HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs

Matthew D. Sinclair Johnathan Alsop Sarita V. Adve
University of Illinois at Urbana-Champaign
hetero@cs.illinois.edu

***Abstract*—Traditionally GPUs focused on streaming, data-parallel applications, with little data reuse or sharing and coarse-grained synchronization. However, the rise of general-purpose GPU (GPGPU) computing has made GPUs desirable for applications with more general sharing patterns and fine-grained synchronization, especially for recent GPUs that have a unified address space and coherent caches. Prior work has introduced microbenchmarks to measure the impact of these changes, but each paper uses its own set of microbenchmarks. In this work, we combine several of these sets together in a single suite, HeteroSync. HeteroSync includes several synchronization primitives, data sharing at different levels of the memory hierarchy, and relaxed atomics. We characterize the scalability of HeteroSync for different coherence protocols and consistency models on modern, tightly coupled CPU-GPU systems and show that certain algorithms, coherence protocols, and consistency models scale better than others.**

I. Introduction

Traditional heterogeneous systems had loosely coupled memory hierarchies and required programmers to explicitly copy data between different accelerators via main memory to keep data coherent. In an effort to make heterogeneous systems more programmable and efficient, industry has recently transitioned to more tightly coupled heterogeneous systems with a unified global address space and coherent caches across CPUs and accelerators (primarily GPUs) [1], [2]. A global address space allows data to be transparently moved between accelerators in hardware and provides high performance for the simple, streaming, data parallel applications that heterogeneous systems traditionally run. Since these applications have little or no sharing or data reuse, heterogeneous systems use simple, software-driven coherence protocols that assume data-race-freedom, regular data accesses, and mostly coarse-grained synchronization. These protocols invalidate the entire cache at acquires and flush (writethrough) all dirty data before the next release [3]. Since synchronization (implemented with *atomics*) is infrequent, synchronization accesses bypass the private caches and are executed at the next shared level of the hierarchy. Prior work refers to this as *GPU coherence* [4], [5].

Thus, unlike multicore CPU coherence protocols, conventional heterogeneous coherence protocols are simple, without need for writer-initiated invalidations, ownership requests, downgrade requests, protocol state bits, or directories. Further, although consistency models for heterogeneous systems have been slow to be clearly defined [6], [7], [8], heterogeneous coherence implementations were amenable to the familiar sequentially consistent for data-race-free (SC-for-DRF, or DRF) memory model, enabling programmers to reason with the familiar SC model as long as there are no data races.

Although simple heterogeneous coherence protocols work well for traditional heterogeneous applications (e.g., [9]), emerging applications with more general sharing patterns and fine-grained synchronization [10], [11], [12], [13], [14], [15] suffer. For these applications, full cache invalidates, dirty data flushes, and executing synchronizations at the last level cache (LLC) are inefficient. Multicore CPUs use hardware coherence protocols, such as MESI, to overcome this problem but prior work has observed that hardware coherence protocols are a poor fit for conventional heterogeneous applications [3], [16].

Recent work on coherence and consistency for heterogeneous CPU-GPU systems has explored how to provide better efficiency for these emerging applications [3], [4], [5], [14], [17], [18]. Many of these papers introduce new microbenchmarks to measure the efficiency of their proposed changes, but often use different benchmarks that make it difficult to compare the proposed schemes. In this paper, we combine microbenchmarks from some of these papers [4], [5], [15] into a single suite, HeteroSync. HeteroSync includes microbenchmarks implementing various synchronization primitives, along with annotations for locally and globally scoped atomics [4], as well as relaxed atomics [5]. This paper describes these microbenchmarks and analyzes their scalability for a tightly coupled CPU- GPU system with a unified address space, coherent caches (implementing conventional GPU and DeNovo coherence protocols), and the DRF and heterogeneous-race-free (HRF) [12], [19] memory consistency models. We will release HeteroSync shortly to help researchers to further explore synchronization, coherence, and consistency for CPU-GPU systems.[1]

Other prior work designed more efficient algorithms with fine-grained synchronization for discrete GPUs [20], [21], [22], [23], [24], [25] or collaborative CPU-GPU computing [26], [27]. But unlike our work they either do not consider

[1] https://github.com/mattsinc/heterosync/

tightly-coupled systems or mostly focus on applications that do not use fine-grained synchronization. We further address related work in Section VI.

