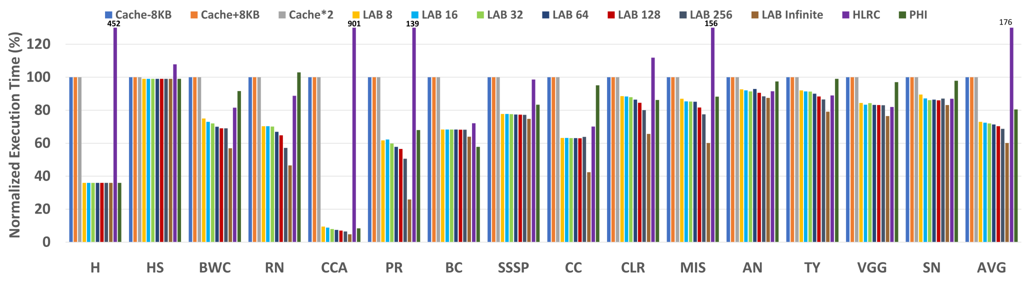


Figure 3.3: Execution time for different cache configurations, LAB sizes, hLRC, and PHI, normalized to the baseline configuration without LAB from Table 3.1.

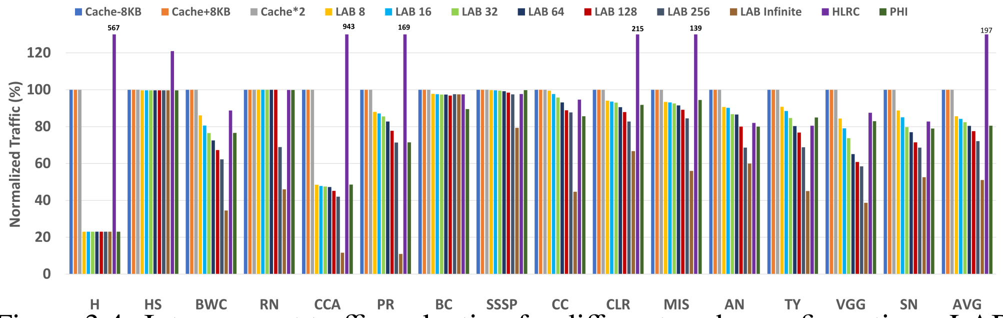


Figure 3.4: Interconnect traffic reduction for different cache configurations, LAB sizes, hLRC, and PHI, normalized to the baseline configuration without LAB from Table 3.1.

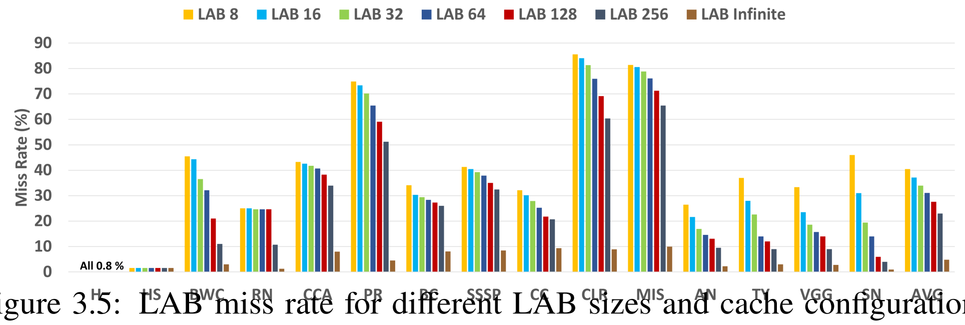


Figure 3.5: LAB miss rate for different LAB sizes and cache configurations, normalized to the baseline configuration without LAB from Table 3.1.

may increase LAB misses. For benchmarks with High ATGR, the connectivity of the graphs played impacts performance improvement. For example, in PR_75 24% of all nodes are strongly connected, while PR_50's graph has less connectivity: only 5% of the nodes are strongly connected. As a result, LAB provides more reuse for PR_75 than PR_50, and further improves performance. For CCA,

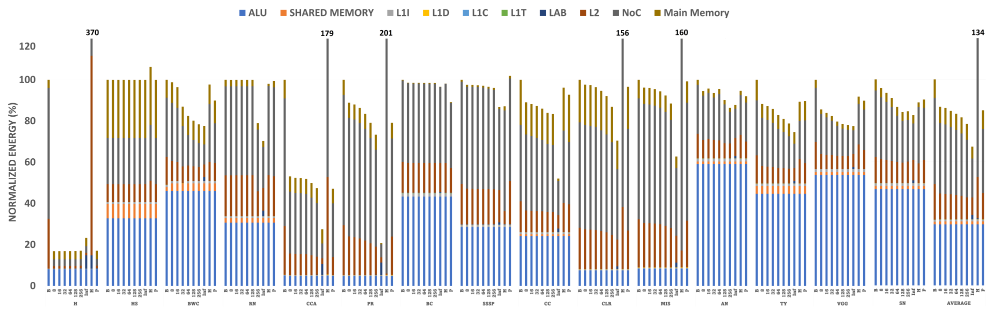


Figure 3.6: Energy consumption normalized to the baseline without an LAB from Table 3.1. For each application, left to right is the baseline (B), LAB-8 (8), LAB-16 (16), LAB-32 (32), LAB-64 (64), LAB-128 (128), LAB-256 (256), LAB-Inf (Inf), hLRC (H), and PHI (P).

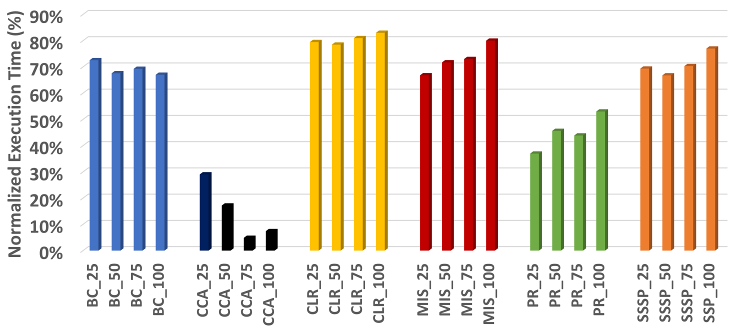


Figure 3.7: Execution time for the graph analytics workloads with different utilization levels, averaged across LAB-8 to LAB-256, normalized to the baseline configuration without an LAB from Table 3.1.

both CCA_75 and CCA_100 have a few very strongly connected nodes. Thus, LAB captures most of the reuse even for smaller LAB sizes and enables them to outperform CCA_25 and CCA_50. However, CCA_75 slightly outperforms CCA_100 because it has less contention. The lower utilization graphs also outperform by the higher utilization graphs in a few other cases. For BC and SSSP, which generally have low locality, the majority of LAB's benefits come from reducing the serialization benefit. As a result, performance is similar for all utilization levels. However, for benchmarks with Moderate ATGR such as CLR and MIS, higher utilization levels consistently increase performance slightly. This happens because the larger working sets in the higher utilization graphs of

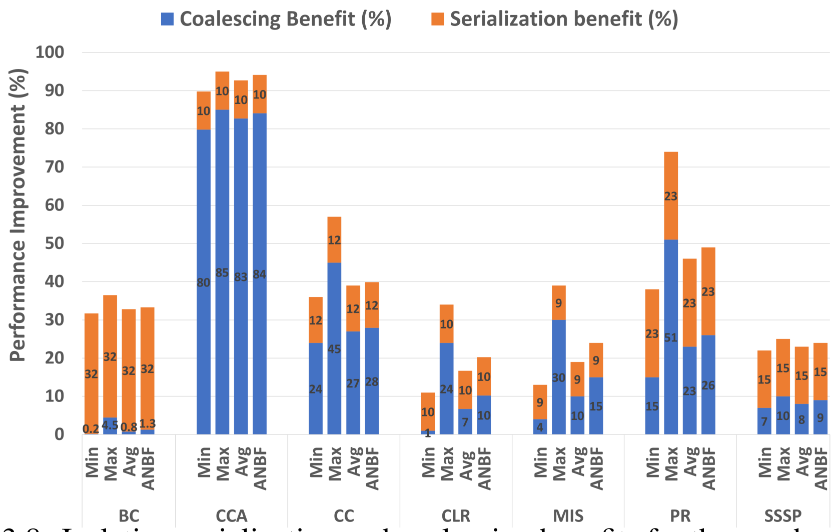


Figure 3.8: Isolating serialization and coalescing benefits for the graph analytics workloads. ANBF: average without bursty flush. ML workloads not included due to space constraints.

CLR and MIS dominate compared to the additional reuse they offer. Nevertheless, since these differences are small and the overall performance gains are similar for all four utilization levels, we focus on the full (100%) utilization graphs in the remaining analysis.

High Locality & ATGR: As shown in Figure 3.1, *PR* (0.72) and *CCA* (0.84) have high ATGRs ratio and many commutative atomics. For CCA, a small subset of vertices are very strongly connected. Thus, even a small LAB significantly improves performance: across all LAB sizes, PR improves performance up to 74% (42% on average) and CCA up to 95% (92% on average), with similar improvements in energy consumption, interconnection traffic, and miss rate. As LAB size increases, LAB buffers more atomics locally, but also reduces L1 cache space. However we observed that for both CCA and PR (especially CCA) the average reuse distance often decreases for increasingly strongly connected nodes: CCA and PR's average reuse distance decreases by 96% for the most strongly connected nodes). Moreover, larger LAB sizes are tolerant to higher reuse distances and hence capture more reuse and further improve performance.

Reducing the cache size has less effect because the load locality is relatively lower. The interconnect traffic reduction follows similar trends: a maximum reduction of 89% for LAB-Inf for PR and 88% for CCA. Finally, the overall energy trends are directly proportional to interconnect traffic since device-scoped atomics dominate: on average, energy is reduced by 16% (max 79%) for PR and by 48% for CCA.

LAB also improves CCA's and PR's performance by decreasing serialization cost. As shown in Figure 3.8, since small LABs (min results) have fewer coalescing opportunities, the serialization cost reduction provides 15% of PR's benefits. However, as LAB size increases (average and max results), coalescing opportunities increase and progressively make up a larger percentage of PR's and CCA's improvements.

Low ATGR & Locality: BC's (0.05) and SSSP's (0.19) ATGRs are significantly lower than CCA and PR. However, as also observed in prior work [58], these atomic accesses cause bottlenecks in BC and SSSP due to serialization. As a result, across all LAB sizes on average performance improves by 30% for BC and 21% for SSSP. Nevertheless, since there are few atomics, even with an infinite LAB, energy and network traffic gains are small (e.g., 3% less network traffic for BC with up to 1% less energy). Interestingly, although BC and SSSP have less locality than CCA and PR, their average reuse distance is 82% lower than PR and CCA for weakly connected nodes. Consequently, all LAB sizes capture similar amounts of reuse, resulting in smaller differences in miss rate until the data completely fits in the LAB (LAB-Inf). Since BC and SSSP have fewer coalescing opportunities, unsurprisingly the vast majority of LAB's benefits come from reducing serialization latency (Figure 3.8).

Moderate ATGR & Locality: CC, CLR, and MIS have fewer device-scope atomics than CCA and PR but more than BC and SSSP. Although CC, CLR, and MIS have similar ATGR and locality, they have different access patterns. Some of CC's kernels are streaming with little or no reuse; in these kernels most of LAB's benefits come from reducing serialization costs. Nevertheless, most of CC's kernels have moderate to good reuse; in these kernels LAB provides more benefits from coalescing. Like BC and SSSP, the kernels in CC that have at least moderate reuse have small average reuse distances, enabling even small LABs to provide good performance. CLR (ATGR: 0.24) initially performs device-scoped atomics on many cache lines, then reduces the working set as the application proceeds. Thus, all LAB sizes perform similarly once working set decreases. Furthermore, when the working set is large, larger LABs reduce more atomic traffic, but the reduced cache size also increases cache misses. However, again LAB's benefits outweigh the reduced cache locality, although the reduce cache hits reduce the performance difference between the LABs. Somewhat similar to CLR, MIS (ATGR: 0.43) has a large device-scoped atomic working set. Thus, a larger LAB is needed to capture the possible reuse. Accordingly, reducing serialization provides most (69%) of the benefit for small LABs, but a smaller portion (47%) for larger LABs. Moreover, the burstiness of the flushing the LAB is small (Figure 3.8): 0.5% (BC) - 5% (MIS) on average, and outweighed by LAB's overall gains. Overall, on average performance increases by 37% for CC, 14% for CLR, and 16% for MIS.

3.4.2 ML Training Workloads

Overall, on average across all DNN workloads LAB improves performance by 18%, energy by 11%, and interconnect traffic by 19%. However, different training algorithms exhibit different trends in terms of benefits, especially as the number of layers, parameters, and batch sizes vary. For the ML benchmarks LAB's gains are larger for deeper networks and bigger batch sizes – trends that are expected to continue in next generation ML workloads. However, the overall gains are sometimes limited because CNNs are largely compute bound on modern GPUs. Nevertheless, given the significant efforts to optimize compute for CNNs, the memory system will become more of a bottleneck, increasing LAB's utility.

AlexNet, Tiny YOLO: *AlexNet* (AN) and *Tiny YOLO* (TY) have relatively low ATGRs because, like other CNNs, they are compute bound [70]. Thus, LAB improves interconnect traffic and performance for kernels with atomics, but the overall gains are smaller. AN is relatively unaffected by increasing LAB size (9% average performance improvement). Here, reducing cache size while incrementing LAB size is sometimes detrimental: performance declines a little when using a larger LAB due to reduced cache locality. Although AN and TY have a similar number of layers, TY spends more time in kernels with device-scoped atomics (85% compared to 74% for AN). Thus, LAB improves TY's performance more than AN's. On average LAB improves performance by 10%, decreases interconnect traffic by 18%, and decreases energy by 12% for AN and TY.

Moreover, AN's reuse pattern is different for bwd_filter and bwd_data. Bwd_filter is similar to RN, where addresses only exhibit temporal reuse after a certain number of accesses to unique addresses. Thus, the overall reuse depends on batch size and layer parameters. Bwd_data has a lower ATGR than bwd_filter, but more temporal reuse since a small subset of addresses are repeatedly reused. Thus, miss rate decreases as LAB size increases: LAB 64 captures an average of 71% of the reuse for the most heavily accessed addresses, while LAB 16 only provides 28% of the same reuse.

VGG19, SqueezeNet: *VGG19* (VGG) and *SqueezeNet* (SN) are deeper networks, and VGG has a larger batch size (64). Thus, they have more device-scoped atomics than smaller networks like AN, and accordingly larger improvements from LAB: for VGG performance improves up to 24% (16% average), energy decreases up to 22% (19% average), and interconnect traffic decreases up to 61% (29% average). Similarly, more of SN's bwd_filter and bwd_data kernels have ATGR $> 30\%$. As a result, SN has better average performance improvements (13%) than AN, but smaller improvements than VGG, which has a larger batch size and thus more device-scoped atomics. Similar to the other DNNs, on average LAB reduces interconnect traffic by 21% and energy by 18%. Although VGG and SN see additional benefits from larger LABs, due to more temporal reuse in the bwd_data kernels and to a lesser extent in bwd_filter kernels, a size 16 LAB again provides the majority of the benefits, for the same reasons as previously described networks.

3.4.3 Comparison to hLRC

hLRC improves reuse and reduces the serialization penalty by obtaining ownership for atomics. However, hLRC struggles for large working sets and high contention because atomics and data accesses contend for L1 cache entries. Moreover, when multiple SMs perform atomics on the same cache line, hLRC must forward ownership to remote L1s, adding additional overhead. Although hLRC performed well for smaller graphs in prior work [6], our larger graphs have more frequent remote L1 ownership requests, larger working sets, and higher contention. Consequently, hLRC performs poorly for them, especially for CCA, MIS, PR, and Histogram which have high inter-SM contention for atomics. In Histogram and CCA this is amplified by heavy contention across a small subset of addresses, significantly increasing network traffic due to remote invalidations and further hurting performance. Conversely, most ML workloads have less contention or perform atomics in a small window, reducing remote invalidations and enabling more reuse. Consequently, hLRC provides similar performance to LAB for ML workloads, especially for small LAB sizes. However, as LAB size increases (e.g., 64+ entries for AN and VGG), LAB batches more updates without evictions than hLRC. Overall for the full sized benchmarks, compared to the baseline hLRC reduces performance by 76% (3% performance improvement without CCA and Histogram), energy by 72% (12% reduction without CCA and Histogram), and network traffic by 102% (20% reduction without CCA and Histogram). Thus, LAB outperforms hLRC by enabling multiple SMs to update local copies before sending out partial updates.

3.4.4 Comparison to PHI

Overall, PHI outperforms hLRC and the baseline. Although, like hLRC, PHI caches atomic updates locally, it does not suffer from remote invalidations [146]. This helps for Histogram, PR, and CCA, which have numerous commutative atomics that PHI buffers locally, significantly reducing network traffic and improving its performance and energy versus hLRC. Furthermore, avoiding expensive remote invalidations also helps PHI outperform hLRC for some applications with moderate or low ATGR and locality like SSSP, CLR, and MIS. However, hLRC outperforms PHI for CC and ML workloads. In the ML workloads hLRC has fewer remote invalidations since SMs usually update their own fixed set of weights. Unlike PHI, hLRC also must wait for ownership for atomics, which reduces these workload's contention and stalls compared to PHI.

Although PHI provides some of LAB's benefits, and offers the second best performance of all configurations, overall LAB outperforms PHI (on average 20.94% better performance, 0.2% better on network traffic and 3% better on energy, and , respectively, for LAB-64; 20% performance, 1.5% energy, and 0.3% network traffic for LAB-Inf). For the histograms, AN, CCA, and SSSP, PHI provides most or all of LAB's benefits because their working set fits in the cache. Moreover, PHI outperforms LAB for BC by better leveraging BC's limited locality across atomics, since PHI can evict either a regular data read or an atomic when an atomic misses, unlike LAB. However, PHI significantly increases L1 cache stalls due to increased L1 contention, which usually occurs when PHI utilizes all the MSHRs for pending data read misses that are evicted by interspersed atomics. Thus, while PHI reduces network traffic for some benchmarks, like LAB, in others it increases stalls and contention between reads and writes. Accordingly, PHI cannot provide all of LAB's benefits in applications with large working sets that cause frequent L1 cache evictions (e.g., CC, CLR, MIS, SN, and VGG). Although these workloads sometimes evict data loads or stores with limited locality, in other situations PHI's co-mingling of data and atomic accesses increases stalls and limits atomic reuse. As a result, LAB provides more consistent improvements than either hLRC or PHI, by both explicitly exploiting commutativity and decoupling where data and atomic accesses are stored.

3.4.5 Traditional GPU Workloads

In Figure 3.9 we evaluate the baseline's, LAB's, PHI's, and hLRC's performance for more traditional GPGPU workloads. We allocate LAB space only for applications that have atomics (Huffman and HybridSort). Thus, we use LAB-0 for all other applications in Figure 3.9 (adjusted for LAB-0's overheads, see Section 3.4.6), and LAB-64, which was big enough for other applications, for Huffman and HybridSort. Since Huffman and HybridSort perform histograms, unsurprisingly both PHI and LAB improve performance by 24% and 26% respectively, while hLRC again suffers from inter-SM contention. Since the other applications do not use atomics, both LAB and hLRC perform similar to the baseline: all were within 1.2% of the baseline. However, PHI again increases contention between reads and writes, hurting performance for Backprop, BFS, DWT2D, MummerGPU, and SRAD. PHI also performs worse for NW, NN, pathfinder and particlefilter, though the performance degradation is smaller, either because these applications have fewer writes or because their read locality is low enough that writes taking up cache space does not significantly impact performance. Conversely, PHI does better for LUD, gaussian and b+tree, which have write locality. Overall, on average PHI is 14% worse and LAB is 0.9% better than the baseline. This further highlights the importance of decoupling data and atomics in GPUs, like LAB, whose reconfigurability allows it to work well across both synchronization-heavy and traditional streaming GPGPU workloads, unlike PHI.

3.4.6 Area

Although LAB dynamically partitions the local SRAM, LAB does have some small overheads within the reconfigured SRAM block. Each LAB line requires 4 additional bits (e.g., 32 bytes in total for an LAB of size 64) to identify the atomic operation. Additionally, LAB also adds a 2-1 mux (Section 3.2.1) and a 4 bit comparator to ensure the atomic operation matches. In total, this requires 4 caches lines of overhead in a 128 KB L1 GPU cache, and our results in Section 3.4.5 show

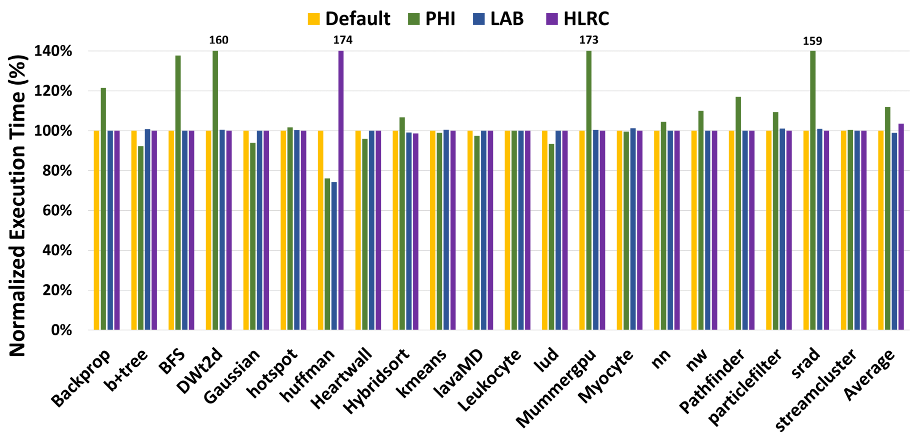


Figure 3.9: GPGPU results for PHI, hLRC and LAB, normalized to baseline.

this has minimal impact on performance. Alternatively, to avoid extra storage, we can exploit the fact that the atomic values did not require all of the data bits, and instead use 4 data bits per line for this.

3.5 Discussion

Software vs. Hardware: Histogram obtains significant benefits from using shared memory to exploit commutativity. However, as discussed in Sections 5.4.4 and 3.3.3, the larger graph analytics and ML training workloads cannot use shared memory for their vertices (graph analytics) and weights (ML training) because GPUs must statically allocate the entire array in shared memory – even if a given TB only accesses a small subset of the locations. Moreover, these large arrays exceed the maximum shared memory size per TB. To demonstrate this effect, we increased the number of histogram bins from 256 to 8192. Since Histogram Shared creates partial histograms for each TB, using more bins reduces how many TBs can run simultaneously from 8 to 3 TBs per SM, hurting performance.

In comparison, Histogram has no such limitation. Thus, it outperforms Histogram Shared by at least 3X for 8192 bins with LAB as shown in Figure 3.10. Moreover, as histogram bins increase, Histogram Shared eventually cannot run even 1 TB per SM. Thus, using GPU software optimizations like shared memory can improve performance, but only when the working set is sufficiently small. In comparison, LAB dynamically retains the most highly used locations, improving reuse even for applications with large working sets. For example, on average AlexNet's weight array size is 466540 bytes, which requires ~1900 KB of storage – whereas modern GPUs only allow up to 192 KB of shared memory per SM. Finally, for graph analytics algorithms, the vertex updates are not predictable at compile time and hence it is difficult to use shared memory to improve performance.

It is also possible to virtualize and manage shared memory allocations manually in software [8, 43, 115]. This enables programs with large shared memory requirements to run. However, this requires programmers to handle issues such as evictions, significantly increasing overhead (especially from thread divergence), and prior work has shown that such approaches provide mixed results for CPUs [113].

Applicability to Other ML Training Algorithms: Our results focused on CNN

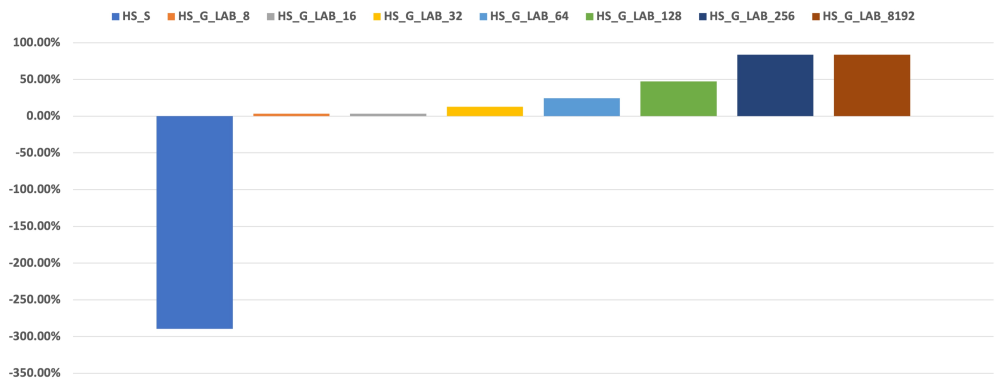


Figure 3.10: Performance improvement with respect to Histogram Global (HS_G)

training algorithms (Section 3.3.3). However, LAB is also applicable to other ML training algorithms: any ML training algorithm that atomically updates shared weights at the end of a training iteration, which is common in data parallel training, could utilize a similar approach. Similar to DAB [40], we attempted to examine recurrent neural network (RNN) training. Like DAB, we found that current versions of cuDNN do not use atomics for weight updates in RNN training. Nevertheless, we expect that other ML training algorithms such as Reinforcement Learning and GANs would obtain similar benefits to CNNs.