II. Background

A. Modern GPU Coherence and Consistency

Recent work has introduced a new consistency model, sequentially consistent for heterogeneous-race-free (SC-for-HRF, or HRF). HRF uses scoped synchronization to associate a synchronization access with a level of the memory hierarchy where the synchronization should occur [12], [19]. For example, a synchronization access with a local scope indicates that it synchronizes only the data accessed by the threads within its own CU, which all share the L1 cache. As a result, the synchronization can execute at the CU's L1 cache, without invalidating or flushing data to lower levels of the memory hierarchy. Thus, locally scoped synchronizations can significantly improve performance. However, scopes significantly complicate the consistency model and expose the memory hierarchy to the programmer.

B. DeNovo: A Hybrid Coherence Protocol

Other work has argued that heterogeneous consistency models should not be even more complex than the CPU models [4]. Instead, the authors extend the hybrid DeNovo coherence protocol for multicore CPUs to heterogeneous systems. Their results demonstrate that the DeNovo coherence protocol is a good fit for heterogeneous CPU-GPU systems because it provides high performance for a wide variety of applications and allows the use of a DRF consistency model that maintains SC semantics. The DeNovo coherence protocol is a hybrid of both ownership- and GPU-style coherence protocols: like ownership-based protocols, DeNovo obtains ownership on both data stores and atomics and uses writeback caches; like GPU-style coherence protocols, it uses data-race-freedom to do reader-initiated self-invalidations.

C. Relaxed Atomics

DRF consistency models require all racy accesses be identified as synchronization accesses (e.g., using the atomic keyword in C, C++, OpenCL, and HSA). By default, DRF models severely restrict the reordering or overlapping of program ordered atomics with respect to each other. Since atomics are usually infrequent, this requirement is reasonable for most applications. However, in some cases these constraints cause unnecessarily high overheads. This led to the introduction of relaxed atomics in the C++ memory model (and later other memory models) [28]. If an access is identified as being relaxed, it can be reordered and overlapped with all other (non-dependent) memory accesses in the program. Thus, efficiency can potentially be significantly improved by using relaxed atomics.

However, it is widely acknowledged that it is extremely difficult to use relaxed atomics correctly. As a result, it is strongly recommended that only experts writing absolutely performance critical code use them [29], [30]. Nevertheless, programmers still use relaxed atomics in heterogeneous systems, because they can significantly improve efficiency. We introduced the DRF-Relaxed (DRFrlx) consistency model was introduced to address this dichotomy: DRFrlx provides SC-centric semantics for the common uses of relaxed atomics in heterogeneous systems while retaining the efficiency benefits of relaxed atomics [5].

III. Microbenchmarks in HeteroSync

A. SyncPrims

The Synchronization Primitives (SyncPrims) microbenchmarks were originally designed by Stuart and Owens to study the performance of synchronization primitives on discrete GPUs [15]. These microbenchmarks, listed in Table I, include mutexes (Section III-A1), semaphores (Section III-A2), and barriers (Section III-A3) with a wide range of scalability characteristics. Their focus was on the performance of the atomic operations used to implement the synchronization primitives, not the overheads associated with properly synchronizing the global data accessed in a critical section. Consequently, the original SyncPrims algorithms do not have any global data accesses.

In prior work, we updated SyncPrims in three significant ways [4]: (1) to use global data accesses (instead of scratchpad accesses) in the critical sections; (2) to use synchronization loads and stores to enforce ordering instead of using inefficient (and sometimes incorrect) fences; and (3) creating two versions of most microbenchmarks to share data a different levels of the memory hierarchy to study synchronization with HRF's local and global scopes.

GPUs employ a relaxed memory model and provide some fence primitives to help enforce ordering. However, GPU memory model behavior has been slow to be clearly defined [6], [7], [31]. As a result, synchronization and consistency on modern GPUs is neither well-defined nor intuitive. Unsurprisingly, subsequent work identified situations where the fences in SyncPrims did not properly enforce ordering, partly due to the opaqueness and complexity of the CUDA memory consistency model [6], [31]. As mentioned above, we updated the SyncPrims microbenchmarks to use synchronization loads and stores to enforce ordering. On a synchronization load, all potentially stale data is invalidated and on a synchronization store all dirty data must be made visible to other CUs.[2]

The locally scoped versions only share data locally between thread blocks (TBs) on a CU, so they can improve efficiency and reduce contention by performing the synchronization accesses locally (with a per-CU mutex or semaphore), reusing data locally, and avoiding expensive

[2]To make dirty data visible to other CUs, DeNovo obtains ownership for the dirty data while GPU coherence writes the dirty data through to the shared LLC.

Microbenchmark	Description	Input
Mutexes		
Spin Mutex (SPM)	Test-and-set lock	4 TBs/CU, 10 Ld&St/thr/iter
Spin Mutex with backoff (SPMBO)	Test-and-set lock with backoff	
FA Mutex (FAM)	Centralized ticket lock	
Sleep Mutex (SLM)	Decentralized ticket lock	
Semaphores		
Spin Semaphore (SPS)	Semaphore with mutex lock	4 TBs/CU, readers: 10 Ld/thr/iter writers: 30 St/thr/iter
Spin Semaphore with backoff (SPSBO)	Semaphore with mutex lock and backoff	
Barriers		
Tree Barrier (TB)	Two-level atomic tree barrier	4 TBs/CU, 10 Ld&St/thr/iter
Tree Barrier with local exchange (TBEX)	Two-level atomic barrier with local exchange	
Lock-Free Tree Barrier (LFTB)	Two-level lock-free tree barrier	
Lock-Free Tree Barrier with local exchange (LFTBEX)	Two-level lock-free tree barrier with local exchange	

Table I: Synchronization primitive microbenchmarks with input sizes.

Algorithm 1 Basic structure of SyncPrims microbenchmarks. Although this example shows the Spin Mutex algorithm, the other SyncPrims microbenchmarks use a similar structure.

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

cache invalidations and store buffer flushes. The globally scoped versions share data across multiple CUs, so they must use global synchronization and cannot benefit from HRF.

The basic structure of the Spin Mutex microbenchmark is shown in Algorithm 1; the other SyncPrims microbenchmarks use a similar structure. Without loss of generality we use the CUDA terminology [32] to describe these algorithms and describe the globally scoped version (since the locally scoped versions are identical except for their use of per-CU mutexes or semaphores). Annotated code snippets for all algorithms we discuss in this section are available elsewhere [33].

1) Mutexes:

Spin Mutex (**SPM**): In this algorithm, a straightforward port of spin mutex algorithms on CPUs, each TB repeatedly tries to obtain a single mutex lock. Once the TB is done with the critical section, it releases the lock. However, SPM may suffer from high contention and is not deterministic or fair.

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

Fetch-and-Add Mutex (**FAM**): FA Mutex operates like a centralized ticket lock. Instead of directly acquiring a lock, each TB obtains a "ticket" that represents when the TB will be able to acquire the lock. Each TB then polls the current ticket number. When the ticket number equals this TBs ticket number, then the TB has acquired the lock. To release the lock, the TB simply increments the current ticket number. Although this algorithm is fair, because it uses two centralized counters (one for the head, one for the tickets) it can suffer from heavy contention.

Sleep Mutex (**SLM**): In the original Sleep Mutex algorithm, each TB places itself on the end of a ring buffer [15]. Afterwards, the TB repeatedly checks if it is at the front of the buffer by checking the shared head pointer. To release the lock, a TB increments the head pointer. However, this algorithm scales poorly because it requires more reads than FAM.

Subsequent work redesigned Sleep Mutex to be a *decentralized ticket lock* [4]. In the updated algorithm, after getting a ticket (similar to FAM), each TB spins on a unique location in the ring buffer. Reducing contention for a given location in the ring buffer improves performance by allowing each TB to spin locally[3] on its location. To transfer ownership of the lock to the next TB, instead of updating the head pointer, a TB instead updates the value in the next location in the ring buffer. Upon seeing this update, the TB spinning on that location now owns the lock and can proceed.

2) Semaphores:

Originally, the semaphore algorithms were reader-only [15]. To make the critical section more practical, subsequent work updated the semaphore algorithms to be reader-writer [4]: each CU has one writer TB and $N-1$ reader TBs. Each reader reads a subset of the data and the writer writes all of the data.