Simplicity: Although our proposed additions are relatively simple, LAB still provides significant benefits by intelligently exploiting algorithmic properties. Moreover, LAB seamlessly fits in the existing, per-SM reconfigurable SRAM, which allows programmers to utilize the LAB only when it is useful (unlike prior approaches). Prior approaches (Section 3.6) provide some of the same benefits as LAB, but often require more invasive coherence protocol or consistency model changes [6, 24, 203, 242] or suffer from cache contention [5, 146]. Thus, LAB's simplicity is a strength and demonstrates how the additional complexity of prior approaches is unnecessary, while also improving efficiency over the state-of-the-art (Section 3.4).

Simplicity: Although our proposed additions are relatively simple, LAB still provides significant benefits by intelligently exploiting algorithmic properties.

Moreover, LAB seamlessly fits in the existing, per-SM reconfigurable SRAM, which allows programmers to utilize the LAB only when it is useful (unlike prior approaches). Prior approaches (Section 3.6) provide some of the same benefits as LAB, but often require more invasive coherence protocol or consistency model changes [6, 24, 203, 242] or suffer from cache contention [5, 146]. Thus, LAB's simplicity is a strength and demonstrates how the additional complexity of prior approaches is unnecessary, while also improving efficiency over the state-of-the-art (Section 3.4).

Feature	hLRC[6]	DeNovo[203]	COUP[242]	PHI[146]	RMOs[71, 200]	TS[190, 216]	LAB
No coherence protocol change	X	X	X	✓	✓	X	✓
No memory consistency model change	X	✓	✓	✓	✓	✓	✓
Low degree of atomic L2 traffic	✓	✓	✓	✓	X	X	✓
Reduces atomic serialization penalty	✓	✓	✓	✓	X	X	✓
No overhead for remote invalidations or ownership requests	X	X	✓	✓	✓	✓	✓
Decouple data & atomic accesses	X	X	X	X	X	X	✓
Applied to GPUs	✓	✓	X	X	✓	✓	✓

Table 3.4: Comparing LAB to prior work.

3.6 Related Work

Table 3.4 compares LAB to prior work.

Remote Memory Operations [71, 114, 200, 229]: RMOs send and perform update operations to a fixed memory location or memory controller, and have been used in Cray T3E, NYU Ultra, SGI Origin, and NVIDIA's Fermi GPUs. RMOs avoid contention for cache lines since updates are sent to a fixed, shared memory location. However, this approach increases memory traffic, which hurts performance, and sometimes require programmers to explicitly allocate shared memory locations. In comparison, LAB buffers commutative atomics locally and does not increase network traffic.

Coherence Protocol or Consistency Model Changes [6, 7, 24, 65, 203, 242]: Other work exploited commutativity by extending or modifying the coherence protocol or memory consistency model. Sinclair, et al. extended DeNovo to CPU-GPU systems and showed how obtaining ownership for written data reduced global memory traffic for atomics [7, 203, 204]. Subsequently, hLRC extended DeNovo to only obtain ownership for atomics [6]. Coup [242] and CCache [24] apply similar concepts to CPU coherence protocols and software. Although these approaches provide some of LAB's features, LAB outperforms hLRC (Section 3.4.3), and they significantly change the coherence protocol or consistency model. Moreover, GPU coherence protocols differ significantly from CPU coherence protocols [81, 82, 206], which makes adopting CPU coherence-based techniques like Coup or CCache on GPUs difficult. Similarly, AtomicCoherence makes caching GPU atomics in the L1 easier, but requires coherence and interconnect changes [65]. Finally, GPU timestamp (TS)-based protocols [190, 206, 216] improve efficiency, especially for streaming GPGPU workloads. However, these protocols are write-through or write-no-allocate for stores (and atomics) and block further accesses to the addresses that are written through to the L2, limiting intra- and inter-TB reuse opportunities.

Add Buffers to Caches: AIM uses special instructions to perform aggregation for commutative updates throughout the memory hierarchy [5]. However, similar to hLRC, AIM uses coherence to transfer aggregation updates between remote caches. For workloads with large working sets and high contention, this will hurt performance as discussed in Section 3.4.3. PHI [146] improves commutative access performance without changing the coherence or consistency by buffering and coalescing updates in the cache. As shown in Section 3.4.4, LAB outperforms PHI, since the buffered and data lines are not partitioned, contention from other GPU memory accesses can evict buffered lines prematurely and increase stalls, reducing coalescing and increasing in global memory traffic. In comparison, since LAB utilizes a separate space it is unaffected by data accesses, and thus increases reuse. Additionally, PHI requires both a buffered update bit and bits to identify the atomic operation for the entire cache, whereas the LAB only needs atomic operation bits (Section 3.4.6).

Avoiding Collisions [58]: Egielski, et al. reduced GPU atomic collisions with software optimizations to coalesce atomics [58]. Although this reduces serialization, LAB targets an orthogonal problem: performing atomics locally to reduce serialization. Moreover, their approach could further improve LAB's performance by increasing hits.

ML Training: Prior work also optimized ML training. However, unlike our work, most of this work optimizes the width or number of memory accesses [90, 225, 248], utilizing compression [192], optimizing synchronization in distributed training [240], or rewriting code to keep memory accesses in the register file or shared memory [54, 108, 164, 246]. In comparison, we focus on a different bottleneck – fine-grained synchronization. Moreover, these works are complementary because applying them removes other sources of inefficiency and makes fine-grained synchronization even more important. Prior work also converted fine-grained synchronization into data accesses [49, 154]. This removes atomics, but potentially increases convergence time. In comparison, we make device-scoped atomic accesses cheaper without increasing iterations.

3.7 Conclusion

As GPGPU applications increasingly use fine-grained synchronization, improving device-scoped atomics support is imperative. We exploit the insight that atomic accesses in graph analytics and ML training are commutative, and utilize recent work to identify commutative atomics. Next, we introduce a small, per-SM buffer (LAB) that combines commutative atomics and utilizes reconfigurability to avoid hurting applications that do not use commutative atomics. LAB improves locality for commutative atomics, reduces serialization costs, overall it improves performance 28% energy consumption 19%, and on-chip memory traffic 19% compared to state-of-the-art solutions. Moreover, as GPU vendors move to chiplet-based designs (Section 5), GPU cache hierarchies will get deeper. Consequently, device-scope GPU atomics to globally shared variables will be even more expensive. Thus, LAB will become even more important in these designs.

4 IMPROVING THE SCALABILITY OF GPU SYNCHRONIZATION PRIMITIVES

4.1 Motivation

Chapter 3 showed that programmers can leverage relaxed atomics for global synchronization in certain situations. However, in other use cases it is either not possible or more efficient to use device-wide synchronization primitives such as mutex locks, semaphores, and global barriers [78, 205, 208, 212]. For example, recent work in scientific and high performance computing (including machine learning) use GPUs for persistent kernels [54, 108, 246], buddy allocation [69], and particle partitioning [31] while other work fuses GPU kernels [2, 74, 147] and uses concurrent streams [12, 162]. Unlike traditional, streaming, data parallel GPGPU applications, these applications repeatedly access shared data in global memory across threads from different TBs. Thus, they use device-wide synchronization primitives such as mutex locks, semaphores, and global barriers to avoid data races and ensure correctness. One common HPC algorithm that uses device-wide synchronization is reduction [98]. Listing 4.1 shows a high-level example of how parallel reductions use global barriers for device-wide synchronization, which we use as an exemplar of how GPU applications use device-wide synchronization. This algorithm (discussed further in Section 4.4) iteratively reduces data by assigning each TB on each streaming multiprocessor a subset of the data, which the TB performs a partial reduction on using shared memory (line 4). As the TBs complete their partial reductions, they update the corresponding global output array with the reduced value (line 9). Although each TB reduces an independent portion of the input data, if additional reduction steps remain (line 1), in the next loop iteration TBs will often access on data produced from a different TB in the previous iteration (line 17). Thus, to ensure the updated data is visible to all TBs before the next iteration, a global barrier (discussed further in Section 4.2.1.1)

Listing 4.1: High level example for performing a parallel reduction using explicit device-wide synchronization.

```
while (additionalReductionSteps) {
    // perform local reduction: read from
    // global memory, write to shared memory
    perTB_sharedMem_reduction( global_inputData, sharArr);
    __syncthreads();

    // write reduced output to global memory
    if (tid == 0) {
        global_outputData[blockId] = sharArr[0];
    }
    __syncthreads();
    // Ensure all TBs have finished their portion
    // of the reduction
    globalBarrier();

    // Swap pointers for next reduction step
    swap(global_inputData, global_outputData);
}
```

must be performed to ensure all TBs reach this point before subsequent iterations operate on the data (line 14). Threads generate repeated accesses to these synchronization primitives, and many of the accesses contend with one another. Thus we refer to this as fine-grained synchronization. Although prior CPU work classifies global barriers as coarse-grained synchronization [193], we use fine-grained synchronization to distinguish from the standard GPU definition of coarse-grained synchronization, which often refers to synchronizing via an implicit CPU-side global barrier at the end of a kernel [203, 208, 212, 233].

As discussed in Chapter 1 fine-grained synchronization is an expensive operation for GPUs. Thus it is often a bottleneck for workloads that utilize it (discussed further in Section 4.4). Additionally, synchronization overheads are exacerbated by the level of parallelism on GPUs: GPUs typically run kernels with thousands of threads or more. Thus, synchronization variables are heavily contended, even after common optimizations such as using one master thread per TB to perform the synchronization. Accordingly, it is important to improve support for fine-grained GPU synchronization, especially at high contention levels.

Recent work has also improved software synchronization support. For example, NVIDIA proposed CUDA Cooperative Groups (CCG) [78], a hierarchy of synchronization methods with support ranging from small groups of threads to multi-GPU devices. CCG is tightly integrated with CUDA and performs well, especially at low contention levels. However, as we show in Section 4.5, CCG suffers from high contention for shared synchronization variables as the number of threads the synchronization is performed across increases [241]. NVIDIA also introduced libcu++, which provides fully compatible C++ synchronization support [166]. Researchers have also developed device-wide software barriers, which either use global counters with atomic operations, lock-free synchronization [233], or portable atomics to implement GPU barriers [208]. However, these approaches limit the number of TBs that can be launched on an SM to avoid deadlock. Another popular open source device-wide barrier is HeteroSync's tree barrier [218]. However, HeteroSync uses two consecutive barriers in order to properly support context switching, which significantly increases global atomics to update the shared counter variables [205]. We discuss related work further in Section 4.6.

However, these existing techniques are inefficient as contention increases. Thus we propose optimizations for both global barriers and semaphores. First, we design a multi-level sense-reversing tree barrier (SRB). Although tree barriers and SRBs are widely used in CPUs [83, 218], in Section 4.5 we show that naively applying CPU SRB concepts to GPUs performs poorly. Instead, like prior work on Xeon Phi's [195], GPU SRB algorithms must be designed to fit the GPU's processing model in order to successfully meet scalability demands. Accordingly, our multi-level SRB naturally conforms to the GPU memory hierarchy and scoped consistency model (as discussed in Section 2.2). As a result, most threads use cheap, locally scoped atomics, improving scalability. Second, we optimize GPU reader-writer semaphores to remove scalability bottlenecks. For example, when a writer attempts to exit the semaphore and multiple readers attempt to enter it simultaneously, the contention causes current semaphores to scale poorly and frequently livelock. Thus, we propose a priority mechanism that favors TBs exiting the semaphore. Overall, we make the following contributions:

1. We propose a two-level GPU SRB that reduces atomic transactions required by 50% compared to HeteroSync's two-level tree barrier.

2. We propose a priority mechanism for GPU semaphores, which reduces the time spent on synchronization and eliminates livelock compared to the HeteroSync's reader-writer semaphores, thereby improving scalability.

3. We first evaluate our optimized synchronization primitives across several microbenchmarks. Compared to CPU-style SRBs and HeteroSync, our work improves average performance by 34% on Volta, 38% on Turing, and 27% on Ampere GPUs for barriers, and by 89% for Volta, 68% for Turing, and 90% for Ampere GPUs for semaphores. Moreover, for five larger benchmarks that utilize global barriers on average G-SRB improves performance by 36% on Volta, 34% on Turing, and 32% on Ampere at medium and high contention levels (i.e. greater than threads joining the barrier per SM, where each thread joining the barrier is a master thread per TB) compared to CUDA's CCG. G-SRB also outperforms an optimized CPU-style SRB by 12% on Volta, 7% on Turing, and 8% on Ampere for medium and high contention levels, showing the need for intelligently designing SRBs for GPUs – especially as contention increases.

4. We also propose a hybrid GPU barrier that combines the best of CCG and G-SRB to deliver high performance at all levels of contention.[1]

[1]We have released our code at: https://github.com/hal-uw/. The results in this chapter can be reproduced using code and instructions are available at: https://zenodo.org/record/7264009

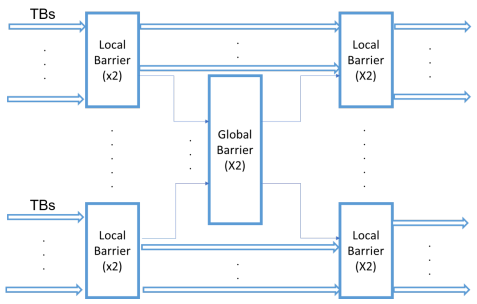


Figure 4.1: Baseline Two-Level Tree Barrier [205] where a per-SM leader TB joins the global barrier.

4.2 Background

4.2.1 GPU Synchronization Primitives

Several recent projects sought to develop a common set of GPU synchronization primitives. For example, Synchronization Primitives (SyncPrims) released a set of mutexes, barriers, and semaphores focused on GPU atomic performance [212]. HeteroSync extended SyncPrims to model memory accesses that required fine-grained synchronization,[2] improve the algorithms, and added both local and globally scoped versions of most algorithms [205]. NVIDIA also introduced CCG [78] and libcu++ [166], which allows kernels to dynamically organize groups of threads across all levels of granularity from a small group of threads in a GPU to a multi-GPU device. These primitives enable new patterns of cooperative parallelism within CUDA, including producer-consumer parallelism. Thus, we next describe HeteroSync's barrier and semaphore and CCG's barrier implementations.

4.2.1.1 Barriers

Figure 4.1 illustrates HeteroSync's two-level centralized tree barrier, which uses a hybrid local-global scope. In a centralized barrier, TBs increment a shared

[2]SyncPrims' critical section memory accesses used local scratchpads, and thus did not require synchronization to ensure ordering.

```
1  __shared__ bool done_local = false;
2  numTBsThisSM = gridDim.x / NUM_SM;
3  __threadfence_block();
4  if (threadIdx == (0,0,0)) { // leader thread
5    // perSMBarr = localBarr1 or localBarr2
6    // inc by 1 with wraparound
7    atomicInc(&perSMBarr, 0x7FFFFFFF);
8    while (!done_local) {
9      if (atomicCAS(&perSMBarr,numTBsThisSM,0)
           == numTBsThisSM) {
10       __threadfence_block();
11       done_local = true;
12 } } }
13 __syncthreads(); // other threads wait here
```

(a) Baseline per SM local barrier.

```
1  __shared__ bool done = false;
2  numJoin = (gridDim.x < NUM_SM) ? gridDim.x :
                                    NUM_SM;
3  __threadfence();
4  if (threadIdx == (0,0,0)) { // leader thread
5    // inc by 1 with wraparound
6    atomicInc(&globBarr, 0x7FFFFFFF);
7    while (!done) {
8      // globBarr = globalBarr1 or globalBarr2
9      if (atomicCAS(&globBarr,numJoin,0)==numJoin) {
10       __threadfence();
11       done = true;
12 } } }
13 __syncthreads(); // other threads wait here
14 if (!done) { DoBackoff(); }
```

(b) Baseline GPU global barrier.

```
1  localBarrier(localBarr1[smID]); // Part A
2  localBarrier(localBarr2[smID]); // Part A
3  if (perSMLeaderTB) {
4    globalBarrier(globalBarr1);   // Part B
5    globalBarrier(globalBarr1);   // Part B
6  }
7  localBarrier(localBarr1[smID]); // Part A
8  localBarrier(localBarr1[smID]); // Part A
```

(c) Overall baseline tree barrier.

Figure 4.2: Pseudo-code for components of baseline GPU two-level tree barrier [205].

counter as they reach the barrier and spin until the counter indicates that all TBs are present. HeteroSync also includes a decentralized, two-level lock-free barrier, which extends a prior decentralized, single-level lock-free barrier [233]. Decentralized barriers trade off increased memory consumption for reduced contention, improved efficiency, and improved scalability. However, unlike our approach (and CCG), they consume significantly more memory. In HeteroSync's implementation all TBs on a SM access unique data before joining a local barrier (lines 1-2, Figure 4.2c). In each TB, assuming no races between threads in the same TB, a single leader thread joins the local barrier (line 4, Figure 4.2a). Next each TB spins (using a per-TB scratchpad variable, `done_local`) until all TBs on the SM join the local barrier (lines 8-11, Figure 4.2a). Once all TBs have reached the local barrier, a statically designated leader TB from each SM proceeds to join the global barrier (Figure 4.2b), while all other TBs join a second group of local barriers (line 7, Figure 4.2c).

However, in order to ensure correctness across potential context switches [3, 174, 220, 227, 228, 231, 234], HeteroSync's tree barrier uses a second barrier (lines 2, 5, 8, Figure 4.2c), similar to a sense-reversing barrier. To do this, HeteroSync passes unique variables for the counts of the two barriers from Figure 4.2a to Figure 4.2c (`localBarr1` and `localBarr2` per SM are passed to `perSMBarr`) and Figure 4.2b (`globalBarr1` and `globalBarr2` are passed to `globBarr`). Without these unique variables, if a TB `i` was context switched out after it joined a barrier and not scheduled again until after the barrier completed, the barrier may deadlock since TB `i` would not be able to distinguish the next barrier from the one it joined previously. Once the expected number of leader TBs reach the global barrier (`numJoin`), these leader TBs join their respective second group of local barriers (line 7, Figure 4.2c, where other TBs from the same SM are spinning). Unfortunately, this approach significantly increases the number of atomics, which adversely impacts performance. Using a tree barrier partially mitigates this overhead by making many of the atomics locally scoped, but tree barriers do not completely mitigate the overhead.

Although CCG is closed source, and thus not all of its implementation details are known, we used NVIDIA's NVBit to disassemble and study the SASS for a Volta `grid_group.sync()` (barrier) [161, 222]. Based on the SASS, CCG appears to use multiple different barrier implementations. Although we could not determine when each barrier is dynamically selected, all of them use a single-level global memory barrier. Specifically, the barrier CCG selected for our configurations (Section 4.4) is an aggressive, single-level global memory barrier, similar to open source barrier implementations [208, 212]. Like HeteroSync, each TB elects a leader thread and the remaining threads spin waiting for the barrier to complete. CCG also uses a sense direction counter that exploits integer overflow to avoid requiring a second barrier to ensure correctness in the presence of context switches. Interestingly, CCG does not appear to perform any backoff,[3] likely because it may increase latency at low contention levels. However, this

[3]Backoff represents a period of time where, after an access fails, a thread waits before attempting to perform the access again. This can significantly improve scalability by reducing contention [17].

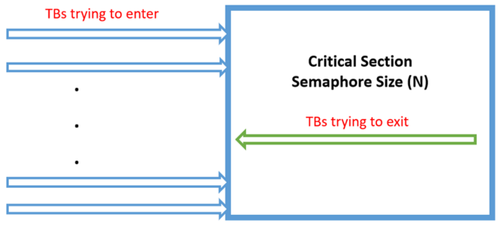


Figure 4.3: Baseline Semaphore Implementation [205].

```
1   __shared__ bool acqLock = false;
2   decAmt = isWriter ? maxSemCnt : 1;
3   if (threadIdx == (0,0,0)) { // leader thread
4     while (!acqLock) {
5       if (atomicCAS(&lock,0,1) == 0) {
6         __threadfence();
7         acqLock = true;
8         if (semSize >= decAmt) {semSize -= decAmt;}
9     } }
10    __threadfence();
11    atomicExch(&lock,0);
12  }
13  __syncthreads(); // other threads wait here
```

(a) Post routine of the baseline GPU semaphore

```
1   __shared__ bool acqLock = false;
2   incAmt = isWriter ? maxSemCnt : 1;
3   if (threadIdx == (0,0,0)) { // leader thread
4     while (!acqLock) {
5       if (atomicCAS(&lock,0,1) == 0) {
6         __threadfence();
7         acqLock = true;
8         // no need to check semSize when exiting
9         semSize += incAmt;
10    } }
11    __threadfence();
12    atomicExch(&lock,0);
13  }
14  __syncthreads(); // other threads wait here
```

(b) Wait routine of the baseline GPU semaphore

Figure 4.4: Pseudo-code for baseline GPU semaphore

approach struggles when contention increases (e.g., as the number of threads or TBs joining a barrier increase) [241].

4.2.1.2 Semaphores

Figure 4.3 provides a high-level overview of HeteroSync's reader-writer semaphores. Each SM has one writer TB that tries to write all the data, and $N-1$ reader TBs that try to read a subset of the data. When a TB tries to enter the critical section (CS, Figure 4.4a `post` sub-routine), it first acquires a mutex lock and checks to see if there is enough capacity in the semaphore for the TB (line 8, Figure 4.4a). If there is capacity, the TB updates the semaphore and releases the lock. Similarly, when a TB leaves the CS (the `wait` sub-routine), it acquires the mutex lock and updates the semaphore (lines 8-9, Figure 4.4b) to remove itself from the semaphore before releasing the lock. When the size of the semaphore is greater than one, multiple reader TBs can enter the CS simultaneously. Unlike on CPUs, which have better OS support, both the post and wait sub-routines must utilize a lock to ensure ordering for accesses to the semaphore count ($semSize$, lines 4-9 in Figure 4.4a and lines 4-10 in Figure 4.4b. Thus, as the number of TBs increases, the mutex lock is a bottleneck and causes contention for reader and writer TBs trying to enter and exit the semaphore simultaneously. For example, a TB leaving the semaphore may not be able to exit if it cannot access the mutex variable because other TBs are repeatedly trying to enter the semaphore. To reduce contention, we can add backoff to make the TBs wait for a short period of time between each unsuccessful acquire.

SyncPrims [212] also contains a lock-free GPU semaphore ("SleepSem") that eschews a mutex. In reader-only semaphores like SyncPrims' studied, this approach had races, but did not affect correctness since all data was read-only. However, the lack of load linked store conditional (LLSC)-type mechanisms in GPUs made this version very livelock-prone for reader-writer semaphores like the ones we are studying. In a reader-writer semaphore, each thead attempting to join the semaphore must a) check if there is space in the semaphore and, once there is space b) update the semaphore to indicate that it has joined. Without GPU LLSC-type memory accesses or a single, combined atomic operation [57], these two operations in a lock-free semaphore must be performed separately. Consequently, the accesses may be *repeatedly* interleaved such that another thread (thread B) may update (and fill) the semaphore in between thread A's a) and b) operations – thus preventing forward progress. Accordingly, we do not include a lock-free semaphore in our study.

4.3 Design

4.3.1 Sense Reversing Barriers

As discussed in Section 4.2.1.1, in HeteroSync's atomic tree barrier every TB joins the local barrier twice (Figure 4.2c). Although HeteroSync's second

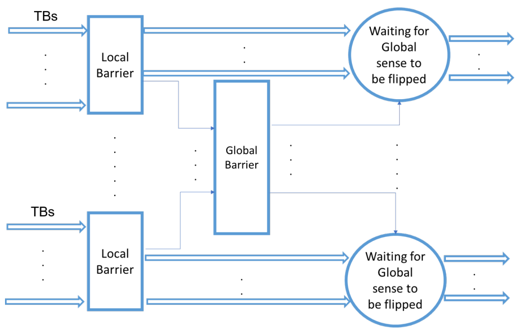

TBs
Local Barrier
Waiting for Global sense to be flipped
Global Barrier
TBs
Local Barrier
Waiting for Global sense to be flipped

Figure 4.5: Overview of GPU Sense Reversing Barrier design with statically elected per-SM leader that joins the global barrier and flips the sense.

```
__shared__ bool s = !(*sense);
numTBsThisSM = gridDim.x / NUM_SM;
__threadfence_block();
if (threadIdx == (0,0,0)) { // leader thread
  // perSMBarr = localBarr[smID]
  // inc by 1 with wraparound
  atomicInc(&perSMBarr,0x7FFFFFFF);
  while (*sense != s) {
    if (atomicCAS(&perSMBarr,numTBsThisSM,0) == numTBsThisSM){
      __threadfence_block();
      *sense = s; // flip sense
 } } }
 __syncthreads(); // other threads wait here
```

(a) Local (per SM) phase of G-SRB.

```
numJoin = (gridDim.x < NUM_SM) ? gridDim.x : NUM_SM;
__threadfence();
if (threadIdx == (0,0,0)) { // leader thread
  // globBarr = globalBarr
  // inc by 1 with wraparound
  atomicInc(&globBarr,0x7FFFFFFF);
  while (*global_sense != *sense) {
    if (atomicCAS(&globBarr,numJoin,0) == numJoin) {
      __threadfence();
      *global_sense = *sense; // flip sense
    } else { DoBackoff(); }
 } }
 __syncthreads(); // other threads wait here
```

(b) Global phase of G-SRB.

```
localSRBarrier(localBarr[smID]); // Fig 6a
if (perSMLeaderTB) {
  globalSRBarrier(globalBarr); // Fig 6b
} else {
  // wait for global barrier to complete
  if (threadIdx == (0,0,0)) { // leader thread
    while(ld_glb_cg(global_sense) != ld_gbl_cg(sense)) { ; }
  }
  __syncthreads(); // other threads wait here
}
```

(c) Overall G-SRB design.