Spin Semaphore (**SPS**): In the original reader-only algorithm, a TB first acquires a mutex lock, then checks to see if there is room in the semaphore for it; if there is it updates the semaphore [15]. After this check, the TB releases the lock. When a TB is leaving the critical section, it uses a similar process: acquire lock, update semaphore to remove self, release lock. By using a semaphore, multiple TBs can enter the critical section simultaneously (as long as the max semaphore size is > 1). This scheme requires that the data

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

being accessed by each of these TBs is either independent or read-only. Moreover, if the centralized semaphore and mutex lock are heavily contended, performance suffers.

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

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

3) Barriers:

The original barrier microbenchmarks used globally scoped synchronization [15]. As in previous sections, subsequent work optimized the barrier algorithms [4]. First, they changed the single-level barriers into tree barriers where all TBs on a CU access unique data and join a local barrier before one TB from each CU joins the global barrier. After the global barrier, TBs exchange data across CUs for the subsequent iteration.

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

Lock-Free Tree Barrier (**LFTB**): The Lock-Free Barrier decentralizes the single-level barrier to reduce contention and improve both efficiency and scalability [15]. Each TB joins the barrier by setting a unique location in an array, then waits (using exponential backoff) for all other TBs to join the barrier. We converted the lock-free barrier into a tree barrier, similar to the atomic tree barrier, and ported it to the tightly coupled system.

B. Relaxed Atomics

Table II lists the relaxed atomics microbenchmarks. These microbenchmarks represent various use cases for relaxed atomics, which we identified in previous work [5]. Annotated code snippets for all algorithms we discuss in this section are available elsewhere [33].

Flags: Flags uses shared global flags to communicate between threads. The worker threads repeatedly loop until the main thread sets a ”stop” flag. While the workers are iterating, if certain conditions are met, then the worker sets a ”dirty” flag to signify something has been accessed that the main thread needs to clean up later. After the main thread has set stop, the workers exit and join a global barrier (the LFTB from Section III-A3) to ensure that all worker threads exit before the main thread accesses the dirty flag. Relaxed atomics (for setting the flags) allow more reordering without violating SC in Flags [5].

Microbenchmark	Input
Flags[30]	60 TBs
Histogram_global (HG)[34]	64 TBs, 256 KB, 256 bins
Histogram_global (HG-2K)[34]	64 TBs, 256 KB, 2K bins
Histogram_global-Non-Order (HG-NO)	64 TBs, 256 KB, 256 bins
RefCounter (RC)[30]	64 TBs
Seqlocks (SL)[35]	512 TBs
SplitCounter (SC)[36]	112 TBs

Table II: Relaxed atomic microbenchmarks with input sizes.

Histogram Global (**HG**): Histograms have multiple threads concurrently reading a shared array and atomically incrementing the corresponding bin of shared array. These updates can safely be relaxed (without violating SC) because the updates are commutative with one another. Similarly, once the increments to the shared counts are complete, accesses to the shared counters can also safely be relaxed because they do not order any other requests [5].

Previous work created several variants of this algorithm, including varying the number of bins and histogram_global_NO (HG-NO), which uses non-ordering relaxed atomics to access the shared counters in a separate kernel after the updates are complete [5]. These implementations are based on previous CUDA histogram implementation [34]. HG-NO uses a single TB, because the number of bins is relatively small.

Reference Counter (**RC**): Reference counters use shared global counters to keep track of how many threads are accessing a shared object. When an thread is no longer accessing the shared object, it atomically decrements the count for that object. If this thread was the last thread accessing object, then it is responsible for marking the object to be deleted. Similarly, when a thread starts accessing a shared object, it must atomically increment the object's shared counter. Unlike the previous microbenchmarks, relaxing the atomics may violate SC. Nevertheless, programmers are willing to tolerate some imprecision as long as the final decrement to each counter marks the shared object to be freed [30].

Seqlocks (**SL**): Seqlocks are a low cost alternative to mutex locks in applications where updates are infrequent. Seqlocks use a sequence number to track who else is accessing the shared data. In the (unlikely) case that a writer is trying to update the shared data, then the readers will throw away their speculatively read data and retry. On the other hand, if there were no concurrent writers (the common case), the reader(s) was able to read the shared data without any intervening synchronization. The atomic accesses to the shared data can

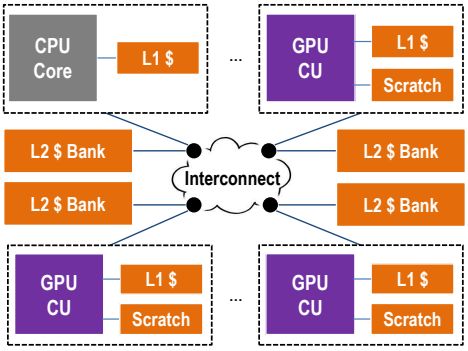


Figure 1: Baseline heterogeneous architecture [37].