Figure 4.6: Pseudo-code for components of proposed G-SRB

barrier is necessary for correctness, it causes significant overhead (as we show in Section 4.5). To overcome this we design *G-SRB*, a hierarchical SRB [83]. Although SRBs already exist in CPUs, in Section 4.5 we show that directly applying CPU SRB concepts to GPUs results in performance and scalability issues. This also differs from CCG, which uses aggressive, single-level barriers without backoff – which exploit integer overflow to avoid needing a second global barrier (like HeteroSync), but which do not scale well because of high contention for the single, shared global variable. Thus, our key insight is how to design SRBs while taking the GPU's unique memory hierarchy and higher level of parallelism into account (also discussed in Section 4.7).

Figure 4.5 shows a high-level overview of G-SRB. Like HeteroSync, G-SRB utilizes a tree barrier to reduce contention (Figure 4.6). We chose two levels for our tree barrier because this naturally conforms to the memory hierarchy and scoped synchronization of modern GPUs – locally scoped atomics avoid frequent, expensive cache invalidations and flushes for the local, per SM barriers, while the more expensive globally scoped atomics are only necessary for the second level, global barrier [58, 60, 65, 73, 85, 106, 130, 170]. Like Figure 4.2c, G-SRB has two parts: a local barrier per SM (Figure 4.6a) and a global barrier across all SMs (Figure 4.6b). Moreover, like CCG and HeteroSync (Section 4.2.1.1), we elect a leader thread per TB to join the global barrier to reduce contention (Figure 4.6b).

Local Barrier: All TBs on a SM share a sense variable (`sense`) initialized to false the first time the barrier is called. Moreover, each TB also has a local sense variable (`s`) initialized to the inverse of `sense` and stored in the scratchpad to reduce latency. TBs spin until `sense`'s value matches `s` (line 9, Figure 4.6a). As in CPU SRBs, sense matches (i.e., is flipped) when all TBs on a SM join the local barrier. Afterwards, like HeteroSync (lines 4-7, Figure 4.2c), a statically designated leader TB (lines 2-4, Figure 4.6) proceeds to the global barrier. The remaining TBs on each SM wait for the global barrier to complete (lines 6-9, Figure 4.6c). `sense` and `done_local` (Figure 4.2a) are used to identify when the local barrier is complete. However, in combination with `s`, `sense` serves an additional purpose, similar to SRBs in CPUs. If TB `i` is context switched out after it joins the local barrier and is rescheduled after the other TBs on this SM join and exit another local barrier (and flip `sense`), TB `i`'s local copy of `s` will match `sense`. As a result, TB `i` can immediately progress, without causing deadlock as in Section 4.2.1.1.

Global Barrier: All leaders increment a global counter for the global barrier across TBs. When this count reaches the total number of active SMs, all leader TBs have reached the global barrier and the global sense (`global_sense`) is flipped (line 10, Figure 4.6b – like `sense` `global_sense` is initialized to false the first time the barrier is called). Next, these TBs advance to their final local, per-SM barriers (line 5, Figure 4.6c), and the overall barrier completes. Like the local SRB, the global SRB's `global_sense` variable serves a similar purpose to `done` from Figure 4.2b – identifying when all the leader TBs per SM have joined the global barrier. However, by tracking `sense` and `global_sense` separately, any TB that is context switched out will see that `sense` and `global_sense` now match (line 7, Figure 4.6b) and proceed to the next barrier. Moreover, if the other TBs have advanced to the next barrier, as in CPU SRBs, they must wait for this TB to arrive at the corresponding barrier before they can proceed. Thus, unlike HeteroSync G-SRB does not require a second set of barriers to ensure correctness on context switches (Figure 4.6c). Furthermore, G-SRB retains the benefits of HeteroSync's tree barrier by keeping most thread's atomics locally scoped.

4.3.1.1 Hybrid Barrier

G-SRB's two-level barrier and use of locally scoped atomics helps reduce contention as the number of threads joining the barrier scale. However, at low contention levels G-SRB requires additional atomics relative to a single-level global barrier like the one CCG uses. Conversely, CCG's single-level global barrier does not scale very well as the the number of threads or TBs in the program increases [241], because a single level barrier without backoff can result in added contention, especially when many threads or TBs join the barrier. Although some optimizations, such as eliding G-SRB's local barriers when there is a single thread (from a single TB) per SM, are simple, a more general approach is needed. Accordingly, since each design is optimized for different design points, we also create a hybrid global barrier implementation that uses CCG at low contention

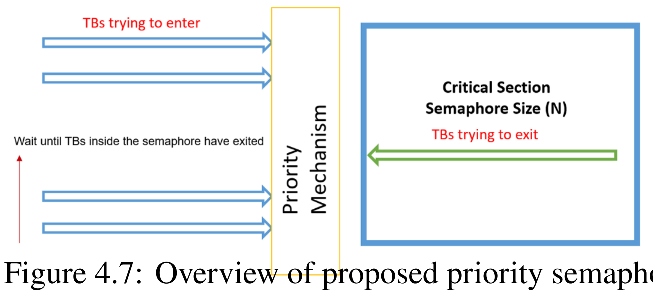


Figure 4.7: Overview of proposed priority semaphore.

levels and G-SRB at medium and high contention levels to examine the benefits of integrating our design into future libraries.

4.3.1.2 Over-subscription

Our proposed algorithms require that all synchronizing TBs must be scheduled on the GPU simultaneously. If that is not the case (e.g., because the GPU is oversubscribed), the algorithms would need to be adjusted. For example, the grid would need to be divided into chunks of TBs that can run simultaneously on the GPU, and these TBs would need to all synchronize before the next chunk of work could run. However, we leave this as future work.

4.3.2 Semaphores

Semaphores allow multiple TBs to enter the CS simultaneously based on the semaphore size. Typically, to avoid data races semaphores either require each thread to access unique data or only allow readers in the semaphore simultaneously. As discussed in Section 4.2.1.2, due to the lack of OS and hardware support in modern GPUs, current semaphore implementations often use mutex locks to prevent multiple threads from updating the semaphore count simultaneously. However, this centralized mutex is highly contented in GPU programs and creates a bottleneck for threads attempting to exit the semaphore, since they contend with threads acquiring the lock while trying to enter the semaphore. We demonstrate that this approach livelocks when the number of threads accessing the semaphore

```
__shared__ bool acqLock = false;
decAmt = isWriter ? maxSemCnt : 1;
if (threadIdx == (0,0,0)) { // leader thread
  while (!acqLock) {
    // wait if threads exiting semaphore
    while (atomicCAS(&priority,0,0) != 0) {
      DoBackoff();
    }
    if (atomicCAS(&lock,0,1) == 0) {
       __threadfence();
       acqLock = true;
       if (semSize >= decAmt) { semSize -= decAmt; }
   } }
   __threadfence();
   atomicExch(&lock,0);
 }
 __syncthreads(); // other threads wait here
```

(a) Post routine of the proposed GPU priority semaphore

```
__shared__ bool acqLock = false;
__shared__ bool setPriority = false;
incAmt = isWriter ? maxSemCnt : 1;
if (threadIdx == (0,0,0)) { // leader thread
  while (!acqLock) {
    if (atomicCAS(&lock,0,1) == 0) {
      __threadfence();
      acqLock = true;
     // no need to check semSize when exiting
       semSize += incAmt;
     } else {
       if (!setPriority) {
         atomicOr(&priority,1);
         setPriority = true;
   } } }
   __threadfence();
   atomicExch(&lock,0);
   atomicAnd(&priority,0);
   setPriority = false;
 }
 __syncthreads(); // other threads wait here
```

(b) Wait routine of the proposed GPU priority semaphore

Figure 4.8: Pseudo-code for baseline GPU semaphore

scaled beyond 32 or 64 (Section 4.5.2).[4] Even software backoff does not solve this livelock, because it does not guarantee that threads trying to exit the semaphore can obtain the lock in a timely fashion.

Thus, we propose to add a priority mechanism to prioritize threads exiting the semaphore (*PriorSem*). This helps ensure forward progress and reduces the serialization penalty resulting from multiple attempts at acquiring the lock variable by a group of TBs in which some are trying to exit while others enter the semaphore. Although this approach temporarily favors threads exiting the CS, it does not cause starvation or unfairness in the long run because it is solely focused on releasing already held resources, not acquiring them – by prioritizing threads exiting the semaphore, other threads are able to enter the semaphore sooner. Figure 4.7 shows a high-level overview of the proposed design, and Figures 4.8a and 4.8b show the post and wait components of the proposed priority semaphore, respectively. Prioritizing the threads exiting the CS does not impact the TB or warp schedulers. Instead, our priority mechanism has other threads perform backoff instead of attempting to enter the CS (lines 5-8, Figure 4.8a. Once at least one thread exits the CS (lines 16-19, Figure 4.8b, the semaphore is no longer full so the priority flag is unset, and the other threads resume attempting to enter the CS again (lines 4-13, Figure 4.8a). These changes are the key differences from Section 4.2.1.2.

4.4 Methodology

4.4.1 System Setup

We study our proposed algorithms (Section 5.3) on three NVIDIA GPUs: Titan V [168], RTX 2080Ti [160], and RTX 3090 [119] We focus on desktop-class GPUs because they have significant parallelism and are widely used. Table 4.1

[4]Recent work uses similar GPU reader-writer semaphores [144]. Although we have not found significant differences in their post and wait routines compared to Figures 4.4b and 4.8a, it scales to more GPU threads. However, our approach still improves performance over this style of reader-writer semaphore (Section 4.5.2).

GPU Feature	**Titan V**	**RTX 2080Ti**	**RTX 3090**
Architecture	Volta	Turing	Ampere
# SMs	80	68	82
# CUDA Cores/SM	64	128	128
Max TBs/SM	32	16	32
Process	12 nm	12 nm FFN	8 nm

Table 4.1: GPUs used with their relevant system parameters.

lists the system configurations. We use CUDA 11 for all experiments [157]. Since GPUs do not have a dedicated, low latency OS, all of our algorithms are written at the user-level and do not involve the OS. We ran all benchmarks repeatedly with different input sizes to empirically ensure deadlock did not occur. Moreover, to ensure correctness we also check the outputs of each benchmark.

4.4.2 Benchmarks

We use barrier and semaphore microbenchmarks and benchmarks to evaluate our proposed algorithms. The microbenchmarks allow us to easily compare different algorithms and CS sizes (represented by number of instructions). This version of the paper does not show results for the variation of CS sizes. For the baseline microbenchmarks, we use HeteroSync's tree barrier and semaphores with and without exponential software backoff, as described in Section 4.2. Since HeteroSync extends SyncPrims, we do not include a separate comparison against SyncPrims [212]. Table 4.2 lists the synchronization primitive microbenchmarks and benchmarks used in the experiments.

Barriers: We compare against CCG's barrier[5] and a CPU-style SRB: *G-CPUSRB*, which we optimize to reduce contention by electing a leader thread per TB to join the global SRB (having all threads join the global SRB, like CPU algorithms do, resulted in extremely poor performance). Thus, the main difference between

[5]We also compared against libcu++'s barrier implementations but found that CCG always provided equivalent or better performance. After discussing this finding with one of the developers, we believe CCG outperforms libcu++ because libcu++ focuses on full compatibility with C++, and thus cannot always optimize GPU code as well as CCG.

Applications	**Description**
Microbenchmarks	
Barriers	
atomTreeBarr[205] (Baseline)	Two-level atomic tree barrier.
G-SRB	Proposed two-level tree SRB.
G-CPUSRB	Two-level CPU-style tree SRB.
CCG[78]	NVIDIA's CCG barrier.
Hybrid	CCG (low); G-SRB (medium/high contention).
Semaphores	
SpinSem1, 10, 120[205]	Semaphore with size [1, 10, 120].
SpinSemEBO1, 10, 120[205]	SpinSem* with exponential software backoff.
PriorSem1, 10, 120	SpinSem* extended with proposed priority flag.
PriorSemEBO1, 10, 120	PriorSem* with exponential software backoff.
Benchmarks	
BFS[29]	Graph traversal algorithm.
PageRank[29]	Ranks search engine website results.
Parallel Scan[201]	Element `i` is sum of all elements up to `i`.
Reduce[98]	Reduces array elements into one result.
SSSP[29]	Computes shortest path of each node from a source node.

Table 4.2: Microbenchmarks and benchmarks studied.

G-SRB and G-CPUSRB is that all TBs per SM join the global barrier in G-CPUSRB, where in G-SRB we select a leader TB per SM to join the global barrier. Finally, we also compare against a hybrid global barrier (*Hybrid*), as discussed in Section 4.3.1.1.

We also compare the performance of each barrier for five modern GPGPU benchmarks [29, 32], which are representative of larger GPU programs that use barriers. BFS, SSSP, and PageRank are widely use in graph analytics, which often utilize GPUs [6, 29, 37, 171, 204, 226]. Since prior work has shown that different graph analytics algorithms provide the best GPU performance in different situations [112, 199, 209], we focus on implementations for these algorithms that require explicit synchronization. Similarly, performing Scan and Parallel Reduce on GPUs are foundational building blocks in Computing Aided Engineering (CAE), including computational fluid dynamics (CFD) [138], finite element analysis [110], multibody dynamics [137, 176, 219], and granular dynamics (GD) [139, 175]. Many CAE simulations repeatedly compute the norm-i of a vector, be it norm-1, norm-2 (Euclidean), or infinite norm, which are all Parallel Reduce operations. For instance, computing the kinetic energy of a granular system in its flow or the largest value of the residual in the momentum balance equation in the Finite Element Analysis method require a Parallel Reduce operation via the "$+$" or "max" operator, respectively. Likewise, any collision detection task in GD simulation requires a Scan operation at each time step (in one second of physics, there are approximately one million scans performed, which are sometimes performed on arrays as large as 50 million entries). Any Lagrangian method used for CFD also requires a Scan operation to determine the number of neighbors each fluid particle interacts with. Finally, both Scan and Parallel Reduce operations are used in gaming, performed in many cases at each time step of the simulation. Thus, Scan and Parallel Reduce are two of the most common operations done in CAE and gaming, and these operations often utilize GPUs to take advantage of their parallelism. For Scan and Parallel Reduce we use the CUDA Samples implementations as references, but replace the implicit CPU-side barrier with explicit grid synchronization [32]. As Zhang, et al. also observed, we saw comparable performance between the two methods [241].

Each thread in the barrier microbenchmarks accesses unique memory locations. To model various contention levels (TB/SM discussed in Section 4.4.3) we use a variety of input graphs [22, 29, 53, 196] for BFS, SSSP, and PageRank: bgg.gr (abbreviated 1-bgg, 1 TB/SM), USA-road-d.NY (2-NY, 2 TBs/SM), soc-academia (4-SA, 4 TBs/SM), USA-road-d.FLA (4-FLA, 4 TBs/SM), USA-road-d.W (8-W, 8 TBs/SM), web-google (8-WG, 8 TBs/SM), CoAuthorsDBLP (8-CO, 8 TBs/SM), and USA-road-dUSA-road-d.USA (16, 16 TBs/SM). Similarly, for Scan and Parallel Reduce, for maximum contention we use 163840 elements and halve the elements for each lower contention level.

Semaphores: Although some GPU workloads have started using semaphores [69], to the best of our knowledge they are not publicly available. Thus, we compare the baseline HeteroSync semaphores against our proposed semaphores (*PriorSem*). We also attempted to compare against libcu++'s semaphores, but its binary and counting semaphores do not support reader-writer semaphores. For all semaphores, the readers each access a fraction ($1/N$) of the global memory data, while the writers write all N memory locations.

4.4.3 Configurations

For all experiments, we report steady state behavior by running the benchmarks in Table 4.2 ten times and averaging the results. The number of threads per TB varies across benchmarks. However, since all variants we study (including the CCG barriers we used) use an optimization where a single thread per TB joins the barrier or semaphore, we express contention in terms of the number of TBs per SM (TBs/SM). As the number of TBs/SM increase, contention for the synchronization primitives also increases, regardless of number of threads per TB. We set the maximum exponential backoff to 1024 for all results, because we empirically found this provided a good balance of reduced contention without sleeping too long.[6] Moreover, to ensure each SM has the same number of TBs, we utilize NVIDIA's RR scheduler arbitration scheme to schedule TBs across SMs.

For the barrier implementations, these accesses use the barriers to ensure there are no data races, while for the semaphores these accesses are all performed in the CS. Although our primary focus is high contention cases, we also use weak scaling [72] to examine the effect of contention. Here, we hold the CS size constant (at 100 global loads and stores) and vary the number of TBs from 1 TB/SM to 32 TBs/SM for Volta, and from 1 TB/SM to 16 TBs/SM for Turing since Turing allows a maximum of 16 TBs/SM. We show the runtime for 10 iterations of the micro-benchmark across all data points.

[6]We also examined inlined assembly with CUDA's `nanosleep` PTX instruction. The results showed $< 2\%$ improvement over software backoff.

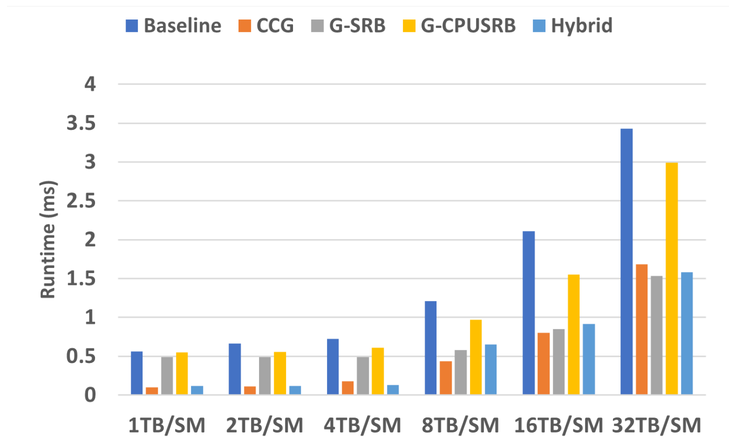
For the semaphore, we compare the baseline's (with and without exponential backoff) and proposed implementation's (with and without exponential backoff) execution times across different contention levels. We create these different contention levels by varying either the semaphore size, number of TBs/SM, or number of load and store instructions while keeping the other two constant. For example, we use semaphore sizes of 1 (single reader or writer in CS), 10 (up to 10 readers or 1 writer in CS), and 120 (up to 120 readers or 1 writer in CS), as in prior work [205, 212], to model different reader-writer ratios.

Finally, we evaluated synchronization and non-synchronization breakdown for all the microbenchmarks, to analyze how well our optimizations reduce the number of atomics and global synchronization for the barriers, and the degree of CS contention for the semaphores. To estimate this percentage we measured the number of clock cycles spent in the synchronization call relative to the number of clock cycles spent in the entire kernel, averaged across all TBs.

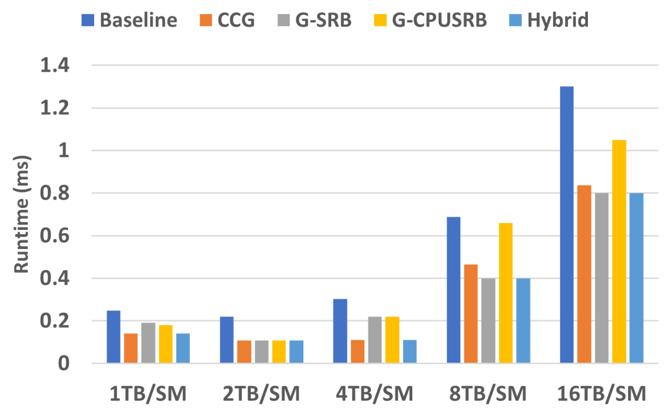
4.5 Evaluation

4.5.1 Barriers

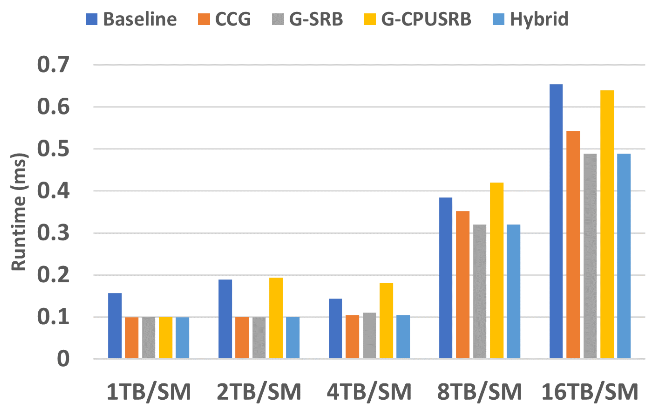
Microbenchmarks: Figure 4.9 compares the barrier's execution times as the number of TBs vary. Overall, CCG and our sense reversing barrier (*G-SRB*) significantly outperform the baseline. The CPU-style SRB (*G-CPUSRB*) also outperforms the baseline but is less efficient than CCG and G-SRB, especially at high contention levels. On average, G-SRB improves performance by 34% on Volta, 38% on Turing, and 27% on Ampere while G-CPUSRB improves performance by 15% on Volta, 26% on Turing, and 3% on Ampere respectively, compared to the baseline. By removing the second barrier call, G-SRB significantly reduces the number of atomic accesses versus the baseline by over 50%. G-SRB's gains over the baseline also increase as contention increases; although the baseline also uses a tree barrier, its additional atomics (relative to G-SRB) hurt



(a) Volta Architecture



(b) Turing Architecture



(c) Ampere Architecture

Figure 4.9: Execution time for barrier microbenchmarks as contention increases.

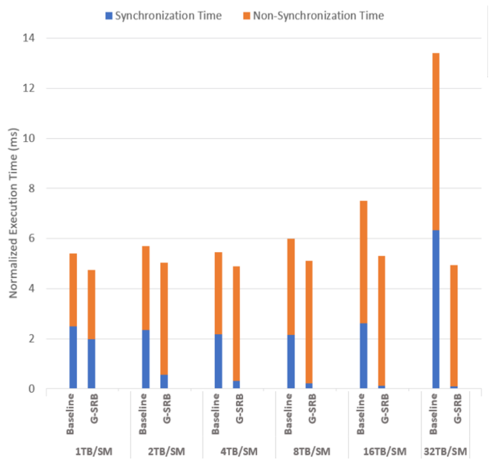


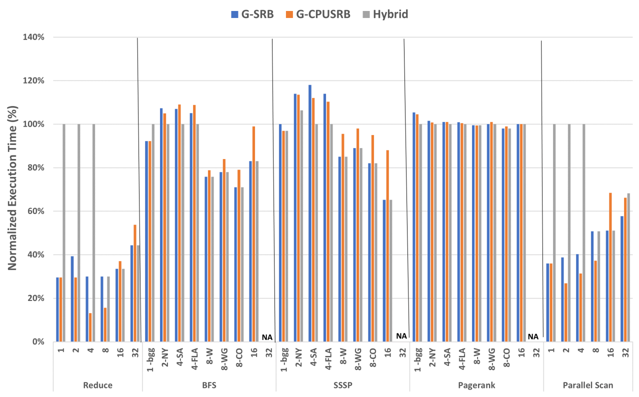
Figure 4.10: Sync versus non-sync time for Volta barriers.

performance. Thus, G-SRB improves scalability. To further examine why G-SRB improves on the baseline, Figure 4.10 breaks down their execution time into synchronization and non-synchronization time for the Titan V GPU. On average, G-SRB reduces synchronization time by 78% over the baseline. Moreover, non-synchronization time only increases a little for G-SRB as contention increases. Thus, the synchronization time reduction is not replaced by power-hungry active waiting.

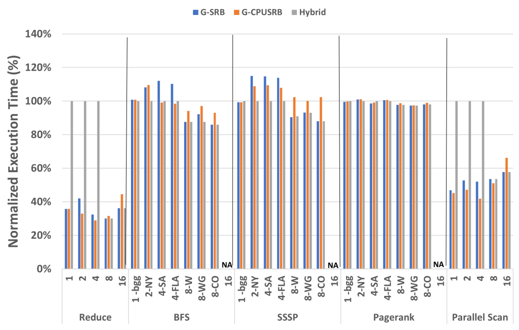
CCG also provides significant benefits over the baseline. Notably, CCG outperforms G-SRB at lower contention levels because at low contention levels G-SRB's software backoff and hierarchical design offer less benefit (Section 4.3.1.1). Although Ampere provides better hardware support for barriers, this support focuses on barriers that can use shared memory [167], which our barriers cannot use since they synchronize across multiple TBs which are often running on different SMs. G-CPUSRB also provides similar benefits at lower contention levels. However, as contention increases, G-SRB's more scalable design approaches and then outperforms CCG and G-CPUSRB. Thus, *Hybrid*, which utilizes CCG at low contention levels and G-SRB at medium and high contention levels, provides high performance regardless of contention level. We set *Hybrid*'s threshold in between selecting CCG and G-SRB at 8 TBs/SM, as it provides a good transition point to switch between the two approaches. Moreover, since CCG, G-CPUSRB, G-SRB, and Hybrid significantly outperform the baseline, we focus on these barriers for the full-sized benchmarks.