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

Table III: Simulated heterogeneous system parameters.

be relaxed because any accesses that violate SC will only happen when there is a concurrent writer – and these values will be thrown away and not used [35].

Split Counter (**SC**): Somewhat like reference counters, split counters are often used in situations where programmers are willing to trade off SC guarantees for improved performance. In split counter, each thread has a counter that it atomically updates. Simultaneously, other threads are atomically reading the value of all of the counters to get a partial sum of the counters. Even though using relaxed atomics may cause non-SC behavior, relaxed atomics are often used in split counters because programmers do not need a precise value and desire getting a fast approximation of the partial sum [36].

IV. Methodology

A. Baseline Heterogeneous Architecture

To analyze the performance of the HeteroSync benchmarks, we model a tightly coupled CPU-GPU architecture with a unified shared memory address space and coherent caches. Our baseline system, illustrated in Figure 1 and similar to our prior work [4], [5], [37]. In this system, all CPU cores and GPU CUs are connected via an interconnection network. Each CPU core and GPU CU is on a separate network node. Each network node has an L1 cache (local to the CPU core or GPU CU) and a bank of the shared L2 cache (shared by all CPU cores and GPU CUs).[4] The GPU nodes also have a scratchpad. The coherence protocol, consistency model, and write policy depend on the system configuration studied (Section IV-C).

B. Simulation Environment and Parameters

To simulate a tightly coupled memory system, we use an integrated CPU-GPU simulator, which we obtained from the authors [4], [5], [37]. The integrated CPU-GPU simulator is built from the Simics full-system functional simulator to model the CPUs, the Wisconsin GEMS memory timing simulator [38], and GPGPU-Sim v3.2.1 [39] to model the GPU (the GPU architecture is similar to an NVIDIA GTX 480, although we use larger caches to reflect recent trends). The simulator also uses Garnet [40] to model a 4x4 mesh interconnect with a GPU CU or a CPU core at each node. We use CUDA 3.1 [32] for the GPU kernels in the applications since this is the latest version of CUDA that is fully supported in GPGPU-Sim. We modified GPGPU-Sim's atomic operations to provide the appropriate acquire-release, relaxed, and scoped semantics. Table III summarizes the common key parameters of the system.

C. Hardware Configurations

We evaluate the following configurations, using the implementations described in Section II:

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

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

DeNovo-DRF0 (DD0): *DD0* uses DeNovo coherence, which performs all atomic operations at the L1s and coalesces atomics at the L1 MSHRs, and the DRF0 memory model.[5]

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

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

Some of these configurations are only useful for analyzing either the SyncPrims or relaxed atomics microbenchmarks. Thus, in our evaluation we do not use GDR or DDR for the SyncPrims microbenchmarks (since they do not use relaxed atomics) and do not use GH for the relaxed atomics microbenchmarks (since only Flags would benefit from scoped

[4]HRF [12] uses a three-level cache hierarchy; we use two levels because the GEMS simulation environment (Section IV-B) only supports two levels. We believe our results are not qualitatively affected by the depth of the memory hierarchy.

[5]Prior work added a read-only optimization to DD0 to avoid invalidating read-only data on acquires [4] and remove the performance gap between DD0 and GH. To avoid cluttering our graphs, we do not include results for this configuration, but we observed similar trends.

synchronization). We use the remainder of the configurations for all of HeteroSync's microbenchmarks. Since our focus is the GPU, for all configurations, the CPU always uses the DeNovo coherence protocol. We also assume support for performing synchronization accesses (using atomics) at the L1 and L2. Finally, we use self-relative speedups to compare the execution time of the microbenchmarks.

D. Benchmarks Configurations

We examine how HeteroSync's performance scales as the number of CUs vary. Tables I and II summarize the microbenchmarks and the input sizes we use. Due to space limitations, we show results for a single input size for each microbenchmark. All codes use a single CPU core. For all relaxed atomic microbenchmarks, we use strong scaling: as the number of CUs increase, the total amount of work stays the same but is divided across more CUs. For all SyncPrims, we perform 100 iterations of the critical section for two versions described in Section III-A: a locally scoped version that shares data locally on an CU (denoted with "_L") and a globally scoped version that shares data globally (denoted with "_G"). The tree barriers, which use both local and global scope, are denoted with "_LG." We use weak scaling for all SyncPrims: as the number of CUs increase, the amount of work per thread (and per CU) remains constant. Thus, as the numbers of CUs increase contention also increases for the hybrid and globally scoped microbenchmarks; contention stays the same for the locally scoped microbenchmarks because they only compete with other threads on the same CU. Although we would have preferred to use strong scaling, the SyncPrims microbenchmarks was designed to do weak scaling [4]. In addition to the tree barriers, we also examine a version where each CU exchanges data locally before joining the global barrier [4].

V. Results

Respectively, Figures 2, 3, and 4 show how the local/hybrid scoped SyncPrim, globally scoped SyncPrim, and relaxed atomic microbenchmarks in HeteroSync scale as the number of CUs varies. Overall, our results show the benefits to emerging coherence and consistency techniques for heterogeneous systems and how the microbenchmarks are able to pinpoint differences in these approaches. For the locally scoped microbenchmarks, DD0 and GH provide near perfect scaling, due to low contention (recall we analyze weak scaling for these benchmarks). However, GD0 scales poorly due to increased contention. For the hybrid and globally scoped SyncPrims, increasing the number of CUs increases contention and execution time for GH and DD0, but DD0 scales better than GH. The relaxed atomics microbenchmarks show mixed results: for some microbenchmarks relaxed atomics improve (strong) scalability, while for others they increase execution time, only provide small benefits, or are unaffected by scaling the number of CUs.

A. Local/Hybrid SyncPrims

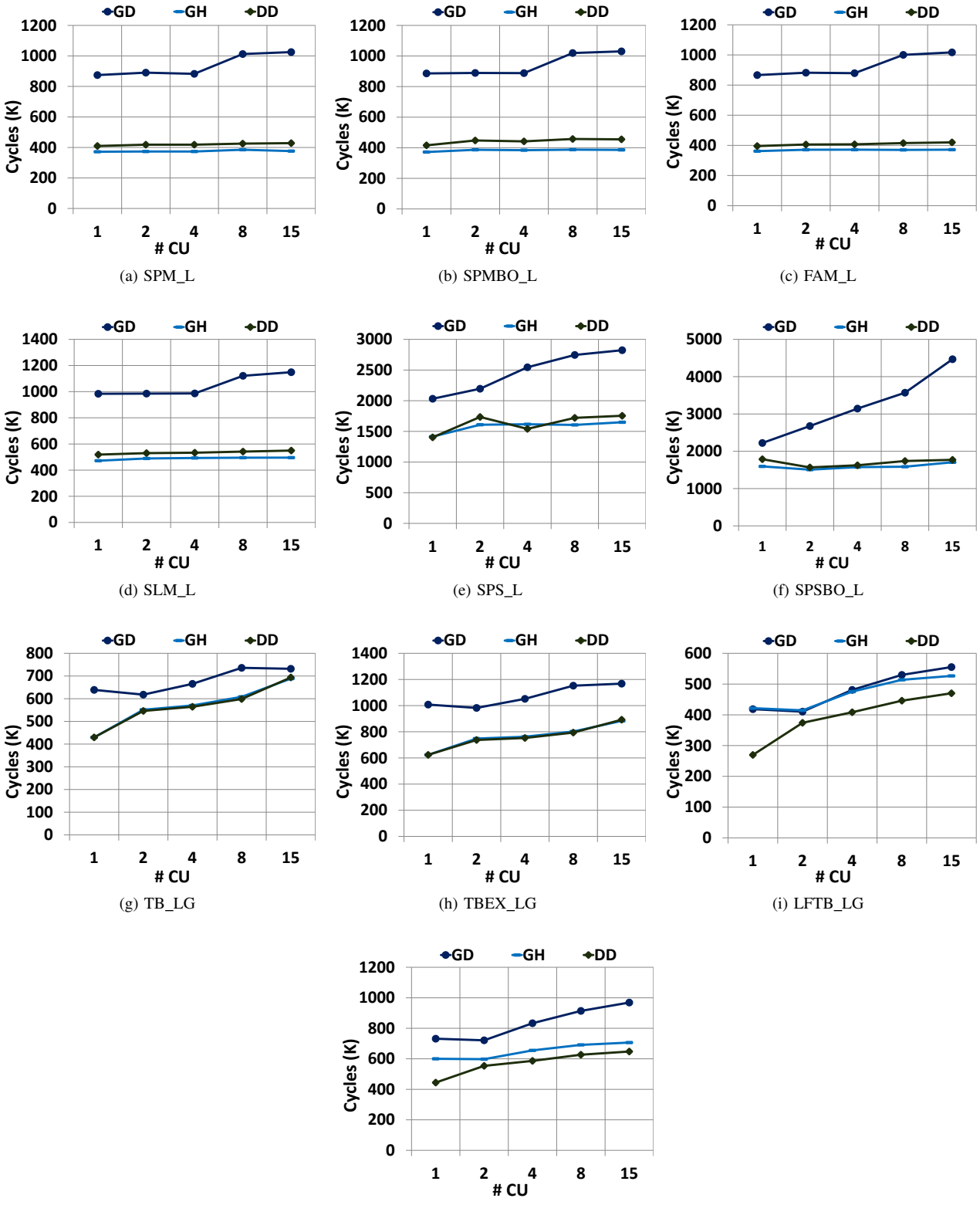
Mutexes: The locally scoped mutex algorithms (Figure 2a-2d) all show similar scaling trends: with weak scaling, GH and DD0 provide near perfect scaling while GD0 suffers from increased contention and scales poorly. GH and DD0 scale well because they can keep the data local and perform the atomics at the L1 cache; GD0 scales poorly because it writes through all dirty data on releases, invalidates all valid data on acquires, and performs atomics at the L2. For all locally scoped mutexes, as the number of CUs increase, GH and DD0 clearly outperform GD0, while GH slightly outperforms DD0. This confirms the findings of prior work [4]. Furthermore, as the number of CUs increases, GD0's execution time increases due to increased contention at the L2 for the atomic variables – a new finding from our scalability study.