Benchmarks: Figure 4.11 examines the barrier's normalized performance as contention varies for the five full-sized benchmarks. Although Figures 4.9 and 4.10 used absolute execution time in – to demonstrate how the performance scaled as contention increased, Figure 4.11 uses normalized performance because each benchmark's runtime differs significantly. Similar to the microbenchmarks, at lower contention levels CCG often outperforms G-SRB. However, unlike the microbenchmarks G-SRB consistently outperforms CCG and G-CPUSRB at much lower levels of contention (e.g., 8 TBs/SM). This happens because G-SRB's locally scoped atomics reduce the number of global flushes and invalidations, which improves performance relative to CCG, which must globally flush and invalidate more frequently due to its single-level design. As discussed in Section 4.2.1.1, every atomic in CCG performs a `threadfence` that flushes and invalidates (Section 4.2). In comparison, in G-SRB only one TB per SM that takes part in the second level barrier (Figure 4.6b) must perform a `threadfence`, and the remaining TBs can perform the cheaper `threadfence_block` at the local, per-SM barriers (Figure 4.6a). This difference is further magnified since CCG does not perform backoff. Thus, as contention increases, G-SRB reduces unnecessary accesses to the shared global synchronization variables – which is magnified by the degree of synchronization in some workloads. For example, usually the gains were slightly lower for social media and similar for web networks and road networks at the same contention level because the social media networks require less synchronization.

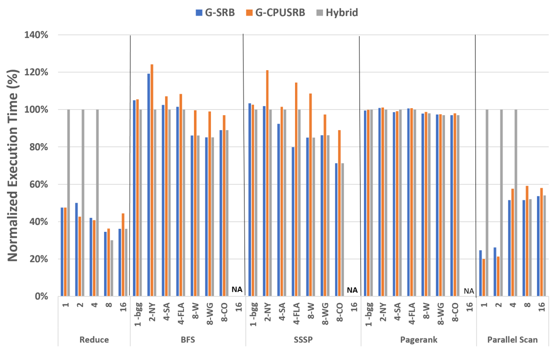
For all five benchmarks, G-SRB's performance improvement over CCG is closely tied to the percentage of total execution time spent performing global synchronization (Section 5.4.3). As expected, global synchronization is a major component in *Reduce* and *Scan* where on average 90% and 82%, respectively, of



(a) Volta Architecture



(b) Turing Architecture



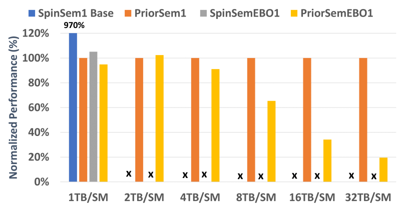
(c) Ampere Architecture

Figure 4.11: Benchmark's with barriers performance as contention varies, normalized to CCG. NA's denote configurations the benchmark cannot run due to input size. Since Baseline is much worse for the microbenchmarks we do not include it here.

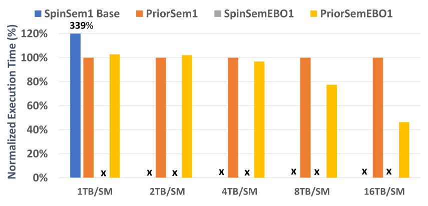
kernel runtime is spent synchronizing at the maximum contention level. As a result, G-SRB outperforms CCG by an average of 62% and 49%, respectively, for Reduce and Scan at contention levels $\geqslant$ 8 TBs/SM. In contrast, on average PageRank spends only 3% of kernel time synchronizing. Consequently, G-SRB only improves performance by 0.1% on average over CCG. BFS and SSSP represent a middle point between Reduce, Scan, and SSSP: they spend 36% and 47%, respectively, of their time synchronizing at maximum contention. As a result, CCG mostly performs better than G-SRB for lower contention levels (e.g., 3% and 8% better for BFS and SSSP, respectively, on average at contention levels $<$ 8 TBs/SM). For these lower contention levels G-CPUSRB also does worse than CCG but is slightly better than G-SRB. Since G-CPUSRB has a more centralized design, at lower contention levels, scalability is less important, so G-SRB's additional local barriers add overhead relative to G-CPUSRB. However, as contention increases, G-SRB's improved scalability again enable it to outperform CCG and G-CPUSRB: at contention levels $\geqslant$ 8 TBs/SM G-SRB is 19% and 19% better than CCG on average across the three GPUs, respectively, for BFS and SSSP. Interestingly, for high contention levels G-CPUSRB also often outperforms CCG. Finally, Hybrid again gives the best performance in almost every case, effectively blending CCG and G-SRB. Note that the full-sized benchmarks utilize a wide variety of input sizes, including ones that cause higher contention. Moreover, we expect that input sizes will continue to increase in the future. Thus, identifying more scalable synchronization solutions is important.

4.5.2 Semaphores

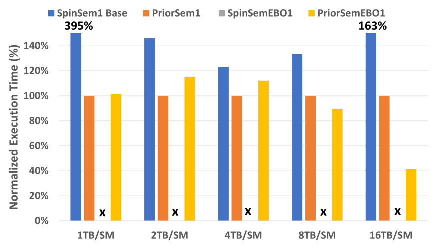
Microbenchmarks: Figures 4.12 shows the execution times of semaphore algorithms across different contention levels ranging from 1 TB/SM to the maximum TB/SM across different GPU architectures (Table 4.1). We create these different contention points by either varying the number of TBs/SM, or number of load and store instructions while keeping the other two constants. Overall, on average for semaphore of size 1 our proposed implementation improves



(a) Volta Architecture

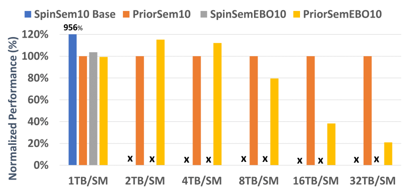


(b) Turing Architecture

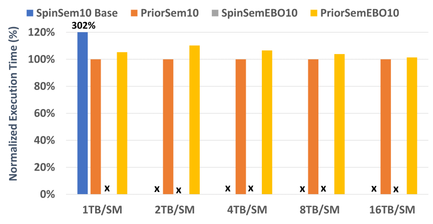


(c) Ampere Architecture

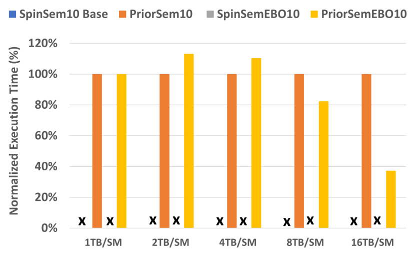
Figure 4.12: Size 1 semaphore's execution time as contention increases, normalized to *PriorSem1*. X's denote deadlocks.



(a) Volta Architecture

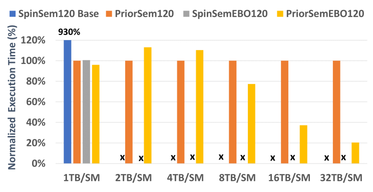


(b) Turing Architecture

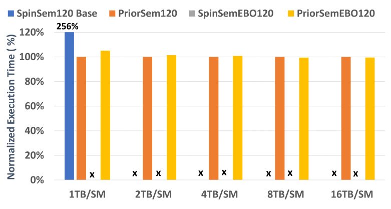


(c) Ampere Architecture

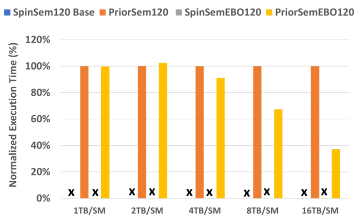
Figure 4.13: Size 10 semaphore's execution time as contention increases, normalized to *PriorSem1*. X's denote deadlocks.



(a) Volta Architecture



(b) Turing Architecture



(c) Ampere Architecture

Figure 4.14: Size 120 semaphore's execution time as contention increases, normalized to *PriorSem1*. X's denote deadlocks.

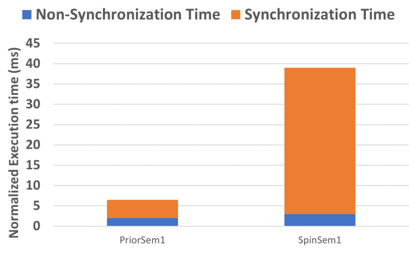
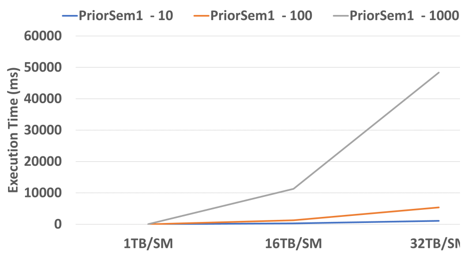
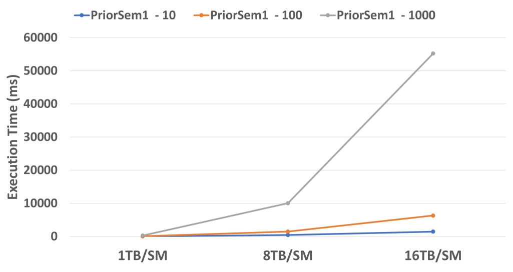


Figure 4.15: Semaphore sync versus non-sync time for 1 TB/SM.

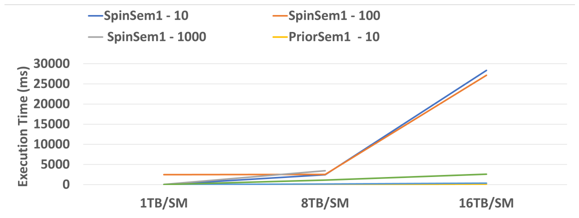
performance by 89% over the baseline implementation on the Volta, 70% on the Turing GPU, and 90% on Ampere GPU. The baseline implementation with exponential backoff shows much better performance and is only 5% slower on the Volta GPU for a semaphore of size 1. However, *SpinSem* livelocks as contention increases for its centralized semaphore on the Volta and Turing GPUs, and is so significant that it always deadlocks on the Ampere GPU. As a result, we were only able to obtain *SpinSem* data for 1 TB/SM on the Volta and Turing GPUs. Although *SpinSemEBO* reduces contention compared to *SpinSem*, it also livelocks for $\geqslant$ 1 TB/SM. In comparison, *PriorSem* and *PriorSemEBO* avoid livelock for



(a) Volta Architecture



(b) Turing Architecture



(c) Ampere Architecture

Figure 4.16: Semaphore's execution time as CS size varies. The labels show the microbenchmark name suffixed with the CS size. We do not include the baseline semaphores because they deadlock beyond 1 TB/SM.

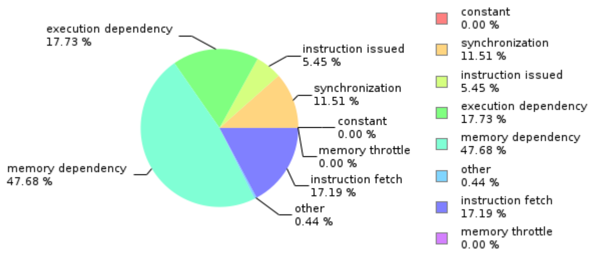


Figure 4.17: GPU kernel profile for the baseline semaphore.

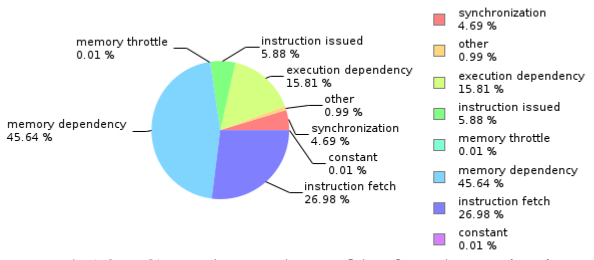


Figure 4.18: GPU kernel profile for the priority semaphore.

Feature	HeteroSync Tree Barrier [204]	CPU-style SRBs	CCG [78]	G-SRB
Scalable, low contention synch variables	✓	✓	X	✓
Optimized for GPU processing model	✓	X	✓	✓
Efficient context switching support	X	✓	✓	✓

Table 4.3: Comparing G-SRB to prior work.

all the evaluated data points. Thus, the priority mechanism successfully prevents TBs entering the semaphore from stopping TBs trying to exit the semaphore and ensures forward progress. Moreover, the GPU kernel activity profiles (Figures 4.17 and 4.18) show that the priority mechanism also reduces redundant attempts to acquire the lock, reducing the overall synchronization time from 12% in baseline to 5% in *PriorSem*. Further, Figure 4.15 shows the priority mechanism also reduces CS synchronization time by 88% over the baseline.

Varying CS Size: Finally, like G-SRB, we evaluate the semaphores across different CS sizes and for different numbers of TBs/SM. The semaphores (Figure 4.16) and G-SRB have similar trends: as CS size increases the overall gains decrease because the percentage of time spent in synchronization decreases as we increase the size of the CS. Overall, these results show that our proposed modifications improve performance and avoid livelock.

4.6 Related Work

CPU Synchronization Primitives: CPUs have long utilized efficient synchronization primitives for fine-grained synchronization, including centralized and decentralized mutexes [17, 140, 142, 177, 178, 197], ticket locks [79, 121], barriers [83, 218], and semaphores [55]. Modern OSs often support these synchronization primitives [63]. For example, a simple realization of a CPU mutex is a spin lock implemented with atomics [191, 194, 217]. However, these synchronization primitives often do not scale well on GPUs because GPUs do not have robust OS support and most CPU spin lock primitives are implemented using linked data structures which causes warp divergence [232]. Further, as threads increase these mutexes heavily use globally scoped atomics that degrade performance (Section 2). Moreover, GPU applications traditionally use fine-grained synchronization sparingly, and the level of GPU parallelism necessitates simpler coherence protocols and less OS involvement.

GPU Synchronization Primitives: SyncPrims and HeteroSync developed GPU synchronization primitives microbenchmarks, including locally and globally scoped atomic variants [204, 212]. However, as shown in Section 4.5, HeteroSync's barrier and semaphore implementations scale poorly and suffer from livelock. The proposed approaches address these shortcomings, scale better, and perform significantly fewer atomics. Prior work has also shown that ticket locks can be used instead of mutex locks in some GPU algorithms to improve scalability [151, 152, 205]. However, we focus on mutex locks because they are more commonly used in GPU applications and because ticket locks imply an ordering on when a TB accesses the CS that was not implied in either CCG or the benchmarks. Nevertheless, this is an interesting alternative for future work.

Other prior work dynamically estimates a GPU kernel's occupancy for barriers [208]. This approach restricts the number of TBs based on a discovery protocol estimate but guarantees fair scheduling and ensures deadlock-freedom. Finally, GPU manufactures such as AMD and NVIDIA have started to add hardware support for inter-block synchronization [11, 119, 127, 167]. However, this hardware support is limited in scope and does not significantly impact our results on NVIDIA GPUs.

Overall, Table 4.3 compares G-SRB to prior work on synchronization primitives across three key metrics: scalability of accessing synchronization variables as contention increases, compatibility with the GPU's processing model, and efficient support for context switching. As discussed above, HeteroSync was designed both for the GPU's processing model and with scalability in mind, but requires two barriers to support context switches. In contrast, CPU-style SRBs efficiently support context switching and scale efficiently, but perform poorly when applied to GPUs (as we show in Section 4.5). Finally, CCG is designed explicitly for GPUs and efficiently supports context switches (Section 4.2.1.1), but suffers scalability issues as the number of TBs joining the barrier increase

(Section 4.5).

Accelerator Synchronization Primitives: Other prior work has also explored how to design optimized barriers for Xeon Phi co-processors [195]. Like our work, Rodchenko, et al. identify the importance of designing multi-level barrier algorithms with the accelerator's memory hierarchy in mind. However, designing barriers for Xeon Phi's and GPUs requires different considerations because the accelerators have different memory hierarchies, coherence protocols, consistency models, and access patterns. For example, Xeon Phi barrier algorithms must consider inter-core communication since atomics may be performed locally. In comparison, both locally- and globally-scoped GPU atomics must be performed at the last level cache. Thus, GPU barrier algorithms utilize different optimizations. Moreover, unlike our work, Rodchenko, et al. did not explore semaphore optimizations.

4.7 Discussion & Conclusion

Modern GPU applications increasingly utilize fine-grained synchronization. However, existing solutions scale poorly or suffer from livelock at high contention levels. We propose optimizations to state-of-the-art GPU barriers and semaphores. Although our techniques extend well known approaches like SRBs, we demonstrate how utilizing these concepts on GPUs requires careful consideration of the GPU memory hierarchy, coherence protocol, consistency models, and threading model. Directly applying CPU-style SRBs is sub-optimal, especially as contention increases. We show how synchronization primitives should be designed with the GPU's parallelism, memory hierarchy, and consistency models in mind. We design G-SRB accordingly, utilizing a two-level barrier that scales efficiently by mirroring the GPU's memory hierarchy and exploiting the GPU's scoped consistency to frequently perform synchronization locally. We also utilize the GPU's shared memory to spin on local, per-TB variables to improve performance – without affecting SM utilization because each thread only store a couple of booleans (e.g., Figure 4.6a) in the shared memory.

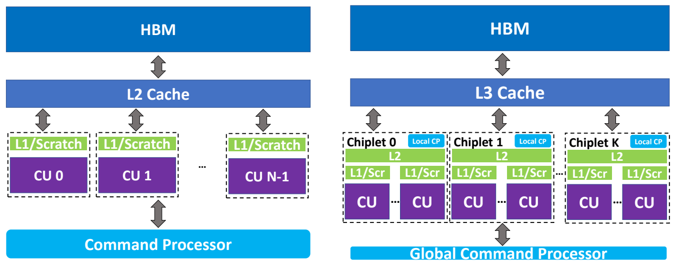
Overall, our algorithms significantly improve the performance and scalability of GPU barriers and semaphores for Volta, Turing, and Ampere GPUs, and avoid livelocks. On average our proposed techniques improve performance by 36% relative to the baseline tree barrier and 79% relative to the baseline semaphore. Since our barrier scales better than CCG and CPU-style SRBs, at levels of contention $\geqslant$ 8 TBs/SM, for five full-sized benchmarks, it outperforms CCG by 36% on Volta, 34% on Turing, and 32% on Ampere; likewise it outperforms CPU-style SRBs by 12% on Volta, 7% on Turing, and 8% on Ampere GPUs. Collectively, this makes a case for G-SRB to replace CCG for higher levels of contention. Moreover, although we focus on single GPU synchronization, the ideas are also easily extensible to multi-GPU and multi-chiplet setups by adding an additional hierarchy level to keep most synchronization local per GPU.

5 CPCOH: EFFICIENT MULTI-CHIPLET GPU COHERENCE VIA DEPENDENCY TRACKING

5.1 Motivation

Thus far we have seen bottlenecks with explicit fine-grained synchronization using software primitives in Chapter 3 and Chapter 4. However, implicit fine-grained synchronization is another important aspect that comes with its own sets of bottlenecks. These bottlenecks have gotten even worse as GPUs have continued to evolve. The search for more computer power on the device while still meeting modern technology constraints has led GPU vendors to redesign the memory hierarchy, however, this redesign introduces new sets of challenges that need to be addressed to ensure optimal performance. Newer GPU architectures introduce additional levels of cache hierarchy in the system while also making some caches that were shared (L2 cache) now private to a subset of CUs. However, this increases the performance impact of implicit global synchronization that happens at kernel boundaries. GPU's simple software-driven coherence protocols rely on acquire/release semantics at kernel boundaries to maintain coherence. The more levels of private cache in the device, the greater the impact of implicit synchronization on application performance. Performance is affected because inter-kernel reuse becoming more difficult. For example, in monolithic GPUs application could reuse data across kernels from the L2 cache. However, in chiplet-based designs, L2 caches are private. Thus, they are also subject to invalidation (acquire) and writeback (release) due to implicit synchronization – eliminating inter-kernel reuse from the L2 caches and hurting application performance.

For decades transistor scaling allowed vendors to fit an order of magnitude more transistors on a die per technology generation. For example, modern GPUs often have a hundred compute units (CUs) per GPU, quadruple the number of CUs from the previous decade [105], and run applications with millions or billions



(a) Monolithic. (b) Proposed chiplet-based.

Figure 5.1: Overall heterogeneous system.

of threads. However, continuing to scale performance and energy efficiency for future heterogeneous systems is hampered by technology scaling and Moore's Law slowing [102]. General-purpose CPUs and accelerators also often have different timing, density, and bandwidth requirements. Thus, integrating them on a single die is difficult. Moreover, cost, die, and yield limitations make designing larger, monolithic systems difficult [20, 101, 105, 173].

Recent research has combined multiple smaller chips into a large, aggregated system, an approach known as multi-chip modules (MCMs) or chiplets [20, 21, 66, 105, 143, 202, 221], as shown in Figure 5.1. Likewise, industry has demonstrated how chiplet-based CPUs (e.g., AMD's Epyc [148]) can continue scaling performance. Since the chiplets are smaller, they do not face the same die and yield challenges as monolithic systems. Moreover, combining multiple chiplets together (e.g., using interposers [94, 100, 101, 133] or other packaging technologies [198]) into a single, larger system improves memory bandwidth, memory capacity, and I/O scalability [91, 92, 105, 169, 207, 221]. This enables closer integration of components than was previously possible, without the technology integration challenges monolithic designs faced.

However, chiplet-based heterogeneous systems introduce an additional layer of hierarchy, causing indirection and non-uniform access latency (NUMA) effects that significantly hurt performance. In particular, two key bottlenecks are bandwidth limited inter-chiplet links [189] and more expensive inter-kernel reuse caused by additional cache levels being subject to implicit synchronization [105, 239].

To examine these issues we focus on chiplet-based GPUs [92, 179, 198, 221] because GPUs have become the general-purpose accelerator of choice due to their wide availability and ease of programming. Nevertheless, the issues also apply to other accelerators including DSPs [41, 185], NPUs [9], TPUs [97], Apple's accelerators [18], and the Heterogeneous Systems Architecture (HSA) Foundation [87] (discussed further in Section 5.6).

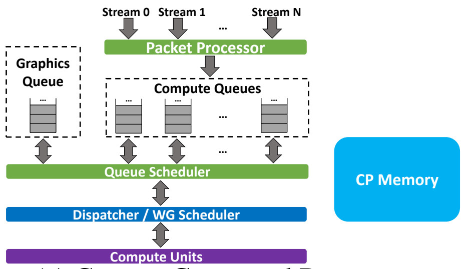
As shown in Figure 5.1b, multi-chiplet GPUs have an additional level of cache. Thus, GPU L2 caches are now shared across CUs within a chiplet, and the L3 cache is a shared LLC across all chiplets. As a result, synchronization operations are even more expensive in multi-chiplet GPUs than monolithic GPUs (discussed further in Section 2.2). Although most GPU applications only have coarse-grained synchronization at kernel boundaries, they still must invalidate all valid data from local caches at kernel launches (similar to an implicit acquire) and write through all dirty data from local caches (similar to an implicit release) when a kernel completes to ensure correctness [67, 85, 136]. In monolithic GPUs, this overhead was relatively small because the L2 cache was shared across all CUs, and GPU L1 caches typically used write-through or write-no-allocate policies. However, in chiplet-based GPUs the L3 is the shared ordering point across chiplets. Thus, the per-chiplet L2 caches must also be invalidated and flushed at kernel boundaries. This increased indirection hurts performance: unlike monolithic GPUs, chiplet-based GPUs cannot exploit inter-kernel locality at the L2.

Prior work has shown that the impact of bandwidth limited inter-chiplet links and the loss of inter-kernel shared L2 reuse is significant: 29%-45% average performance loss [189, 239]. We confirmed these results and found performance is hurt by 54% on average for a subset of the applications we study (Section 5.4). Consequently, efficiently moving data is challenging in chiplet-based heterogeneous systems – a challenge that will become even more acute as systems scale to more chiplets.

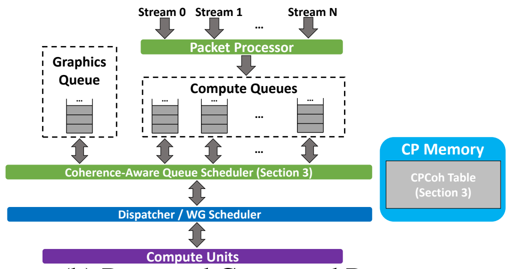
To address this inefficiency we propose CPCoh. CPCoh leverages the key insight that, although current systems do not exploit it (Section 2.2.1), the GPU's

Command Processor (CP) *has a global view of what data is being accessed in each chiplet at a given time*. As shown in Figure 5.1a, modern, monolithic GPUs often utilize a centralized, integrated programmable processor, a Command Processor (CP), to interface between the programmable accelerator and software. Since CPs are programmable, vendors can update them without requiring hardware changes or user-level programmer involvement. However, currently these CPs are largely limited to parsing work contexts and latency-blind scheduling (discussed further in Section 5.2). We propose to redesign the CP to utilize this information, in concert with software information (from the compiler or programmer) to determine the state of data structures in caches. Given this, CPCoh generates the appropriate per-chiplet acquire and release operations at kernel launch time to ensure that data is invalidated and/or flushed right before it will be needed by another chiplet. Effectively, CPCoh converts conservative, per-kernel, GPU-wide implicit acquire and release operations into aggressive, chiplet-specific, on demand acquire and release operations – increasing reuse and reducing the synchronization penalty.

However, modern CPs view the accelerators monolithically, even though accelerators are often distributed across multiple chiplets. Thus, we propose to partition the global, centralized CP's responsibilities between a global CP and local, per-chiplet CPs (Figure 5.1b). The local CPs have access to dynamic, micro-second scale information about their chiplet, which they communicate to the global CP. Likewise, the global CP has a global view of the behavior of all accelerators, including synthesizing the information from the local CPs, and what work is being assigned to the GPU. Additionally, since the global CP has access to the GPU's kernel objects' metadata [87], it knows which data structure(s) each kernel accesses, as well as what chiplet(s) each kernel will be assigned to. The global CP also knows what data structure(s) subsequent kernel(s) will access and which chiplet(s) those kernel(s) will be scheduled on. Thus, the global CP has a complete picture of what data may still be in the chiplet's L1 and L2 caches. CPCoh uses the global CP's complete picture to track, at a data structure granularity, which data structures are being accessed in different kernels and issue



(a) Current Command Processor.



(b) Proposed Command Processor.

Figure 5.2: Current CP versus Proposed CP for chiplet-based systems.

the appropriate per-chiplet synchronization operations.

Prior work has also examined how to improve performance for chiplet-based GPUs. In particular, HMG [189] extends the existing, VI-like GPU coherence protocol from monolithic GPUs (Section 2.2) to be hierarchical. For Multi-GPU (MGPU) systems, Halcone [145] extends timestamp-based monolithic GPU coherence protocols using a timestamp store unit placed in shared physical memory to maintain coherence in MGPU systems. However, our results (Section 5.5) show that HMG's complexity is unnecessary and, in some cases, hurts performance. Shadow tags could be added to reduce the overhead of invalidating valid data [215]. However, there is a sizable overhead to store the shadow tags, the latency to access the shadow tag structure affects the critical path, and flushing per-chiplet dirty data at kernel boundaries would still be expensive. We further discuss related work related to CPCoh in Section 5.7.

Overall, across 22 benchmarks from traditional GPGPU, graph analytics, ML, and HPC workloads, on average CPCoh improves performance by 13% and 14% (17% and 20% for workloads with moderate or higher inter-kernel reuse), energy by 14% and 11%, and network traffic by 19% and 17%, over the baseline 4-chiplet GPU and the state-of-the-art HMG, respectively. Furthermore, for applications without significant reuse CPCoh provides equivalent performance to the baseline. To the best of our knowledge, CPCoh is the first to leverage CP information to mitigate synchronization overheads in multi-chiplet GPUs. Moreover, CPCoh effectively monitors intra- and inter-chiplet behavior without hardware changes.

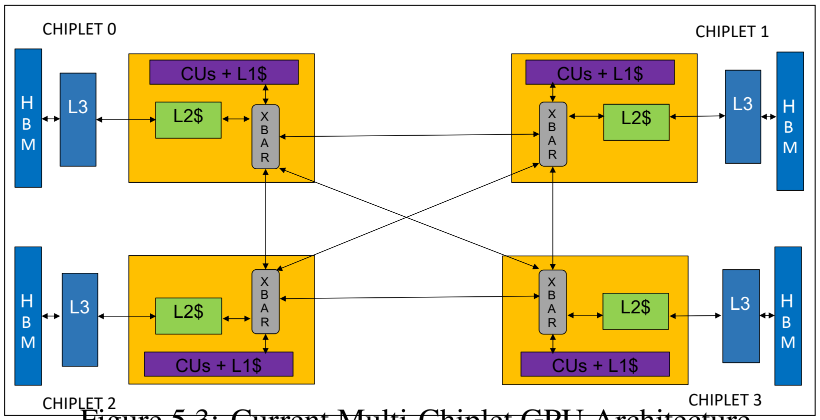


Figure 5.3: Current Multi-Chiplet GPU Architecture

5.2 Background

5.2.1 Multi-Chiplet GPU Architecture

Each GPU chiplet has dedicated CUs, each with a private L1 cache, and a L2 cache shared between its CUs (Figure 5.1b). In some setups each L2's banks are coherent within a single chiplet but incoherent with the rest of the system [189], while in others all banks are coherent with the entire system [198, 221, 238]. Chiplet-based also GPUs introduce an additional level to the memory hierarchy: an LLC which acts as a common shared ordering point across all CUs, although LLC banks are also divided across chiplets, similar partitioning is also done for device's HBM. Moreover, a multi-chiplet GPU's memory subsystem is NUMA and its inter-chiplet links do not provide full aggregated LLC/HBM bandwidth to each chiplet [20] as shown in Figure 5.3. As a result, accesses to another chiplet's memory incur additional latency. Accordingly, inter-chiplet bandwidth is limited [189] and bulk flush and invalidation operations are expensive. Thus, chiplet-based designs have a choice when accessing data that was modified by a thread on another chiplet: a) incur additional latency to access a remote bank of a shared cache [189] or b) perform store releases to flush dirty data from the producer chiplet such that a consumer chiplet can subsequently fetch the data from

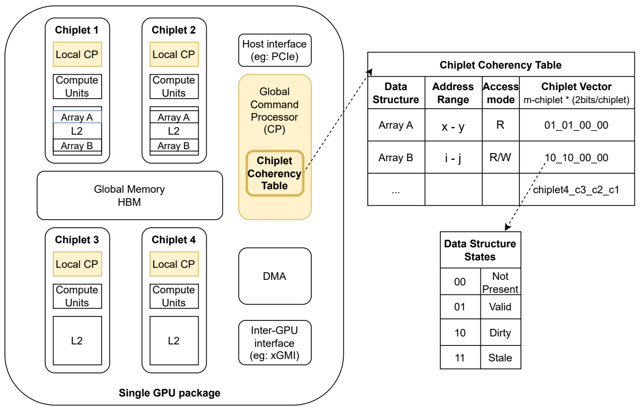


Figure 5.4: Proposed CPCoh architecture

a shared LLC or global memory [198]. Since both approaches incur significant overhead, we investigate alternatives that retain more data in each chiplet's L2 cache.

5.3 Design

5.3.1 Proposed CPCoh Architecture

Figure 5.4 shows CPCoh's overall architecture. To track memory accesses from data structures (e.g., arrays) across chiplets, CPCoh implements a *Chiplet Coherence Table* in the global CP's private memory. This table has 4 fields per row: data structure (e.g., array base address), address range(s) per chiplet, access mode, and a bit vector indicating which chiplets are accessing this data structure and the access state (Section 5.3.2). Each entry is 25 bytes. Since all work dispatched to the GPU goes through the global CP (Section 2.2.1), the global CP has a global view of what data is being accessed in each chiplet at a given time. The global CP gets this information for the kernel argument via the metadata retrieved from kernel packets. Next, when the global CP dispatches the WGs to chiplets, it updates the *Chiplet Coherence Table* for each data structure the kernel accesses. However, the kernel packets do not always provide all of the needed information. For example, some GPU programming languages (OpenCL) provide detailed information about access mode, other languages (CUDA, HIP)

Listing 5.1: Marking array access modes via new API calls.

```
//Square Kernel with Array A (R) as
//input and Array C (R/W) as output
hipSetAccessMode(square, C_d, 'R/W');
hipSetAccessMode(square, A_d, 'R');
hipLaunchKernelGGL(square,..., C_d, A_d, N);
```

do not. Thus, to ensure CPCoh has the necessary information regardless of GPU programming language, we utilize software information (either from the programmer or the compiler) about access mode and address ranges that each chiplet is going to access (Section 5.3.2). Given this information, the global CP has a conservative estimate of what data may still be in the chiplet's L1 and L2 caches at the end of a given kernel. Accordingly, instead of performing implicit acquires and releases on all chiplets at each kernel boundary, CPCoh uses the *Chiplet Coherence Table's* information to generate the appropriate per-chiplet acquire and release operations to invalidate and/or flush data shortly before it will be needed by another chiplet.

5.3.2 Proposed Changes

CPCoh requires several key changes:

Command Processor (CP): Figure 5.2a illustrates some of the details of a simplified CP implementation on modern monolithic GPUs [73]. As discussed in Section 2.2.1, the CP serves as the interface between the software and the GPU hardware. Figure 5.2b shows our redesigned CP, which separates the CPs functionality into two levels: (1) a global CP and (2) a local CP per chiplet. The local CP controls local scheduling decisions (e.g., which CU on the chiplet to schedule a given WG) and passes runtime information back to the global CP. Conversely, the global CP interfaces with the host, dispatches work across chiplets, issues CPCoh's acquires and releases, and stores CPCoh's *Chiplet Coherence Table* inside the CP's private memory.

Labeling Memory Accesses: To identify each global memory data structure, similar to prior work [7, 39, 146, 203, 204] we label each data structure and their

Listing 5.2: Marking both access modes and address ranges for data structures via new API calls. Note that while the programmer does not know which chiplets the kernel will map to the number of chiplets it will use can be configured.

```
//numSchedChip: # chiplets to schedule kernel on
typedef tuple<Addr_t, Addr_t, LogicalchipletID>
rangeChiplet;
//Kernel to be launched on 2 chiplets
//Each chiplets works on half of input & output
vector<rangeChiplet> C_ranges(numSchedChip) =
  {make_tuple(C_d[start], C_d[mid], 0),
   make_tuple(C_d[mid+1], C_d[end], 1)};
vector<rangeChiplet> A_ranges(numSchedChip) =
  {make_tuple(A_d[start] , A_d[mid], 0),
   make_tuple(A_d[mid+1] , A_d[end], 1)};
hipSetAccessModeRange(square, C_d, 'R/W', C_ranges);
hipSetAccessModeRange(square, A_d, 'R', A_ranges);
hipLaunchKernelGGL(square,..., C_d,  A_d, N);
```

access mode: Read-Only (R) or Read/Write (R/W). Although monolithic GPUs generally only need R and R-W labels [118, 203], chiplet-based GPUs must also know **where** these accesses are scheduled. Without scheduling information it is unknown which chiplets have the most up-to-date copy and thus the system must conservatively generate additional acquire-releases.

Although there are several ways for the compiler or programmer to pass this information to the CP, Listing 5.1 shows an example of we propose to modify HIP's open source ROCm [15] to add new API calls for this purpose. Specifically, for each data structure in the kernel, the programmer uses our new `hipSetAccessMode` call to specify if the data structure will be R or R/W in the corresponding kernel. ROCm adds this information to the kernel packet, allowing the global CP's packet processor to access this information. Optionally, programmers can use our new `hipSetAccessModeRange` API call to provide both access mode and address ranges within a data structure that chiplet(s) will be operating on (Listing 5.2). This allows software to specify finer-granularity data structure access patterns. Although the programmer/compiler should be able to determine most GPU access patterns statically, when this is not possible, Listing 5.1's mode-only version should be used. In case the access pattern can be completely determined statically, the responsibility of marking these regions could be added as a feature in the compiler reducing the programming burden. For example, if a kernel accesses different data structures depending on control flow, the programmer must specify all regions that may be accessed by the kernel. Similar to GPU consistency models, this requires that the programmer (or compiler) correctly mark the ranges. If the ranges are incorrect, incorrect outputs may be produced. Moreover, if no information is provided by the programmer/compiler, CPCoh will devolve to baseline behavior and generate conservative acquire and releases.

Tracking Accesses in the CP: Figure 5.4 shows how CPCoh uses a $2n$-bit bit-vector (where n represents the number of chiplets) to track which arrays are accessed, their mode (R, R/W), and by what chiplets. Each table row tracks an array, and the columns specify the virtual address ranges for different chiplets, the access mode, and the $2n$-bit bit-vector tracks what state the array will be in (discussed next) after a particular kernel has completed on the different chiplets. For the GPU applications we studied (Section 5.4), access mode across chiplets for a particular data structure is the same. Thus, we leverage this to generate one chiplet vector per data structure. If this is not true in other applications, CPCoh can be reprogrammed to have a chiplet vector per address range. CPCoh can track up to 8 data structures per kernel; beyond this we coarsen the labels.

Coarsening Data Structure Labels: As in prior work, we find that GPU programs generally access 8 or fewer data structures [120, 222]. However, if this changes in the future, since CPs are programmable the data structure tracking can be increased (at the cost of additional CP memory). Nevertheless, if a kernel accesses more than 8 data structures, then CPCoh will coarsen the information for before adding it to the *Chiplet Coherence Table*. We first search the table to find if any data structures are contiguous in memory. If any are found, CPCoh combines their entries such that both data structures can be indexed with this entry. The combined entry tracks all chiplets any of these data structures were assigned to, and its data structure identifier in the chiplet vector stores the more conservative of the states to ensure correctness. For example, if one data structure is R and the other is R/W, the combined state of the chiplet vector will be R/W. However, if no such structure is found, we then coarsen the data structures closest to one another in memory. Although this approach causes more acquire/releases than required, since the memory between them is not accessed, it ensures correctness.

States: Each Chiplet Coherence Table entry has four possible states, represented by 2 bits per chiplet in the chiplet vector:

- *Not Present (00)*: This state indicates that the data structure does not exist in chiplet `i`'s L2 cache.
- *Valid (01)*: After a given kernel that only reads (access mode R) the data structure, if its data is in the chiplet `i`'s L2 cache, its values will be *Valid*. Thus, if later chiplet `j` wants to write this data structure, chiplet `i`'s copy must be either invalidated or marked as *Stale*.
- *Dirty (10)*: After a given kernel that reads or writes (access mode R/W) a data structure, if its data remains in chiplet `i`'s L2 cache its values may be *Dirty*. Thus, if later chiplet `j` wants to access this data, we must first flush it from chiplet `i`.
- *Stale (11)*: The *Stale* state indicates when the data structure might be in chiplet `i`'s cache, but its values are not the most up-to-date. Thus, the data needs to be invalidated from chiplet `i` before it is safe for it to access them again.

Figure 5.5 shows CPCoh's state diagram, which tracks the state of each data structure in the Chiplet Coherence Table. In the table, each *Chiplet Vector* entry represents the state at a particular chiplet for one data structure.[1] Unlike most

[1]Although we did not observe it, if an application accesses different parts of a data structure in different modes and software cannot statically determine their ranges, CPCoh would create a chiplet vector per range.

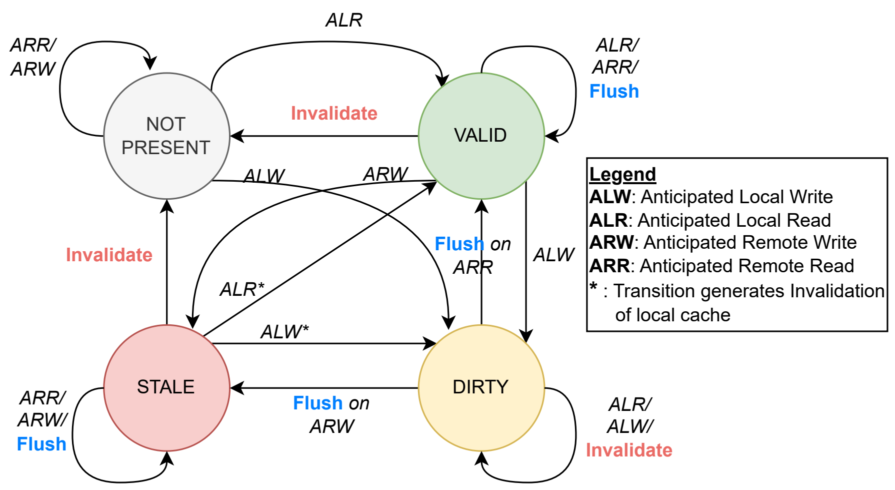

ALR
ARR/ ARW
NOT PRESENT
Invalidate
VALID
ALR/ ARR/ Flush
ALW
ARW
Legend
ALW: Anticipated Local Write
ALR: Anticipated Local Read
ARW: Anticipated Remote Write
ARR: Anticipated Remote Read
* : Transition generates Invalidation of local cache
Invalidate
Flush on ARR
ALW
ALR*
ALW*
STALE
DIRTY
ARR/ ARW/ Flush
Flush on ARW
ALR/ ALW/ Invalidate

Figure 5.5: Internal state tracking mechanism for a data structure in the CPCoh Table for a given chiplet.

coherence protocols, CPCoh does not need transient states since it is not waiting for operations to complete – instead it denotes how the data is being accessed in each chiplet. The transitions show how a Chiplet Coherency Table entries' state changes when different events like Acquires, Releases, Reads, or Writes occur. This state may or may not be the same as the actual state of the cache lines associated with the data structure in the cache since table entries' state is a conservative coarse-grained estimate of the overall state of a data structure in a given chiplet's L2. Moreover, the state transitions in the table occur at kernel launches. For example if a data structure was *Valid* in chiplet 0's cache at the end of kernel 1, and chiplet 0 now receives a kernel that will write the same data structure, the state for this data structure transitions to *Dirty* in the table for chiplet 0, even though the kernel itself has not started. Next, we describe some of Figure 5.5's key state transitions:

Valid $\rightarrow$ ***Stale***: When the address range on a chiplet `i` will be modified by another chiplet `j` in a soon-to-be-launched kernel, CPCoh marks `i`'s entry as *Stale* for this data structure. Thus, chiplet `i`'s data in its L2 is now incoherent. However, CPCoh guarantees it will be invalidated before subsequent uses.

Dirty $\rightarrow$ ***Stale***: When an address range on a chiplet `i` will be written by another chiplet j in a soon-to-be-launched kernel, in order to ensure correctness we must flush the dirty data from chiplet i's L2 cache before chiplet j can access it. Accordingly, CPCoh issues a flush (store release) for the dirty data, then transitions to the *Stale* state. Also, since the baseline coherence protocol keeps the a clean copy of the line in chiplet i's L2 cache, if this address will be subsequently read by chiplet i then CPCoh generates a further invalidate and transitions to Invalid to ensure no stale data is accessed.

Stay in *Valid* on remote accesses: CPCoh elides unnecessary invalidations by allowing caches to retain clean copies if other chiplets are also only reading data from a given address range.

Stay in *Dirty*: Since CPCoh supports software-range based tracking, it retains more data within chiplets. For example, when a chiplet i continues to work on the same data structures across kernels, it is unnecessary to flush the dirty data from i. Instead, CPCoh elides this release operation, increasing reuse.

Lazy Acquire/Release: Monolithic GPUs perform release operations at the end of kernels. Instead CPCoh lazily performs releases as needed based on information the CP receives about subsequent kernel(s). Additionally, with CPCoh we perform the release after the memory acquire associated with the start of a subsequent kernel but before the next kernel issues any memory accesses. In the baseline coherence protocol when a fully dirty line is written back, the cache retains a clean copy of the line and transitions to a shared state. Delaying the release helps CPCoh retain the lines written by the previous kernel. Moreover, this change still produces SC-compliant results for programs with no heterogeneous races (i.e., programs that are SC-for-HRF), since it ensures that no subsequent accesses are performed before any necessary release (flush) and acquire (invalidation) of these operations are completed.

Impact on Memory Consistency Model: We assume the standard SC-for-HRF consistency model. Thus, like current GPUs, any correctly synchronized, simultaneously executing threads either will not be writing the same address or must explicitly synchronize to guarantee correctness. These explicit synchronization operations do not impact the CP's tracking of implicit acquires and releases: even in current programs with explicit synchronization, implicit synchronization operations must still be performed at kernel boundaries.

5.3.3 Functionality

Launching Kernels: Since CPCoh removes the implicit acquire and release operations at kernel boundaries, we adjust how kernels are launched by the global CP. First, when launching a new kernel the global CP must inspect all data structures the kernel accesses, then check its table to determine what state these data structures are in on all chiplets. If any prior kernels accessed the same data structure(s), the global CP must determine if acquire and/or release operations are necessary to ensure that the data accessed by those kernel(s) is flushed or invalidated from the chiplets that accessed it. For any data structures that were accessed by the prior kernels, the global CP will generate synchronization operations for the appropriate chiplet(s) to ensure coherence and consistency are maintained across kernels. These synchronization operations are sent from the global CP to the local CP, which distributes them to its CUs – these requests are then sent to the CU's corresponding L1 and L2 caches to invalidate or flush data. Finally, the local CPs will not launch WGs from the next kernel until the appropriate acknowledgments for acquires and releases are received by the global CP, which then sends a "launch enable" message to the local CPs. Waiting for acknowledgments for acquires/releases and transmitting the final launch signal is on the critical path. Thus, we factor this overhead into our experiments (Section 5.4). Finally, once the acquires and releases are complete, the global CP resets the appropriate bit-vector value to 00 (*Not Present*) to avoid generating further acquires/releases for the same data structure. Once all entries for a given kernel are 00, then we remove that entry.

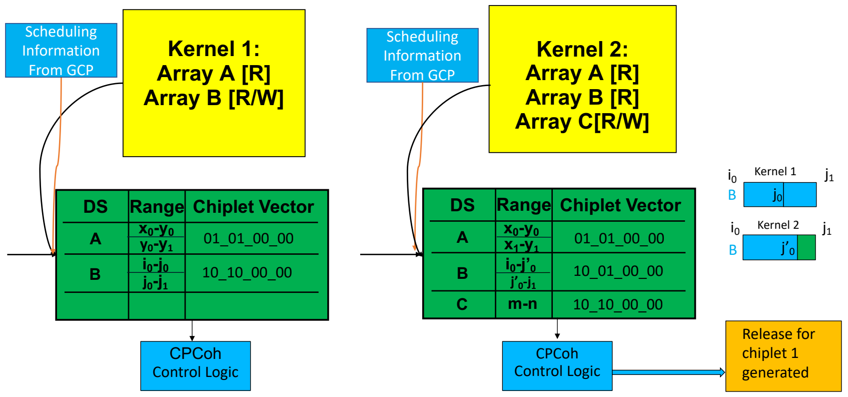
Generating Release Requests: CPCoh sends out a release (flush) request only when a data structure will be accessed in a new kernel and that kernel will be scheduled on either multiple chiplets or a chiplet other than the one in which it is

present in a *Dirty* state. This ensures a chiplet can never read a stale value either from its own cache or the shared LLC. However, if the next kernel accessing this data is scheduled on the same set of chiplets as the previous kernel, CPCoh elides the release if the WGs access the same address ranges in a data structure, on the same chiplets, that they did previously.
Generating Acquire Requests: CPCoh sends out an acquire (invalidation) request only when a data structure will be written to in a new kernel and the new kernel will be scheduled on a chiplet(s) that has the data structure in *Stale*. This ensures that a chiplet does not read stale values, without requiring implicit acquire requests across all chiplets before each kernel.
Removing Entries: When CPCoh generates an acquire or release, CPCoh also updates the chiplet vectors according to the state diagram in Figure 5.5. If the chiplet vector's state is *Not Present* (00) for all chiplets, we remove the entry from the table. Consequently, if an acquire and release are generated for all chiplets, then all entries from the table will be removed.
Indirect Accesses: Some GPU applications use irregular or pointer-based accesses. For irregular accesses, CPCoh's software information identifies how a data structure will be accessed in a given kernel. However, for pointer-based accesses, if this information cannot be determined statically, CPCoh devolves to the baseline and conservatively performs implicit acquires and releases at kernel boundaries.

5.3.4 Putting It All Together

Figure 5.6 shows an example of two kernel launches and how CPcoh generates release/acquire operations for them:

1. At the launch of kernel 1, the CPCoh table is empty. Hence, at this time the data structure information (address range and chiplet vectors) is added to the table with no rel/acq ops generated. Ranges are stored for every chiplet.


Scheduling Information From GCP
Kernel 1: Array A [R] Array B [R/W]
CPCoh Control Logic
Scheduling Information From GCP
Kernel 2: Array A [R] Array B [R] Array C[R/W]
CPCoh Control Logic
Release for chiplet 1 generated
i_0 Kernel 1 j_1
B j_0
i_0 Kernel 2 j_1
B j'_0

DS	Range	Chiplet Vector
A	x_0-y_0 / y_0-y_1	01_01_00_00
B	i_0-j_0 / j_0-j_1	10_10_00_00

DS	Range	Chiplet Vector
A	x_0-y_0 / x_1-y_1	01_01_00_00
B	i_0-j'_0 / j'_0-j_1	10_01_00_00
C	m-n	10_10_00_00

Figure 5.6: CPCoh example operation

2. The two data structures accessed in kernel 1 are A in R mode and B in R/W mode. Since the kernel being scheduled on chiplets 0 and 1, A is in the valid state while B is in the dirty state for these chiplets.

3. Next, kernel 2 is launched. It accesses both array A and B in R mode, while writing to a third array C in R/W mode.

4. The address range for B for chiplet 0 changes and is a superset of its previous range, while the range for chiplet 1 is a subset that decreases and is a subset of the previous range. Thus, a release must be generated for chiplet 1. No coherence action is needed for chiplet 0, since it can continue to read the values it produced while getting the values chiplet 1 produced from main memory. CPCoh also retains reuse from array A in both chiplets.