As discussed in Section III-A, the locally scoped mutex algorithms have per-CU mutexes. As a result, contention for a given mutex is low.[6] Thus, we see that all of the mutex algorithms obtain similar execution times. The one exception is SLM_L, where the decentralized ticket lock increases execution time (e.g., 24% for DD0 with 15 CUs) over the centralized ticket lock. Thus, decentralizing the ticket lock does not help in cases of low contention.

Semaphores: The locally scoped semaphores (Figure 2e and 2f) have very different scaling trends: GH and DD0 do not have perfect scaling and GD0 scales much worse than it did in the mutex algorithms. GD0 scales poorly because it has even more synchronization accesses than the mutexes – whereas GH and DD0 can access the variables locally. The semaphore's reader-writer format also impacts whether DD0 or GH provides the best performance. When the writer enters the semaphore first (as happens to be the case for SPS_L, 4 CUs), DD0 slightly outperforms GH for SPS_L because it immediately obtains ownership for read-write data and can reuse it across subsequent acquires and releases. However, when the readers enter the semaphore first, DeNovo must invalidate this data; in comparison, GH exploits the scope information to retain the data. Adding backoff (SPSBO_L) reduces DD0's overheads such that GH and DD0 provide similar performance (GH is 9% better than DD0 for 8 CUs, and 4% better for 15 CUs).

Barriers: As the number of CUs increase, the barrier algorithms (Figure 2g-2j) execution time increases for GD0, GH and DD0. Although the local barriers stays the same, more TBs join the global barrier as the number of CUs increase. Nevertheless, DD0 and GH's ability to retain some of the data locally reduces execution time compared to GD0 (e.g., DD0 is 12% better than GD0 for 2 CU's). Moreover, adding local data exchange (TBEX_LG, LFTBEX_LG) further increases

[6] Although contention for a given mutex is low, since GD0 sends all atomics to the L2, L2 contention increases as the number of CUs increase, which hurts GD0's scalability.