5.4 Methodology

5.4.1 Baseline GPU Architecture

We model a tightly coupled CPU-GPU architecture with a unified shared memory address space and coherent caches. The system connects all CPU cores and

GPU Feature	**Configuration (Size, Access Latency)**
CPU Parameters	
Num CPUs	1
CPU Clock	4000 MHz
GPU Parameters	
GPU Clock	1801 MHz
Total Num CUs	120,240
Num CUs per Chiplet	60
Num Complexes per Chiplet	1
Num Chiplets	2, 4
Num SIMD units / CU	4
Max WF/SIMD unit	10
Vector Register File Size / CU	256 KB
Scalar Register File Size / CU	12.5 KB
Num Compute Queues	256
LI Instruction Cache / 4 CU	16 KB, 64B line, 8 way
LI Scalar Cache / 4 CU	16 KB, 64B line, 8 way
L1 Data Cache / CU	16 KB, 64B line, 16 way
L1 Latency	140 cycles
L1 Scalar Latency	41 cycles
LDS Size / CU	64 KB
LDS Latency	65 cycles
L2 Cache per chiplet	8 MB, 64B line, 32 way
L2 Latency	269-390 cycles
L2 Write policies	Write-back with write allocate
L3 Size	16 MB, 64B line, 16 way
L3 Latency	330 cycles
Main Memory	16 GB HBM, 4H stacks, 1000 MHz
Inter-chiplet Interconnect BW	768 GB/s
Scheduling policy	Static Kernel Partitioning

Table 5.1: Simulated baseline GPU parameters

GPU CUs via the L3, which also acts as the directory. Figure 5.1 illustrates our baseline GPU in this system, which is similar to prior work [189, 198]. Each GPU chiplet has an L1 cache and LDS per CU, and an L2 cache that is shared across the chiplet's CUs. The private L2 caches are connected via inter-chiplet links using a crossbar [189]. Each address is assigned a home chiplet according to a First Touch page policy (Section 5.4.3.1), and subsequent requests to that address are sent to the home chiplet via an inter-chiplet link. Although it is possible to study CPCoh in the context of Multi-GPU systems, where each GPU has multiple chiplets, we focus on a single GPU because there is significant performance being lost (29%-54%) within a single chiplet-based GPU [105, 239]. However, CPCoh could be extended hierarchically to study Multi-GPU systems.

5.4.2 System Setup

Although CPCoh could be implemented in existing GPUs by re-programming the CP, GPU vendors have not disclosed an API [123, 124, 237]. Thus, we use gem5 [73] to simulate CPCoh, which recent work extended to support multiple chiplets [238]. Although other simulators also support modern GPUs [25, 106, 214], we chose gem5 because it has the most detailed CP model. Specifically, we use gem5 v21.1 [26, 134], which we extended to model local and global CPs, and implemented CPCoh in the global CP. gem5 v21.0 supports ROCm 1.6 [15]. We modified the workloads to label the address ranges and access modes similar to the Section 5.3.2 examples.

Table 5.1 summarizes the common key system parameters, which is based on an AMD Radeon VII GPU. This configuration was validated using microbenchmarks to tune latencies and bandwidths relative to real hardware by Jamieson, et al. [93] and Ramadas, et al. [187]. Our simulated GPU has up to 4 chiplets connected to each other via a fast, high bandwidth interconnect. Similar to prior work, each chiplet has 768 GB/s of bidirectional bandwidth [181]. Finally, like prior work we assume the latency between the local and global CPs is 2 µs [73, 153, 183]. The CPs frequency is 1.5 GHz [159] and the CPs private memory's access latency is 31 cycles [117]. Moreover, since our table uses 1.6 KB, they fit in the CP's private memory and do not change the GPU's area. When measuring energy consumption of the various configurations, we use the same methodology as described in Section 3.

5.4.3 Configurations

We evaluate the following configurations:

Baseline: Baseline implements the multi-chiplet GPU described in Sections 5.2 and 5.4.1.

CPCoh: Our proposed CPCoh approach as discussed in Section 5.3, which uses **Baseline**'s underlying coherence protocol. **HMG (NHCC)**: HMG [189] is a state-of-the-art chiplet-based GPU coherence protocol. Since we study single GPU systems with multiple chiplets, we compare against HMG's NHCC variant. HMG determines the home node (chiplet) for a given physical address using a hash function. The home node always contains each memory location's most up-to-date value.

5.4.3.1 Design Decisions

We also made the following design choices for the different configurations (Section 5.4.3):
Scheduler: We use static kernel-wide WG partitioning, which statically divides the number of WGs in a grid into groups [20, 143]. These groups of WGs are then sent to individual chiplets, where the local dispatcher (in the local CP) round robin schedules them onto individual CUs. Although, LADM proposes more nuanced compile-time static analysis of kernels, we use static kernel-wide partitioning WG scheduling since it is most common and effectively schedules chiplet-based GPUs [105]. Currently CPCoh does not interact with the scheduler. However, CPCoh could also be used to affect the scheduler's decisions to further improve performance as discussed further in Section 5.7.
Page Placement Policy: We use the state-of-the-art First Touch page placement policy for all configurations [20, 189] to isolate the effects of our work as much as possible. However, sometimes this strategy is ineffective [62] and different placement policies can skew performance.
Coherence Protocol: **Baseline** and **CPCoh** use gem5's VIPER GPU coherence protocol with writeback L2 caches, extended for chiplet-based GPUs [73].
Remote Access Allocation Policy: All configurations forward remote requests to the node with the addresses' data. However, **Baseline** and **CPCoh** write-through remote stores and writeback local stores, while **HMG** caches entries in the home node's cache and sends it through to memory.

Application	**Input**
Moderate to high inter-kernel reuse	
Hotspot3D [35]	512 8 20 power_512x8 temp_512x8
Hotspot [35]	512 2 20 temp_512 power_512
Pennant [122]	noh.pnt
FW [37]	512_65536.gr
Color-max [37]	AK.gr
LUD [35]	512.dat
BabelStream [50, 51]	524288
Lulesh [122]	1.0e-2 10
SSSP [37] RNN-GRU [149, 150]	AK.gr BS:4, TS:2, Hidden Layers: 256
Square [16, 28]	524288 1 2 2048 256
RNN-LSTM [149, 150]	BS:4, TS:2, Hidden Layers: 256
Gaussian [35]	256x256
HACC [122]	0.5 0.1 512 0.1 2 N 12 rcb
Backprop [35]	65536
BFS [35]	graph128k.txt
Low inter-kernel reuse	
Pathfinder [35]	200000 100 20
CNN (Conv+Pool+FC) [56]	128x128x3, BS:4
DWT2d [35]	rgb.bmp 4096x4096
SRAD_v2 [35]	2048 2048 0 127 0 127 0.5 2
NW [35]	8192 10
BTree [35]	mil.txt

Table 5.2: Evaluated Benchmarks

5.4.4 Benchmarks

We examine 22 popular traditional GPGPU, graph analytics, HPC and ML applications from gem5-resources [28]. Table 5.2 summarizes these workloads. These workloads represent a wide variety of workloads modern and future GPUs run, and have diverse memory access patterns. We excluded currently unsupported applications in gem5 (e.g., Rodinia [35] ones requiring texture support). For all applications, we configured their input sizes to ensure chiplet-based GPU had reasonable occupancy and memory footprints [88]. Broadly, we grouped the applications into two categories, similar to prior work [84, 88, 107]: (1) applications with moderate to high inter-kernel reuse (2) applications with low to no inter-kernel reuse.

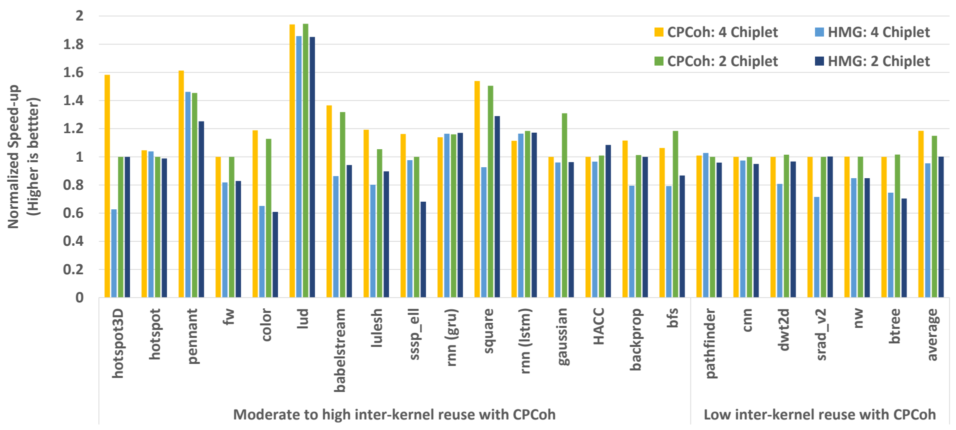


Figure 5.7: Performance of CPCoh and HMG on a 2- and 4-chiplet GPU, normalized to Baseline.

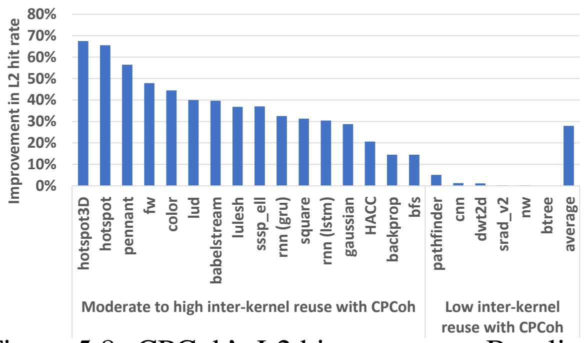


Figure 5.8: CPCoh's L2 hit rate versus Baseline.

5.4.5 Sensitivity Studies

Number of Chiplets: To understand how performance is impacted with number of chiplets, we evaluated all applications and configurations for a system with 2 and 4 chiplets. We use strong scaling – the amount of work is the same, but divide it across the chiplets – because this is representative of an application running on a chiplet-based GPU of a given size.

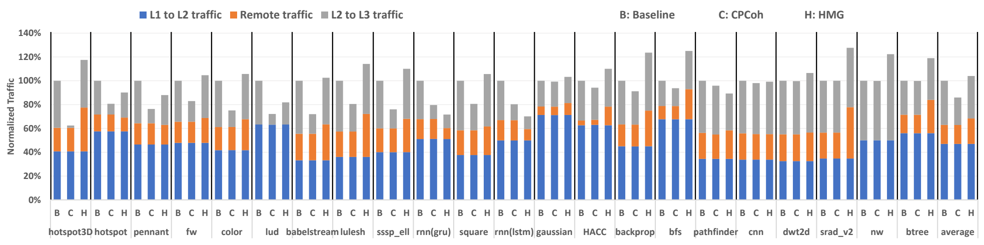


Figure 5.9: Interconnect traffic for Baseline (B), CPCoh (C) and HMG (H) on a 4-chiplet GPU, normalized to Baseline

.

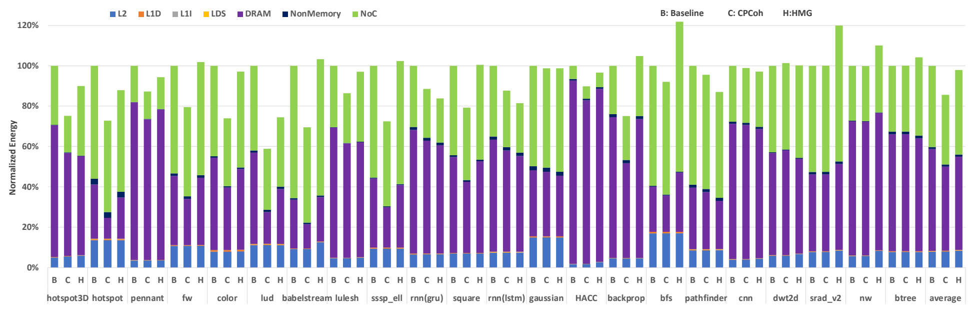


Figure 5.10: Energy expenditure breakdown for Baseline (B), CPCoh (C) and HMG (H) on a 4-chiplet GPU, normalized to Baseline

.

5.5 Results

Figure 5.7 shows the Baseline's, HMG's and CPCoh's normalized performance, across all applications, for 2- and 4-chiplet GPUs. We subdivide this figure into two groups: the group with moderate to high inter-kernel reuse and the group with low inter-kernel reuse. We computed inter-kernel reuse by calculating the reduction in miss rate due to reuse across kernels in a no flush/invalidation caching scheme. Figure 5.8 shows how CPCoh's L2 cache hit rates compare to Baseline. Finally, Figure 5.9 shows the normalized network traffic for a 4-chiplet GPU, measured in flits and divided into multiple components: L1-to-L2, L2-to-L3, and remote. Overall, CPCoh improves performance, energy consumption, and network traffic over both the Baseline and HMG, for both 2-and 4-chiplet GPUs. For example, in 4-chiplet GPUs on average (arithmetic mean) CPCoh outperforms Baseline by 17% and HMG by 20% for applications with moderate to high inter-kernel reuse. Moreover, for applications with limited inter-kernel reuse, CPCoh and HMG provide similar performance to the Baseline. On average CPCoh also reduces energy (14%, 11%) and network traffic (14%, 17%) relative to Baseline and HMG. Thus, CPCoh does not hurt performance for applications without reuse, and improves reuse, network traffic, and hit rate (by 29% on average), including over the state-of-the-art HMG, for applications that have inter-kernel reuse.

5.5.1 4-Chiplet GPUs: CPCoh vs Baseline

Moderate-to-High Inter-Kernel Reuse: Figure 5.7 shows that CPCoh generally improves performance for benchmarks with larger ($> 15\%$) inter-kernel reuse in 4-chiplet GPUs. Since these applications have significant inter-kernel reuse, they benefit from CPCoh preserving their inter-kernel locality. However, the results vary significantly based on the application's access patterns. In particular, applications (e.g., BabelStream and Square) with iterative GPU kernels and uniform access patterns can easily divide WGs into chunks can be scheduled on independent chiplets with limited remote accesses and their working sets fit into the chiplet's aggregate L2 capacity. As a result, CPCoh improves their performance by 31% on average over *Baseline*. Likewise, the GRU and LSTM RNN's reuse input matrix weights across GEMM kernels and have producer-consumer style reuse across kernels. CPCoh preserves this reuse, improving their performance by 11% on average. Finally, Hotspot3D performs a memory bound 3D stencil operation; inter-kernel L2 reuse for its read-only arrays helps CPCoh outperform Baseline by 37%.

More irregular applications like Color, SSSP, and BFS have many read-only memory accesses [89]. Thus, avoiding unnecessary acquires improves their inter-kernel reuse and performance: 16% for Color, 14% for SSSP, and 6% for BFS, which has less potential inter-kernel reuse. Similarly, Pennant and Lulesh use indirect addressing or have unstructured data structures causing irregular memory access patterns [122]. However, since these accesses are limited to a subset of addresses that fit into the aggregate L2 capacity, CPCoh improves their performance by 38% and 16%, respectively.

GPU applications also frequently structure access patterns into three phases, each separated by WG synchronization. First, the data is loaded into the LDS, next it performs compute operations on the data, and finally the data is written back to global memory. Here inter-kernel cache locality only helps for the first (read into shared memory) and the last (write to global memory) phases. Thus, the benefits for these applications depend on ratio between the three phases: compute-bound applications see little benefit, whereas memory-bound applications with few ALU operations benefit more: e.g., Backprop and LUD see 10% and 48% benefit respectively.

For other applications inter-kernel reuse has a weak correlation with speed-up. Hotspot and CNN are compute-bound with on-chip memory bandwidth being sufficient to keep the CUs busy – hence CPCoh's speedup for them is low. Hotspot is bottlenecked by compute stalls. Thus, loading the LDS faster via more L2 hits does little to alleviate this problem. Moreover, sometimes (e.g., FW, Gaussian, HACC) there is sufficient memory-level parallelism to hide the L2 cache misses caused by implicit kernel boundary synchronization. Thus, although CPCoh improves their L2 inter-kernel reuse, other accesses must go to main memory. Consequently, hitting more in the L2 cache does not significantly improve their performance.

Low-to-No Inter-Kernel Reuse: Unsurprisingly, CPCoh and Baseline perform similarly for workloads (e.g., Pathfinder, DWT2D, BTree and NW) with limited or no inter-kernel reuse. Since these applications do not have significant reuse, eliding acquires and releases does not significantly affect them.

5.5.2 4 Chiplet System: CPCoh vs HMG

Moderate-to-High Inter-Kernel Reuse: Figure 5.7 also compares CPCoh's and HMG's performance. For applications with little to no remote accesses (e.g., BabelStream, Square), CPCoh elides all flushes and invalidations except the final ones. However, since HMG uses write-through L2 caches, it always sends writes through to memory, generating much more L2-L3 traffic than CPCoh. This significantly slows down HMG compared to CPCoh: 37% for BabelStream and

40% for Square. Compared to Baseline, HMG caches remote traffic, evicting some local data from the cache and generating invalidation traffic. Consequently, HMG performs slightly worse than Baseline, which cannot provide inter-kernel reuse.

Likewise, graph analytics workloads like Color, SSSP, and FW have input-dependent memory accesses which cause many remote accesses, since the first-touch page policy is subpar when the access pattern is irregular [62]. HMG caches all remote accesses at their home node. Thus when the data locality in remote accesses is low, it generates considerable invalidation traffic, and reduces space for that particular chiplet's local reads and writes, exacerbating HMG's ability to tap into reuse for local data. Across the graph workloads, CPCoh is 26% faster than HMG on average. Baseline also outperforms HMG sometimes for these workloads. Even though Baseline cannot provide inter-kernel reuse (unlike HMG), it better leverages intra-kernel reuse in the L2 cache for local reads and writes due to HMG caching data in the home node. Lulesh's irregular access patterns cause considerable HMG invalidation traffic, enabling CPCoh to outperform HMG by 33%.

Pennant (38%) and LUD (48%) exhibit significant inter-kernel reuse. However, CPCoh and HMG achieve similar performance for them, since both capture inter-kernel reuse and their invalidation traffic is low in HMG. CPCoh and HMG also perform similarly for compute-bound benchmarks (CNNs and Hotspot) and benchmarks with limited inter-kernel reuse (Pathfinder, DWT2D and HACC). The GRU and LSTM RNNs have good locality in remote reads from the shared input weights and intermediate results, enabling HMG to perform slightly better (3%) than CPCoh because CPCoh does not cache remote reads. HMG also outperforms Baseline because it improves inter-kernel locality – like CPCoh, Baseline does not provide intra-kernel locality for remote reads like HMG. Figure 5.9 shows that for most applications both CPCoh and HMG reduce L2-L3 traffic versus Baseline. Thus, to varying degrees both CPCoh and HMG preserve inter-kernel reuse. However, CPCoh reduces L2-L3 traffic by 37% more than HMG because CPCoh does not cache remote data, which often has low locality. CPCoh also leverages producer-consumer information without maintaining the directory's sharer list like HMG. HMG also generates 23% more remote traffic than CPCoh, due to invalidations from tying four cache lines to one directory entry, which causes unnecessary invalidations and evicting local data on remote accesses. Overall CPCoh reduces network traffic by 17% over HMG.

Low-to-No Inter-Kernel Reuse: For applications with limited inter-kernel reuse (e.g., BTree, SRAD_v2), HMG suffers from many directory evictions because it binds four cache lines to one directory entry. These evictions also generate considerable remote invalidation traffic, hurting its overall performance. Consequently, Baseline outperforms HMG for these workloads by 15% on average, while Baseline and CPCoh perform similarly. Although this was initially surprising, recent work corroborated that HMG suffers in these situations [62]. Thus, while HMG sometimes outperforms CPCoh, in aggregate CPCoh outperforms HMG by intelligently eliding synchronization operations. HMG fares much better against Baseline by leveraging inter-kernel reuse. However HMG's write policy, remote read caching, and binding 4 cache lines to one directory, sometimes hurt its performance compared to Baseline.

5.5.3 Number of chiplets

Figure 5.7 also compares CPCoh and HMG for 2-chiplet GPUs. Generally, the trends for 4-chiplet GPU also hold in 2-chiplet GPUs. However, there are some exceptions. For example, CPCoh does not improve Backprop's, Hotspot3D's, and SSSP's performance in 2-chiplet systems since its aggregate L2 cache capacity is insufficient for their larger memory footprint. HMG also fairs considerably better for benchmarks like SRAD_v2 and DWT2D – since there are fewer places for remote requests to go, there is less invalidation traffic and fewer directory invalidations. HMG also improves performance for benchmarks which suffered from low locality in remote reads, since fewer remote cache lines are cached since there are fewer remote nodes. Thus, reuse from local cache lines increases. Consequently, CPCoh's overall improvement over HMG decreases by 6% relative to the 4-chiplet GPU. However, given that the number of chiplets per GPU are likely to increase in the future, and that CPCoh's gains increase as chiplets scale, CPCoh provides better scalability.

5.6 Discussion

Chiplet-based GPU versus Multi-GPU systems: To the best of our knowledge, CPCoh is the first to leverage CP information to mitigate synchronization overheads in chiplet-based GPUs. As discussed in Section 5.4.1, it is also possible to study CPCoh in multi-GPU systems, where each GPU has multiple chiplets. However, we focused on a single GPU composed of multiple chiplets because there are opportunities for improved reuse within a single GPU (Section 5.5). Our results demonstrate that CPCoh improving multi-chiplet GPU performance can also potentially help multi-GPU systems.

Fine-grained Hardware Range Based Flush: Although CPCoh uses range-based tracking to determine which addresses to flush/invalidate, it must still send out flushes or invalidates for the entire cache even if it was only necessary for some of the addresses. To avoid flushing or invalidating the entire cache would require additional address translation support: since CPCoh tracks accesses via software hints about kernel arguments, it must track virtual addresses, while GPU L2 caches are physically addressed. Thus to perform hardware range-based flushes, CPCoh would need to translate the virtual address ranges to physical addresses. Since most GPU vendors use page-aligned array allocations, the flush/invalidation of these address ranges can be broken down into page-wise requests. These requests can then be sent to the core, where they would be translated into the corresponding physical pages, and bypass the L1 cache (which must always be flushed at synchronization points) to go to the L2 to perform more targeted flushes there. This technique may require multiple cache walks depending on the address range's size. However, if the writeback time is greater than the cache walk time, then critical path latency will not be affected. In return, this additional hardware support can further improve inter-kernel reuse and performance.

Comparison to Directories: Although CPCoh's tracking mechanism bears some similarity to directory-style coherence protocols, they serve different purposes and are complementary. Directory protocols are primarily concerned with fine-grained, cache line granularity coherence ordering from request to request. In comparison, CPCoh's tracking mechanism is focused on coarse-grained tracking of larger (often array-based) data structures. Additionally, CPCoh does not enforce ordering on a request-by-request basis. Instead CPCoh enforces ordering only at implicit synchronization points. Thus, CPCoh complements existing directory protocols, focusing on when to perform and elide synchronization operations.

Multi-Stream Workloads: Although our workloads do not use multiple streams, CPCoh will also be valuable for multi-stream workloads. In multi-stream workloads independent kernels from different streams run simultaneously. Thus, data movement and locality become even more challenging since concurrent kernels may cause contention for shared caching resources. Accordingly, CPCoh's ability to track data placement and elide unnecessary implicit acquire and release can further improve performance. Moreover, CPCoh's information could influence the WG scheduler's scheduling decisions. For example, CPCoh's information could increase locality by scheduling WGs accessing the same data on the same chiplet.

Kernel Fusion: GPU software frequently use optimizations such as kernel fusion [54, 59, 64, 108, 210, 224, 246] to combine operations into a single kernel to avoid reduce data movement and redundant global memory accesses. However, kernel fusion can also increase register and LDS pressure and may limit parallelism. Thus, while kernel fusion may improve data reuse, for larger applications it may not scale and the overall application will still require implicit synchronization. Moreover, kernel fusion can be difficult when the dependencies between kernels are difficult to establish. CPCoh overcomes this problem by only needing the programmer to define access modes, and then tracking the dependencies itself in the Chiplet Coherency Table.

Other Coherence Protocols: We focused on applying CPCoh to the existing

Feature	HMG[189]	Spandex[7, 203]	hLRC[6]	Halcone[145]	SW DSM[95, 245]	HW DSM[132, 230]	CPCoh
No coherence protocol changes	X	X	X	X	X	X	✓
No L2 cache structure changes	X	X	X	X	✓	X	✓
Reduces Kernel Boundary synchronization overhead	✓	✓	✓	✓	✓	✓	✓
Avoids remote coherence traffic	X	X	X	✓	X	X	✓
Designed for chiplet-based systems	✓	X	X	X	X	X	✓
Access to scheduling information to reduce overhead	X	X	X	X	X	X	✓

Table 5.3: Comparing CPCoh to prior work

GPU coherence protocol and consistency model. However, since CPCoh targets kernel boundary overheads, it can also be applied in conjunction with other GPU coherence protocols. For example, CPCoh is compatible with HMG and Halcone [145], and monolithic GPU coherence protocols like hLRC [6], hUVM [118], DeNovo [203], or Spandex [7]. We expect that CPCoh's benefits will be strongly correlated with the cost of implicit synchronization at kernel boundaries in the underlying coherence protocol.

Other Accelerators: Kernels are a GPU-specific way of partitioning work. Other accelerators partition work into different types of phases and granularities. Nevertheless, because CPCoh targets phase (kernel) boundary synchronization, it can be applied to other accelerators using the same approach we use for GPUs. However, since accelerators also access memory very differently and often prefer different levels of integration [7], this may require using techniques like CAPI [213], CHI [19], or Spandex [7] to present a flexible coherence interface for accelerators. Importantly, many other accelerators [9, 18, 41, 87, 97, 185] also use embedded microprocessors (like CPs) to dispatch work to accelerators.

5.7 Related Work

Table 5.3 compares CPCoh to prior work across several important metrics. This prior work significantly advanced the field, but does not target implicit synchronization like CPCoh, so they cannot provide all of the same benefits.

Chiplet-based GPU Coherence Protocols: Halcone [145] and HMG [189] also designed chiplet-based GPU coherence protocols. HMG extends existing monolithic GPU coherence protocols (Section 2.2) to be hierarchical. However, as shown in Section 5.5, CPCoh outperforms HMG. Halcone [145] extends timestamp-based monolithic GPU coherence protocols for multi-GPU systems by adding hierarchical timestamps. However, it is unclear how Halcone will work in a multi-chiplet GPU and it assumes low bandwidth links between GPUs, which is less important in a multi-chiplet GPU. Finally, although eliding implicit kernel boundary synchronization bears some similarity to prior multi-core CPU work like BulkSC [33] and DeNovo [39], neither examines implicit synchronization. Thus, CPCoh provides benefits over the state-of-the-art and is the first to target kernel boundary synchronization overheads in chiplet-based GPUs and redesign the CP to track coherence information. Furthermore, since CPCoh targets kernel boundary overheads, it is compatible with many GPU coherence protocols (Section 3.5).

Reducing NUMA Penalty in Chiplet-based GPUs: CARVE improves NUMA GPU performance by extending the GPU cache hierarchies capacity [239], while LADM uses static analysis to improve intra-kernel locality via better scheduling [105]. Both CARVE and LADM corroborate that implicit synchronization at kernel boundaries ruins the inter-kernel locality, hurting performance. Other work optimized WG scheduling and/or placement algorithms [20, 111]. Intelligent schedulers like these could be used in conjunction with CPCoh, which has detailed information about where data is being accessed and tight coupling with the WG scheduler. However, intelligent schedulers do not target implicit synchronization. AMD propose an architecture where the LLC (the L3) and HBM (High Bandwidth Memory) are logically shared across the chiplets, but are physically sub-divided across them – each chiplet has a portion of the L3 and the HBM [198]. This architecture makes CPCoh even more attractive: unlike HMG, CPCoh will not incur remote latencies for non-local data. TD-NUCA tracks and optimizes the placement of a block across the shared LLC banks [30]. Although this allows TD-NUCA to mitigate non-uniform latency effects, it does not preserve inter-kernel reuse within private caches like CPCoh. To preserve reuse in a chiplet-based GPU, run-time scheduling information is required, which CPCoh leverages via the CP.

Extending the CP: Prior work added networking [123, 124] or priority-based queue scheduling [183] support to CPs. However, our work solves different problems and is concerned with leveraging information available in the CP to mitigate coherence overheads in chiplet-based GPUs.

Coarse-grained Tracking in Distributed Shared Memory: *Software and Hybrid DSM's*: CPCoh also shares some similarities with software and hybrid Distributed Shared Memory (DSM), which also perform coarse-grained memory tracking. In general, these approaches provide coherence in software, often at a page granularity [116, 245]. Follow-on work used manual fine-tuning to reach the performance levels of cache-coherent hardware counterparts [95]. Other DSMs use compilers to eliminate unnecessary barrier calls [76]. However, because they are at the software (or hybrid) level, they require additional support (e.g., duplicate page copies). Moreover, to accurately track states for data structures (Section 5.3.2), run-time scheduling information is required – which is unavailable at the compiler/software level. This information could be passed at run-time to the host software by extending ROCm. However, there would be a significant latency penalty waiting for the host software, which will hurt performance. In comparison, CPCoh leverages lower level access and scheduling information available in the CP to synchronize only when necessary, at different granularities, without requiring additional copies, and without host-side software latency overheads. Although CPCoh could also be enhanced by static compiler analysis [76, 105], it is not always possible.

Hardware DSM's: Hardware-based DSMs monitor the coherence status of large, aligned memory regions in hardware to snoop external requests and provide region snoop responses [132]. However, this requires a warm-up phase and can lead to false sharing if the regions are not appropriately sized. This is unnecessary in CPCoh, which leverages scheduling, access mode, and data range information to make coherence decisions before a kernel starts. Other proposals such as DirSW shift some of the coherence burden to software to identify independent regions [230]. However, this is difficult in GPUs since many kernels use complex data indexing mechanism leveraging multi-dimensional thread grid structures. Most GPUs also lack OS support, which DirSW relies on, making it difficult to adopt DirSW in GPUs.

5.8 Conclusion

Modern systems are increasingly turning towards chiplet-based designs. However, the additional level of hierarchy in chiplet-based heterogeneous systems clashes with how accelerators like GPUs are designed: monolithic GPUs assume relatively flat memory hierarchies with local, per CU L1 caches and shared L2 caches, and software-driven coherence that flushes L1 caches at the end of kernels and invalidates them at the beginning of kernels. Although this approach still works in chiplet-based systems, the L2 cache is no longer shared across all CUs. Thus, both the L1 and L2 must be flushed (releases) and invalidated (acquires) at kernel boundaries – hurting inter-kernel reuse. To overcome this we propose CPCoh, which redesigns GPU CPs to track which chiplets access specific memory addresses. This allows CPCoh to intelligently elide implicit acquires and releases – only performing them when needed and on the appropriate chiplets. Overall, on average CPCoh improves performance by 13% and 19%, energy by 14% and 11%, and network traffic by 14% and 17% over current approaches, respectively, without requiring hardware changes.

6 CONCLUSION

An increasingly wide spectrum of applications is utilizing modern GPUs to improve their performance. However, the characteristics of these applications differ from traditional GPGPU applications. In particular, these new applications frequently perform fine-grained global synchronization, whereas traditional GPGPU workloads synchronize infrequently. Thus, this presents new challenges for efficiently performing global synchronization on GPUs. This thesis presents work that creates more efficient ways to perform global synchronization in GPUs. Though the core GPU principle of massive parallelism will continue to drive their widescale use, modern GPU applications do much more than parallel computing. Global synchronization forms a critical component of execution time for these applications. Moreover, as GPU vendors continue to scale compute power while battling technology constraints, deeper memory hierarchies will cause increased NUMA effect. Consequently, synchronization will become an even bigger component of total application time. This synchronization can be either explicit through atomics or software primitives like locks, mutexes, semaphores, and barriers, or implicit, taking place at kernel boundaries. We target the bottlenecks that arise in synchronization from scaling and provide solutions that benefit performance, energy, and network traffic.

First, to solve the bottleneck of inefficient global synchronization using device-scoped atomics. As discussed in **Chapter 3** modern applications in graph analytics and ML training use commutative relaxed atomics to update globally shared variables. We leverage recent work to identify these commutative atomics in software. Then in the hardware, we introduce a small, per-SM buffer (LAB) that combines commutative atomics locally and utilizes reconfigurability to avoid hurting applications that do not use commutative atomics. Next, in **Chapter 4** we looked at the problem of GPU software synchronization primitives not scaling well, and proposed more efficient, scalable GPU barriers and semaphores. This demonstrates how building scalable synchronization primitives requires careful consideration of the GPU memory hierarchy, coherence protocol, consistency models, and threading model. Lastly, in **Chapter 5** we show how new chiplet-based GPUs increase the penalty of implicit synchronization at kernel boundaries due to loss of inter-kernel reuse from previously shared, now private L2 caches. To overcome this performance penalty we propose CPCoh, which redesigns GPU CPs to track which chiplets access specific memory addresses. This allows CPCoh to intelligently elide implicit acquires and releases, only performing them when needed and on the appropriate chiplet. The solutions we presented in this thesis outperform state-of-the-art schemes. We also explain the future direction of this work in the respective chapters, outlining how these schemes can be scaled as GPUs move to more hierarchical designs.

6.1 Reflections

In this section, I share some opinions on GPU synchronization. These opinions are based on five years of academic research on GPUs, and multiple internships at Microsoft, AMD, and Meta. I base these observations solely on my experiences and conversations I have had with fellow researchers in academia and industry. Some of these can be considered conjectures about the future, these observations are meant to provide a view of the world of GPU synchronization from my eyes. I reserve the right to change my mind about them later.

6.1.1 GPU Synchronization Schemes Should Be Aligned With GPU Architecture

Much of the hidden performance fruit for GPU synchronization hinges on best-leveraging finer nuances of GPUs such as the GPU memory hierarchy, coherence protocol, consistency models, and threading model. For example, GPU synchronization has a huge interplay with memory scopes: as GPUs follow a consistency-directed coherence protocol, synchronization penalties are associated with the scope of synchronization. The ability to leverage these scopes as shown in the proposed G-SRB in Chapter 4, is key to performance and energy benefits. The success of future schemes might be dependent on appropriate help from software. Considering that GPU architecture is throughput-oriented and many GPU applications do not have as much data sharing as CPUs, GPU-scoped memory models can be improved to enable programmers to take a more aggressive approach while specifying these scopes. This will increase the overall efficiency of synchronization schemes that leverage different scopes for more performance gains.

6.1.2 Scalability Is Critical To Adoption For GPU Synchronization Schemes

GPU vendors have been changing the GPU memory hierarchy over generations to better adapt to the needs of modern applications. In some cases introducing an additional level of memory hierarchy increases the cost of global synchronization substantially as shown in Chapter 5. Our solutions are easily extensible and adaptable to the changes in the underlying architecture because the core principles of their design are not tied to specific architectural properties. For example, to apply LAB (Chapter 3) to Multi-GPU or chiplet-based architecture we suggest a hierarchical design. Apart from all SMs having their own LAB, each chiplet would have another level of LAB as part of the L2 cache to batch updates from all LABs on different SMs on the same chiplet.

6.1.3 GPU Synchronization Optimizations Should Not Hurt Data Parallel Streaming Applications

GPUs were traditionally designed for streaming data-parallel applications with little to no fine-grained synchronization. Even though GPUs have now found use in applications that have challenged these semantics and display data reuse with the need for frequent global synchronization. Traditional GPU applications and some modern applications still bank on GPU's massive parallelism and consequent high throughput for performance. Any synchronization schemes that impact the performance of these applications negatively are not a good fit as shown in Section 3.4.5.

REFERENCES

[1] Aamodt, Tor M., Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. *General-Purpose Graphics Processor Architectures*. Morgan and Claypool, Synthesis Lectures on Computer Architecture.

[2] Abdolrashidi, AmirAli, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. 2017. Wireframe: Supporting Data-Dependent Parallelism through Dependency Graph Execution in GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 600–611. MICRO.

[3] Adriaens, J. T., K. Compton, N. S. Kim, and M. J. Schulte. 2012. The Case for GPGPU Spatial Multitasking. In *IEEE International Symposium on High-Performance Computer Architecture*, 1–12. HPCA.

[4] Ahmad, M., and O. Khan. 2016. GPU Concurrency Choices in Graph Analytics. In *IEEE International Symposium on Workload Characterization*, 1–10.

[5] Ahn, Junwhan, Sungjoo Yoo, and Kiyoung Choi. 2016. AIM: Energy-Efficient Aggregation Inside the Memory Hierarchy. *ACM TACO* 13(4).

[6] Alsop, Johnathan, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy Release Consistency for GPUs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, 26:1–26:13. MICRO.

[7] Alsop, Johnathan, Matthew D. Sinclair, and Sarita V. Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 261–274. ISCA, Piscataway, NJ, USA: IEEE Press.

[8] Alvarez, L., L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero. 2015. Coherence Protocol for

Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures. In *ISCA*, 720–732.

[9] Amant, Renée St., Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-Purpose Code Acceleration with Limited-Precision Analog Computation. In *ACM/IEEE 41st International Symposium on Computer Architecture*, 505–516. ISCA.

[10] AMD. 2012. AMD's Asynchronous Shaders White Paper.

[11] ———. 2016. AMD Graphics Core Next (GCN) Architecture, Generation 3. `http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf`.

[12] ———. 2016. OpenCL Programming Guide. `http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf`.

[13] ———. 2018. HIP: Heterogeneous-computing Interface for Portability. `https://github.com/ROCm-Developer-Tools/HIP/`.

[14] ———. 2018. HIP: Heterogeneous-computing Interface for Portability.

[15] ———. 2018. ROCm: Open Platform For Development, Discovery and Education around GPU Computing. `https://gpuopen.com/compute-product/rocm/`.

[16] ———. 2023. HIP-Examples. `https://github.com/ROCm-Developer-Tools/HIP-Examples`.

[17] Anderson, T. E. 1990. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 1(1): 6–16.

[18] Apple. 2021. Dispatch. `https://developer.apple.com/documentation/dispatch`.

[19] ARM. 2018. AMBA 5 CHI Architecture Specification Architecture Specification. `https://developer.arm.com/documentation/ihi0050/c/`.

[20] Arunkumar, Akhil, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the 44th annual international symposium on computer architecture*, 320–332. ISCA '17, New York, NY, USA: ACM.

[21] Arunkumar, Akhil, Evgeny Bolotin, David Nellans, and Carole-Jean Wu. 2019. Understanding the Future of Energy Efficiency in Multi-Module GPUs. In *25th IEEE International Symposium on High Performance Computer Architecture*, 519–532. HPCA.

[22] Bader, David A., Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. 2018. *Benchmarking for graph clustering and partitioning*, 161–171. Springer New York.

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

[24] Balaji, Vignesh, Dhruva Tirumala, and Brandon Lucia. 2017. Flexible Support for Fast Parallel Commutative Updates. `1709.09491`.

[25] Bao, Yuhui, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L. Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2023. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *Proceedings of the International Conference on Parallel Architectures*

and Compilation Techniques, 333–345. PACT ’22, New York, NY, USA: Association for Computing Machinery.

[26] Binkert, Nathan, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39(2):1–7.

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

[28] Bruce, Bobby R., Ayaz Akram, Hoa Nguyen, Kyle Roarty, Mahyar Samani, Marjan Fariborz, Trivikram Reddy, Matthew D. Sinclair, and Jason Lowe-Power. 2021. Enabling Reproducible and Agile Full-System Simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS.

[29] Burtscher, M., R. Nasre, and K. Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *IEEE International Symposium on Workload Characterization*, 141–151. IISWC.

[30] Caheny, Paul, Lluc Alvarez, Marc Casas, and Miquel Moreto. 2022. TD-NUCA: Runtime Driven Management of NUCA Caches in Task Dataflow Programming Models. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15. SC.

[31] Cederman, Daniel, and Philippas Tsigas. 2008. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 57–64.

[32] Center, NVIDIA Documentation. 2022. NVIDIA Cuda samples. `https://docs.nvidia.com/cuda/cuda-samples/index.html`.

[33] Ceze, Luis, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 278–289. ISCA '07, New York, NY, USA: Association for Computing Machinery.

[34] Challapalle, Nagadastagiri, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. Gaas-x: Graph analytics accelerator supporting sparse data representation using crossbar architectures. In *2020 acm/ieee 47th annual international symposium on computer architecture (isca)*, 433–445. IEEE.

[35] Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization*, 44–54. IISWC.

[36] Che, S., J. W. Sheaffer, M. Boyer, L. G. Szafaryn, Liang Wang, and K. Skadron. 2010. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *IEEE International Symposium on Workload Characterization*, 1–11. IISWC.

[37] Che, Shuai, Bradford M. Beckmann, Stephen K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding Irregular GPGPU Graph Applications. In *IEEE International Symposium on Workload Characterization*, 185–195. IISWC.

[38] Chen, Fan. 2021. Puffin: an efficient dnn training accelerator for direct feedback alignment in fefet. In *2021 ieee/acm international symposium on low power electronics and design (islped)*, 1–6. IEEE.

[39] Choi, Byn, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 155–166. PACT '11, USA: IEEE Computer Society.

[40] Chou, Y. H., C. Ng, S. Cattell, J. Intan, M. D. Sinclair, J. Devietti, T. G. Rogers, and T. M. Aamodt. 2020. Deterministic Atomic Buffering. In *MICRO*, 981–995.

[41] Codrescu, Lucian, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. 2014. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro* 34(2):34–43.

[42] Collange, Sylvain, David Defour, Stef Graillat, and Roman Iakymchuk. 2014. Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures. Tech. Rep., INRIA - Centre de recherche Rennes - Bretagne Atlantique.

[43] Cook, Henry, Krste Asanovic, and David A. Patterson. 2009. Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments. Tech. Rep., Electrical Engineering and Computer Sciences University of California at Berkeley.

[44] Dally, William J. 2018. Hardware for Deep Learning. SysML Keynote.

[45] Dalmia, Preyesh, Rohan Mahapatra, Jeremy Intan, Dan Negrut, and **M. D. Sinclair** Sinclair. 2023. Improving the Scalability of GPU Synchronization Primitives. *IEEE Transactions on Parallel and Distributed Systems* 34(1): 275–290.

[46] Dalmia, Preyesh, Rohan Mahapatra, and Matthew D. Sinclair. 2022. Only Buffer When You Need To: Reducing On-chip GPU Traffic with Reconfigurable Local Atomic Buffers. In *IEEE International Symposium on High-Performance Computer Architecture*, 676–691. HPCA.

[47] Dalmia, Preyesh, Rajesh Shashi Kumar, and **M. D. Sinclair**. 2022. Utilizing Embedded Microprocessors to Intelligently Elide Implicit Synchronization at Phase Boundaries in Multi-Chiplet Accelerators. In *US Patent App. submission*. US Patent App. submission, US Patent App. in submission.

[48] ———. 2023. CPCoh: Efficient Multi-Chiplet GPU Coherence. In *submission*.

[49] De Sa, Christopher, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 561–574. ISCA, New York, NY, USA: ACM.

[50] Deakin, Tom, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models. In *High performance computing*, ed. Michela Taufer, Bernd Mohr, and Julian M. Kunkel, 489–507. Cham: Springer International Publishing.

[51] ———. 2018. Evaluating Attainable Memory Bandwidth of Parallel Programming Models via BabelStream. *Int. J. Comput. Sci. Eng.* 17(3): 247–262.

[52] Defour, D., and S. Collange. 2015. Reproducible floating-point atomic addition in data-parallel environment. In *FedCSIS*, 721–728.

[53] Demetrescu, Camil. 2006. 9th DIMACS Implementation Challenge. `http://users.diag.uniroma1.it/challenge9/download.shtml`.

[54] Diamos, Greg, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Y. Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-Chip. In *Proceedings of the 33nd International Conference on Machine Learning*, 2024–2033. ICML.

[55] Dijkstra, Edsger W. 1982. A tutorial on the split binary semaphore. In *Theoretical foundations of programming methodology: Lecture notes of an international summer school, directed by f. l. bauer, e. w. dijkstra and c. a. r. hoare*, 555–564.

[56] Dong, Shi, and David Kaeli. 2017. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *Proceedings of the General Purpose GPUs*, 63–72. GPGPU, New York, NY, USA: ACM.

[57] Dutu, Alexandru, Matthew D. Sinclair, Bradford M. Beckmann, David A. Wood, and Marcus Chow. 2020. Independent Forward Progress of Work-groups. In *Proceedings of the 47th International Symposium on Computer Architecture*. ISCA.

[58] Egielski, Ian J., Jesse Huang, and Eddy Z. Zhang. 2014. Massive Atomics for Massive Parallelism on GPUs. In *Proceedings of the 2014 International Symposium on Memory Management*, 93–103. ISMM, New York, NY, USA: Association for Computing Machinery.

[59] El Hajj, Izzat, Juan Gomez-Luna, Cheng Li, Li-Wen Chang, Dejan Milojicic, and Wen-mei Hwu. 2016. KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism. In *MICRO*, 1–12.

[60] Elteir, Marwa, Heshan Lin, and Wu-Chun Feng. 2011. Performance Characterization and Optimization of Atomic Operations on AMD GPUs. In *IEEE International Conference on Cluster Computing*, 234–243.

[61] Feinberg, B., B. C. Heyman, D. Mikhailenko, R. Wong, A. C. Ho, and E. Ipek. 2020. Commutative Data Reordering: A New Technique to Reduce Data Movement Energy on Sparse Inference Workloads. In *ISCA*, 1076–1088.

[62] Feng, Yuan. 2023. Understanding Scalability of Multi-GPU Systems. In *ACM Workshop on General Purpose GPUs*. GPGPU'23.

[63] Fiestas, J. I., and R. E. Bustamante. 2019. A Survey on the Performance of Different Mutex and Barrier Algorithms. In *IEEE XXVI International Conference on Electronics, Electrical Engineering and Computing*, 1–4. INTERCON.

[64] Fousek, Jan, Jiři Filipovič, and Matuš Madzin. 2011. Automatic Fusions of CUDA-GPU Kernels for Parallel Map. *SIGARCH Comput. Archit. News* 39(4):98–99.

[65] Franey, Sean, and Mikko Lipasti. 2013. Accelerating Atomic Operations on GPGPUs. In *Seventh IEEE/ACM International Symposium on Networks-on-Chip*, 1–8. NoCS.

[66] Fu, Yaosheng, Evgeny Bolotin, Niladrish Chatterjee, David Nellans, and Stephen W. Keckler. 2021. GPU Domain Specialization via Composable On-Package Architecture. `2104.02188`.

[67] Gaster, Benedict R., Derek Hower, and Lee Howes. 2015. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Trans. Archit. Code Optim.* 12(1):7:1–7:26.

[68] Gebhart, M., S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. 2012. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *MICRO*, 96–106.

[69] Gelado, Isaac, and Michael Garland. 2019. Throughput-Oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 27–37. PPoPP.

[70] Google. 2017. Hot Chips 2017: A Closer Look At Google's TPU v2.

[71] Gottlieb, Allan, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. 1983. The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers* C-32(2):175–189.

[72] Gustafson, John L. 2011. Gustafson's law. In *Encyclopedia of parallel computing*, 819–825.

[73] Gutierrez, Anthony, Bradford M. Beckmann, Alexandru Dutu, Joseph Gross, Michael LeBeane, John Kalamatianos, Onur Kayiran, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Matthew D. Sinclair, Michael Wyse, Jieming Yin, Xianwei Zhang, Akshay Jain, and Timothy Rogers. 2018. Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level. In *IEEE International Symposium on High Performance Computer Architecture*, 608–619. HPCA.

[74] Hajj, I. E., J. Gomez-Luna, C. Li, L. Chang, D. Milojicic, and W. Hwu. 2016. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, 1–12. MICRO.

[75] Ham, T. J., L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, 1–13. MICRO.

[76] Han, Hwansoo, and Chau-Wen Tseng. 1998. Compile-time Synchronization Optimizations for Software DSMs. In *Proceedings of the first merged*

international parallel processing symposium and symposium on parallel and distributed processing, 662–669.

[77] Han, Song, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 243–254. ISCA, Piscataway, NJ, USA: IEEE Press.

[78] Harris, Mark, and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. `https://devblogs.nvidia.com/cooperative-groups/`.

[79] He, Bijun, William N. Scherer, and Michael L. Scott. 2005. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. In *High Performance Computing*, 7–18. HIPC.

[80] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *CoRR* abs/1512.03385. `1512.03385`.

[81] Hechtman, B.A., Shuai Che, D.R. Hower, Yingying Tian, B.M. Beckmann, M.D. Hill, S.K. Reinhardt, and D.A. Wood. 2014. QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs. In *20th International Symposium on High Performance Computer Architecture*, 189–200. HPCA.

[82] Hechtman, Blake A., and Daniel J. Sorin. 2013. Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips. In *International Symposium on Performance Analysis of Systems and Software*, 118–119. ISPASS.

[83] Hensgen, Debra, Raphael Finkel, and Udi Manber. 1988. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.* 17(1):1–17.

[84] Hestness, Joel, Stephen W. Keckler, and David A. Wood. 2014. A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior. In *IEEE International Symposium on Workload Characterization*, 150–160. IISWC.

[85] Hower, Derek R., Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 427–440. ASPLOS, New York, NY, USA: ACM.

[86] Howes, Lee, and Aaftab Munshi. 2015. The OpenCL Specification, Version 2.0. Khronos Group.

[87] HSA Foundation. 2015. HSA Platform System Architecture Specification. `http://www.hsafoundation.com/?ddownload=4944`.

[88] Hu, B., and C. J. Rossbach. 2020. Altis: Modernizing GPGPU Benchmarks. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 1–11. ISPASS.

[89] Huzaifa, Muhammad, Johnathan Alsop, Abdulrahman Mahmoud, Giordano Salvador, Matthew D. Sinclair, and Sarita V. Adve. 2020. Inter-Kernel Reuse-Aware Thread Block Scheduling. *ACM Trans. Archit. Code Optim.* 17(3).

[90] Jain, Animesh, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *45th ACM/IEEE Annual International Symposium on Computer Architecture*, 776–789. ISCA.

[91] James, Dave. 2018. AMD's answer to Nvidia's NVLink is xgmi, and it's coming to the new 7nm Vega GPU.

[92] ———. 2019. Nvidia has “de-risked” multiple chiplet GPU designs – “now it’s a tool in the toolbox”. https://www.pcgamesn.com/nvidia/graphics-card-chiplet-designs.

[93] Jamieson, Charles, Anushka Chandrashekar, Ian McDougall, and Matthew D. Sinclair. 2022. GAP: gem5 GPU Accuracy Profiler. In *4th gem5 Users’ Workshop*.

[94] Jerger, N. E., A. Kannan, Z. Li, and G. H. Loh. 2014. NoC Architectures for Silicon Interposer Systems: Why Pay for more Wires when you Can Get them (from your interposer) for Free? In *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 458–470. MICRO.

[95] Jiang, Dongming, Hongzhang Shan, and Jaswinder Pal Singh. 1997. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 217–229. PPOPP ’97, New York, NY, USA: Association for Computing Machinery.

[96] Jouppi, Norman P., Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Nishant Patil, Sushma Prasad, Clifford Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons from Three Generations Shaped Google’s TPUv4i. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA.

[97] Jouppi, Norman P., 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, Gotti Rajendra, 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. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 1–12. ISCA, New York, NY, USA: ACM.

[98] Jradi, W. A. R., H. A. Dantas do Nascimento, and W. Santos Martins. 2018. A Fast and Generic GPU-Based Parallel Reduction Implementation. In *Symposium on High Performance Computing Systems*, 16–22. WSCAD.

[99] Kandiah, Vijay, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO*, 738–753.

[100] Kannan, Ajaykumar, Natalie Enright Jerger, and Gabriel H Loh. 2015. Enabling Interposer-based Disintegration of Multi-core Processors. In *48th Annual IEEE/ACM International Symposium on Microarchitecture*, 546–558. MICRO, IEEE.

[101] Kannan, Ajaykumar, Natalie Enright Jerger, and Gabriel H. Loh. 2016. Exploiting Interposer Technologies to Disintegrate and Reintegrate Multicore Processors. *IEEE Micro* 36(3):84–93.

[102] Keckler, Stephen W. 2011. Life After Dennard and How I Learned to Love the Picojoule. Keynote at MICRO.

[103] Kerr, Andrew, Duane Merrill, Julien Demouth, and John Tran. 2017. CUTLASS: Fast Linear Algebra in CUDA C++. `https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/`.

[104] Khairy, Mahmoud, Akshay Jain, Tor M. Aamodt, and Timothy G. Rogers. 2018. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling. *CoRR* abs/1810.07269. `1810.07269`.

[105] Khairy, Mahmoud, Vadim Nikiforov, David Nellans, and Timothy G. Rogers. 2020. Locality-Centric Data and Threadblock Management for Massive GPUs. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 1022–1036. MICRO.

[106] Khairy, Mahmoud, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 473–486. ISCA.

[107] Khairy, Mahmoud, Mohamed Zahran, and Amr Wassal. 2017. SACAT: Streaming-Aware Conflict-Avoiding Thrashing-Resistant GPGPU Cache Management Scheme. *IEEE Transactions on Parallel and Distributed Systems* 28(6):1740–1753.

[108] Khorasani, Farzad, Hodjat Asghari Esfeden, Nael Abu-Ghazaleh, and Vivek Sarkar. 2018. In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization. In *Proceedings of 51st IEEE/ACM International Symposium on Microarchitecture*. MICRO.

[109] Khronos OpenCL Working Group. 2021. The OpenCL Specification. `https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`.

[110] Khude, N., I. Stanciulescu, D. Melanz, and D. Negrut. 2013. Efficient Parallel Simulation of Large Flexible Body Systems With Multiple Contacts. *ASME Journal of Computational and Nonlinear Dynamics* 8(4):041003–041003.

[111] Kim, Hyojong, Ramyad Hadidi, Lifeng Nai, Hyesoon Kim, Nuwan Jayasena, Yasuko Eckert, Onur Kayiran, and Gabriel Loh. 2018. CODA: Enabling Co-Location of Computation and Data for Multiple GPU Systems. *ACM Trans. Archit. Code Optim.* 15(3).

[112] Kim, Ji, and Christopher Batten. 2014. Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists. In *MICRO*, 75–87.

[113] Kim, W., S. Tavarageri, P. Sadayappan, and J. Torrellas. 2016. Architecting and Programming a Hardware-Incoherent Multiprocessor Cache Hierarchy. In *IPDPS*, 555–565.

[114] Klenk, B., N. Jiang, G. Thorson, and L. Dennison. 2020. An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives. In *ISCA*, 996–1009.

[115] Komuravelli, Rakesh, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Prakalp Srivastava, Maria Kotsifakou, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache it Too. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 707–719. ISCA.

[116] Kontothanassis, Leonidas, Robert Stets, Galen Hunt, Umit Rencuzogullari, Gautam Altekar, Sandhya Dwarkadas, and Michael L. Scott. 2005. Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks. *ACM Trans. Comput. Syst.* 23(3):301–335.

[117] Kotra, Jagadish B, Michael LeBeane, Mahmut T Kandemir, and Gabriel H Loh. 2021. Increasing GPU Translation Reach by Leveraging Under-

Utilized On-Chip Resources. In *54th Annual IEEE/ACM International Symposium on Microarchitecture*, 1169–1181. MICRO.

[118] Koukos, Konstantinos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead. *ACM Trans. Archit. Code Optim.* 13(1):1:1–1:22.

[119] Krashinsky, Ronny, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. 2020. NVIDIA Ampere Architecture In-Depth. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`.

[120] Kuper, Reese, Suchita Pati, and Matthew D. Sinclair. 2021. Improving GPU Utilization in ML Workloads Through Finer-Grained Synchronization. In *3rd Young Architects Workshop*. YArch.

[121] Lamport, Leslie. 1974. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM* 17(8):453–455.

[122] Lawrence Livermore National Labs. 2020. CORAL-2 Benchmarks. `https://asc.llnl.gov/coral-2-benchmarks`.

[123] LeBeane, Michael, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2018. ComP-Net: Command Processor Networking for Efficient Intra-Kernel Communications on GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18, New York, NY, USA: Association for Computing Machinery.

[124] LeBeane, Michael, Brandon Potter, Abhisek Pan, Alexandru Dutu, Vinay Agarwala, Wonchan Lee, Deepak Majeti, Bibek Ghimire, Eric Van Tassell, Samuel Wasmundt, Brad Benton, Mauricio Breternitz, Michael L. Chu, Mithuna Thottethodi, Lizy K. John, and Steven K. Reinhardt. 2016. Extended Task Queuing: Active Messages for Heterogeneous Systems.

In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 933–944. SC.

[125] Lee, J. H., J. Sim, and H. Kim. 2015. BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models. In *International Conference on Parallel Architecture and Compilation*, 241–252. PACT.

[126] Lee, Jinsu, Juhyoung Lee, Donghyeon Han, Jinmook Lee, Gwangtae Park, and Hoi-Jun Yoo. 2019. An energy-efficient sparse deep-neural-network learning accelerator with fine-grained mixed precision of fp8–fp16. *IEEE Solid-State Circuits Letters* 2(11):232–235.

[127] Lelbach, Bryce Adelstein, Olivier Giroux, JF Bastien, Detlef Vollmann, and David Olsen. 2019. P1135R5: The C++20 Synchronization Library. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1135r5.html`.

[128] Leng, Jingwen, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 487–498. ISCA, New York, NY, USA: Association for Computing Machinery.

[129] Lew, Jonathan, Deval Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. 2018. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. *CoRR* abs/1811.08933. `1811.08933`.

[130] ———. 2019. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software*. ISPASS.

[131] Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28(2): 39–55.

[132] Lipasti, M. H., B. Falsafi, J. F. Cantin, J. E. Smith, and A. Moshovos. 2006. Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays. *IEEE Micro* 26(01):70–79.

[133] Loh, Gabriel H., Natalie Enright Jerger, Ajaykumar Kannan, and Yasuko Eckert. 2015. Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems. In *Proceedings of the 2015 International Symposium on Memory Systems*, 3–10. MEMSYS '15, New York, NY, USA: ACM.

[134] Lowe-Power, Jason, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 simulator: Version 20.0+. `2007.03152`.

[135] Luitjens, Justin. 2014. CUDA Streams: Best Practices and Common Pitfalls.

[136] Lustig, Daniel, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 257–270. ASPLOS, New York, NY, USA: Association for Computing Machinery.

[137] Mazhar, H., T. Heyn, A. Tasora, and D. Negrut. 2015. Using Nesterov's Method to Accelerate Multibody Dynamics with Friction and Contact. *ACM Trans. Graph.* 34(3):32:1–32:14.

[138] Mazhar, H., A. Pazouki, M. Rakhsha, P. Jayakumar, and D. Negrut. 2018. A Differential Variational Approach for Handling Fluid-solid Interaction Problems via Smoothed Particle Hydrodynamics. *Journal of Computational Physics* 371:92–119.

[139] Mazhar, Hammad, Toby Heyn, and Dan Negrut. 2011. A Scalable Parallel Method for Large Collision Detection Problems. *Multibody System Dynamics* 26:37–55. 10.1007/s11044-011-9246-y.

[140] Mellor-Crummey, John M., and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9(1):21–65.

[141] Merrill, Duane. 2020. NVIDIA CUB Library. `https://nvlabs.github.io/cub/`.

[142] Metcalfe, Robert M., and David R. Boggs. 1976. Ethernet: Distributed packet switching for local computer networks. *Commun. ACM* 19(7): 395–404.

[143] Milic, Ugljesa, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 123–135. MICRO-50, New York, NY, USA: ACM.

[144] Miller, dePaul, Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. 2021. KVCG: A Heterogeneous Key-Value Store for Skewed Workloads. In *Proceedings of the 14th ACM International Conference on Systems and Storage*. SYSTOR '21.

[145] Mojumder, Saiful A., Yifan Sun, Leila Delshadtehrani, Yenai Ma, Trinayan Baruah, José L. Abellán, John Kim, David Kaeli, and Ajay Joshi. 2020. HALCONE: A Hardware-Level Timestamp-based Cache Coherence Scheme for Multi-GPU systems.

[146] Mukkara, Anurag, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 1009–1022. MICRO, New York, NY, USA: ACM.

[147] Munford, ML, VR Lima, TO Vieira, G Heinzelmann, TB Creczynski-Pasa, and AA Pasa. 2005. AFM In-situ Characterization of Supported Phospholipid Layers Formed by Vesicle Fusion. *Microscopy and Microanalysis* 11(S03):90–93.

[148] Naffziger, Samuel, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. 2021. Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture*, 57–70. ISCA.

[149] Narang, Sharan. 2016. DeepBench. `https://github.com/baidu-research/DeepBench`.

[150] Narang, Sharan, and Greg Diamos. 2017. An update to DeepBench with a focus on deep learning inference. `https://svail.github.io/DeepBench-update/`.

[151] Nelson, Jacob, dePaul Miller, and Roberto Palmieri. 2022. Don't forget about synchronization! Guidelines for using locks on graphics processing units. *Concurrency and Computation: Practice and Experience* 34(2): e5757.

[152] Nelson, Jacob, and Roberto Palmieri. 2019. Don't Forget About Synchronization!: A Case Study of K-Means on GPU. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, 11–20. PMAM.

[153] Neugebauer, Rolf, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 327–341. SIGCOMM '18, New York, NY, USA: Association for Computing Machinery.

[154] Niu, Feng, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, 693–701. NeurIPS, USA: Curran Associates Inc.

[155] Noh, Seock-Hwan, Jahyun Koo, Seunghyun Lee, Jongse Park, and Jaeha Kung. 2022. Flexblock: A flexible dnn training accelerator with multi-mode block floating point support. `2203.06673`.

[156] NVIDIA. 2013. CUDA HyperQ Example. http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[157] ———. 2016. CUDA programming guide. CUDA Programming guide.

[158] ———. 2016. NVIDIA CUDA C Programming Guide v7.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed August 6, 2016.

[159] ———. 2016. NVIDIA RISC-V Story. *4th RISC-V Workshop*.

[160] ———. 2016. Pascal P100. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[161] ———. 2017. Cuda-gdb. http://docs.nvidia.com/cuda/cuda-gdb/index.html/. Accessed Nov 15, 2017.

[162] ———. 2018. CUDA Stream Management. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__STREAM.html.

[163] ———. 2018. Nvidia, cuda stream management.

[164] ———. 2018. NVIDIA cuDNN: GPU Accelerated Deep Learning. https://developer.nvidia.com/cudnn.

[165] ———. 2020. CUDA C++ Programming Guide. CUDA programming guide.

[166] ———. 2020. libcu++: The C++ Standard Library for Your Entire System. https://nvidia.github.io/libcudacxx/.

[167] ———. 2020. Split Arrive/Wait Barrier. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[168] NVIDIA Corp. 2017. Inside Volta: The World's Most Advanced Data Center GPU. https://devblogs.nvidia.com/parallelforall/inside-volta/.

[169] ———. 2018. NVLink Fabric: A Faster, More Scalable Interconnect. `https://www.nvidia.com/en-us/data-center/nvlink/`.

[170] Nyland, Lars, and Stephen Jones. 2013. Understanding and Using Atomic Memory Operations. In *Proceedings of GPU Technology Conference*. GTC.

[171] Orr, Marc S., Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. 2015. Synchronization Using Remote-Scope Promotion. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 73–86. ASPLOS, New York, NY, USA: Association for Computing Machinery.

[172] O'Connor, Mike, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *MICRO*, 41–54.

[173] Pal, Saptadeep, Daniel Petrisko, Matthew Tomei, Puneet Gupta, Subramanian S. Iyer, and Rakesh Kumar. 2019. Architecting Waferscale Processors - A GPU Case Study. In *25th IEEE International Symposium on High Performance Computer Architecture*, 250–263. HPCA.

[174] Park, Jason Jong Kyu, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 593–606. ASPLOS '15.

[175] Pazouki, Arman, Michał Kwarta, Kyle Williams, William Likos, Radu Serban, Paramsothy Jayakumar, and Dan Negrut. 2017. Compliant Contact

Versus Rigid Contact: A Comparison in the Context of Granular Dynamics. *Physical Review E* 96(4):042905.

[176] Pazouki, Arman, Hammad Mazhar, and Dan Negrut. 2012. Parallel Collision Detection of Ellipsoids with Applications in Large Scale Multibody Dynamics. *Mathematics and Computers in Simulation* 82(5):879–894.

[177] Peterson, Gary L., and Michael J. Fischer. 1977. Economical Solutions for the Critical Section Problem in a Distributed System (Extended Abstract). In *Proceedings of the ninth annual acm symposium on theory of computing*, 91–97. STOC.

[178] Peterson, G.L. 1981. Myths about the Mutual Exclusion Problem. *Information Processing Letters* 12(3):115–116.

[179] Pirzada, Usman. 2019. NVIDIA Next Generation Hopper GPU Leaked – Based On MCM Design, Launching After Ampere. `https://wccftech.com/nvidia-hopper-gpu-mcm-leaked/`.

[180] Pourghassemi, Behnam, Chenghao Zhang, Joo Hwan Lee, and Aparna Chandramowlishwaran. 2020. On the Limits of Parallelizing Convolutional Neural Networks on GPUs. In *SPAA*, 567–569.

[181] Pratheek, B., N. Jawalkar, and A. Basu. 2022. Designing Virtual Memory System of MCM GPUs. In *55th IEEE/ACM International Symposium on Microarchitecture*, 404–422. MICRO, Los Alamitos, CA, USA: IEEE Computer Society.

[182] Puthoor, Sooraj, Ashwin M. Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M. Beckmann, and Gregory Rodgers. 2016. Implementing Directed Acyclic Graphs with the Heterogeneous System Architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, 53–62. GPGPU '16, New York, NY, USA: ACM.

[183] Puthoor, Sooraj, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. 2018. Oversubscribed Command Queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs*, 50–60. GPGPU-11, New York, NY, USA: ACM.

[184] Qin, Eric, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 ieee international symposium on high performance computer architecture (hpca)*, 58–70. IEEE.

[185] Qualcomm. 2021. Qualcomm Hexagon DSP. `https://developer.qualcomm.com/sites/default/files/docs/adreno-gpu/developer-guide/dsp/dsp.html`.

[186] Raihan, Md Aamir, Negar Goli, and Tor M. Aamodt. 2018. Modeling Deep Learning Accelerator Enabled GPUs. *CoRR* abs/1811.08309. `1811.08309`.

[187] Ramadas, Vishnu, Daniel Kouchekinia, Ndubuisi Osuji, and Matthew D. Sinclair. 2023. Closing the GAP: Improving the Accuracy of gem5's GPU Models. In *5th gem5 Users' Workshop*.

[188] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 779–788. CVPR.

[189] Ren, X., D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *International Symposium on High Performance Computer Architecture*, 582–595. HPCA.

[190] Ren, Xiaowei, and Mieszko Lis. 2017. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *International Symposium on High Performance Computer Architecture*, 625–636. HPCA, IEEE.

[191] Rhee, I. 1996. Optimizing a FIFO, scalable spin lock using consistent memory. In *17th IEEE Real-Time Systems Symposium*, 106–114. RTSS.

[192] Rhu, Minsoo, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *International Symposium on High Performance Computer Architecture*, 78–91. HPCA.

[193] Rinard, Martin C. 1999. Effective Fine-Grain Synchronization for Automatically Parallelized Programs Using Optimistic Synchronization Primitives. *ACM Transactions on Computer Systems* 17(4):337–371.

[194] Rivas, M. A., and M. G. Harbour. 2003. Evaluation of new POSIX real-time operating systems services for small embedded platforms. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, 161–168.

[195] Rodchenko, Andrey, Andy Nisbet, Antoniu Pop, and Mikel Luján. 2015. Effective Barrier Synchronization on Intel Xeon Phi Coprocessor. In *Euro-par 2015: Parallel processing*, ed. Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, 588–600. Berlin, Heidelberg: Springer Berlin Heidelberg.

[196] Rossi, Ryan A., and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Aaai*.

[197] Rudolph, Larry, and Zary Segall. 1984. Dynamic Decentralized Cache Schemes for Mimd Parallel Processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 340–347. ISCA '84.

[198] Saleh, Skyler J, Samuel Naffziger, Milind S Bhagavat, and Rahul Agarwal. 2020. GPU Chiplets Using High Bandwidth Crosslinks. US Patent App. 16/456,287.

[199] Salvador, Giordano, Wesley H. Darvin, Muhammad Huzaifa, Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. 2020. Specializing Coherence, Consistency, and Push/Pull for GPU Graph Analytics. In *ISPASS*.

[200] Scott, Steven L. 1996. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 26–36. ASPLOS, New York, NY, USA.

[201] Sengupta, Shubhabrata, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan Primitives for GPU Computing. In *Siggraph/eurographics workshop on graphics hardware*, ed. Mark Segal and Timo Aila. The Eurographics Association.

[202] Shao, Yakun Sophia, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 14–27. MICRO '52, New York, NY, USA: Association for Computing Machinery.

[203] Sinclair, Matthew D., Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 647–659. MICRO.

[204] ———. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 161–174. ISCA.

[205] ———. 2017. HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs. In *IEEE International Symposium on Workload Characterization*. IISWC.

[206] Singh, Inderpreet, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. 2013. Cache Coherence for GPU Architectures. In *19th International Symposium on High Performance Computer Architecture*, 578–590. HPCA.

[207] Smith, Ryan. 2018. NVIDIA Develops NVLink Switch: NVSwitch, 18 Ports for DGX-2 & More.

[208] Sorensen, Tyler, Alastair F Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable Inter-Workgroup Barrier Synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 39–58. OOPSLA.

[209] Sorensen, Tyler, Sreepathi Pai, and Alastair F. Donaldson. 2019. One Size Doesn't Fit All: Quantifying Performance Portability of Graph Applications on GPUs. In *IEEE International Symposium on Workload Characterization*. IISWC.

[210] Springer, Matthias, Peter Wauligmann, and Hidehiko Masuhara. 2017. Modular Array-Based GPU Computing in a Dynamically-Typed Language. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 48–55. ARRAY 2017, New York, NY, USA: Association for Computing Machinery.

[211] Stone, John E., David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12(3):66.

[212] Stuart, Jeff A., and John D. Owens. 2011. Efficient Synchronization Primitives for GPUs. *CoRR* abs/1110.4623. `1110.4623`.

[213] Stuecheli, J., B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59(1):7:1–7:7.

[214] Sun, Yifan, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, 197–209. ISCA '19, New York, NY, USA: Association for Computing Machinery.

[215] Sun Microsystems, Inc. 2008. OpenSparc T2 system-on-chip (SoC) microarchitecture specification. `http://www.opensparc.net/opensparc-t2/index.html`.

[216] Tabbakh, A., X. Qian, and M. Annavaram. 2018. G-TSC: Timestamp Based Coherence for GPUs. In *HPCA*, 403–415.

[217] Takada, H., and K. Sakamura. 1994. Predictable spin lock algorithms with preemption. In *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*, 2–6.

[218] Tang, Peiyi, and Pen-Chung Yew. 1990. Software combining algorithms for distributing hot-spot addressing. *Journal of Parallel and Distributed Computing* 10(2):130–139.

[219] Tasora, A., R. Serban, H. Mazhar, A. Pazouki, D. Melanz, J. Fleischmann, M. Taylor, H. Sugiyama, and D. Negrut. 2016. CHRONO: An open source multi-physics dynamics engine. In *High Performance Computing in Science and Engineering – Lecture Notes in Computer Science*, ed. T. Kozubek, 19–49. Cham: Springer.

[220] Ukidave, Y., X. Li, and D. Kaeli. 2016. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *IEEE International Parallel and Distributed Processing Symposium*, 353–362. IPDPS.

[221] Vijayaraghavan, T., Y. Eckert, G. H. Loh, M. J. Schulte, M. Ignatowski, B. M. Beckmann, W. C. Brantley, J. L. Greathouse, W. Huang, A. Karunanithi, O. Kayiran, M. Meswani, I. Paul, M. Poremba, S. Raasch, S. K. Reinhardt, G. Sadowski, and V. Sridharan. 2017. Design and Analysis of an APU for Exascale Computing. In *Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture*, 85–96. HPCA.

[222] Villa, Oreste, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 372–383. MICRO, New York, NY, USA: Association for Computing Machinery.

[223] Vineet, V., and P. J. Narayanan. 2008. CUDA cuts: Fast graph cuts on the GPU. In *CVPR Workshops*, 1–8.

[224] Wang, Guibin, YiSong Lin, and Wei Yi. 2010. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, 344–350. GREENCOM-CPSCOM '10, USA: IEEE Computer Society.

[225] Wang, Linnan, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 41–53. PPoPP, New York, NY, USA: ACM.

[226] Wang, Yangzihao, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP.

[227] Wang, Z., J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing. In *IEEE International Symposium on High Performance Computer Architecture*, 358–369. HPCA.

[228] Wang, Zhenning, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of Service Support for Fine-Grained Sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 269–281. ISCA, New York, NY, USA: ACM.

[229] Wittenbrink, C. M., E. Kilgariff, and A. Prabhu. 2011. Fermi GF100 GPU Architecture. *IEEE Micro* 31(2):50–59.

[230] Wood, David A., Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. 1993. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 156–167. ISCA '93, New York, NY, USA: Association for Computing Machinery.

[231] Wu, Bo, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming*

Languages and Operating Systems, 483–496. ASPLOS, New York, NY, USA: ACM.

[232] Xiang, Ping, Yi Yang, and Huiyang Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *20th IEEE International Symposium on High Performance Computer Architecture*, 284–295. HPCA.

[233] Xiao, Shucai, and Wu Feng. 2010. Inter-Block GPU Communication via Fast Barrier Synchronization. In *IEEE International Parallel and Distributed Processing Symposium*, 1–12. IPDPS.

[234] Xu, Q., H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 230–242. ISCA.

[235] Yang, Dingqing, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2020. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd annual ieee/acm international symposium on microarchitecture (micro)*, 711–724. IEEE.

[236] Yang, Yifan, Zhaoshi Li, Yangdong Deng, Zhiwei Liu, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. GraphABCD: Scaling out Graph Analytics with Asynchronous Block Coordinate Descent. In *ISCA*, 419–432.

[237] Yeh, Tsung Tai, Matthew D. Sinclair, Bradford M. Beckmann, and Timothy G. Rogers. 2021. Deadline-Aware Offloading for High-Throughput Accelerators. In *27th IEEE International Symposium on High Performance Computer Architecture*, 479–492. HPCA.

[238] Yogatama, Bobbi W., Matthew D. Sinclair, and Michael M. Swift. 2020. Enabling Multi-GPU Support in gem5. In *3rd gem5 Users' Workshop*.

[239] Young, Vinson, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW Mechanisms to

Improve NUMA Performance of Multi-GPU Systems. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 339–351. MICRO.

[240] Zhang, Hao, Yuan Li, Zhijie Deng, and Xiaodan Liang. 2020. AutoSync: Learning to Synchronize for Data-Parallel Distributed Deep Learning. In *Advances in Neural Information Processing Systems*. NeurIPS.

[241] Zhang, Lingqi, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. 2020. A Study of Single and Multi-device Synchronization Methods in Nvidia GPUs. In *IEEE International Parallel and Distributed Processing Symposium*, 483–493. IPDPS.

[242] Zhang, Guowei and Horn, Webb and Sanchez, Daniel. 2015. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems. In *48th Annual IEEE/ACM International Symposium on Microarchitecture*, 13–25. MICRO.

[243] Zheng, Bojian, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 1089–1102. ISCA '20, IEEE Press.

[244] Zheng, Long, Jieshan Zhao, Yu Huang, Qinggang Wang, Zhen Zeng, Jingling Xue, Xiaofei Liao, and Hai Jin. 2020. Spara: An energy-efficient reram-based accelerator for sparse graph analytics applications. In *2020 ieee international parallel and distributed processing symposium (ipdps)*, 696–707. IEEE.

[245] Zhou, Yuanyuan, Liviu Iftode, and Kai Li. 1996. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, 75–88. OSDI '96, New York, NY, USA: Association for Computing Machinery.

[246] Zhu, Feiwen, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie. 2018. Sparse Persistent RNNs: Squeezing Large Recurrent Networks On-Chip. In *Proceedings of 6th International Conference on Learning Representations*. ICLR.

[247] Zhu, Hongyu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. TBD: Benchmarking and Analyzing Deep Neural Network Training. In *IEEE International Symposium on Workload Characterization*. IISWC.

[248] Zhu, Maohua, Minsoo Rhu, Jason Clemons, Stephen W Keckler, and Yuan Xie. 2016. Training Long Short-Term Memory With Sparsified Stochastic Gradient Descent. `https://openreview.net/forum?id=HJWzXsKxx`.