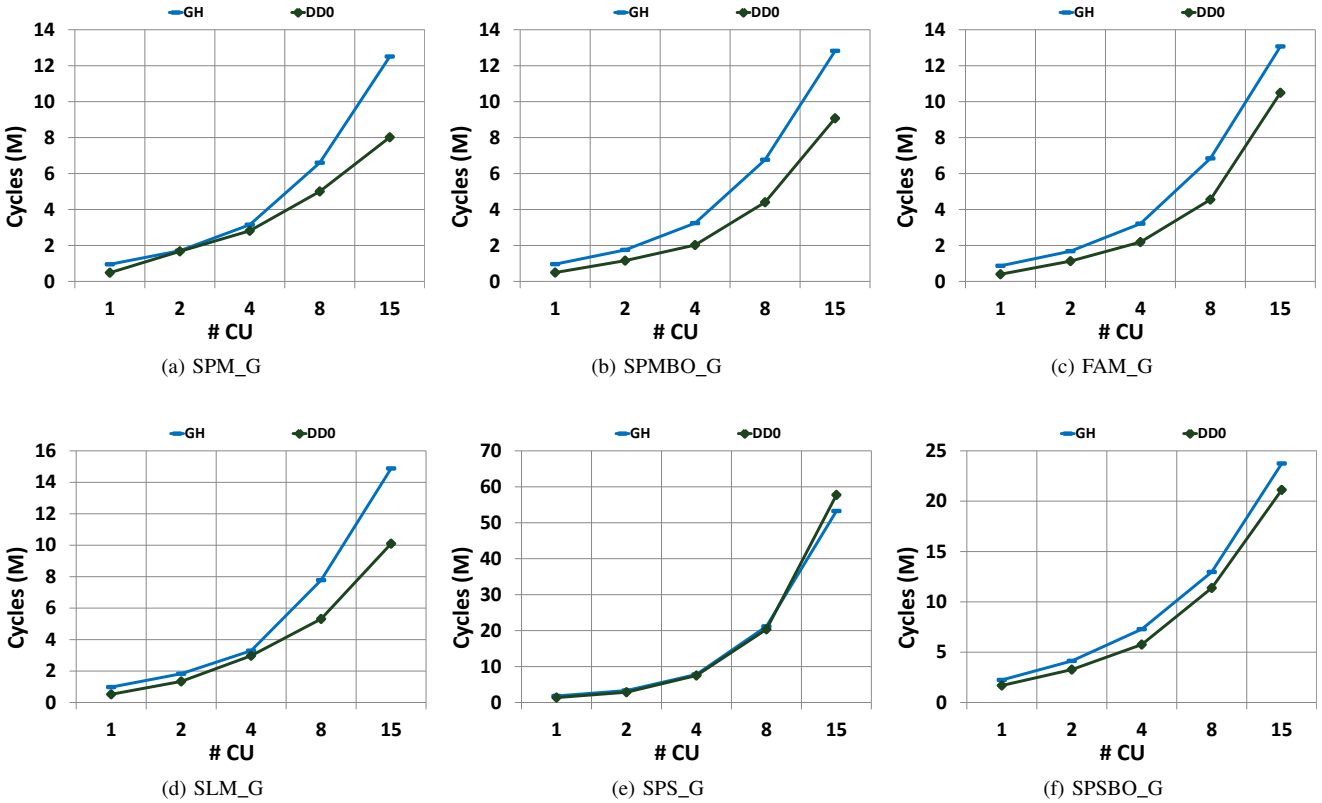
(a) SPM_L (b) SPMBO_L (c) FAM_L

(d) SLM_L (e) SPS_L (f) SPSBO_L

(g) TB_LG (h) TBEX_LG (i) LFTB_LG

(j) LFTBEX_LG

Figure 2: Weak scaling results for local and hybrid scoped synchronization benchmarks.



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

(d) SLM_G (e) SPS_G (f) SPSBO_G

Figure 3: Weak scaling results for globally scoped synchronization benchmarks.

DD0's benefits over GD0, because they can reuse this data. These results confirm findings from previous work that DD0 outperforms GH when hybrid scope is used [4], but also show that the amount DD0 and GH benefit over GD0 varies with the number of CUs. Moreover, as expected, the lock-free tree barrier reduces execution time (e.g., GD0 is 28% faster for 8 CUs) and scales better than the atomic tree barrier.

B. Global SyncPrims

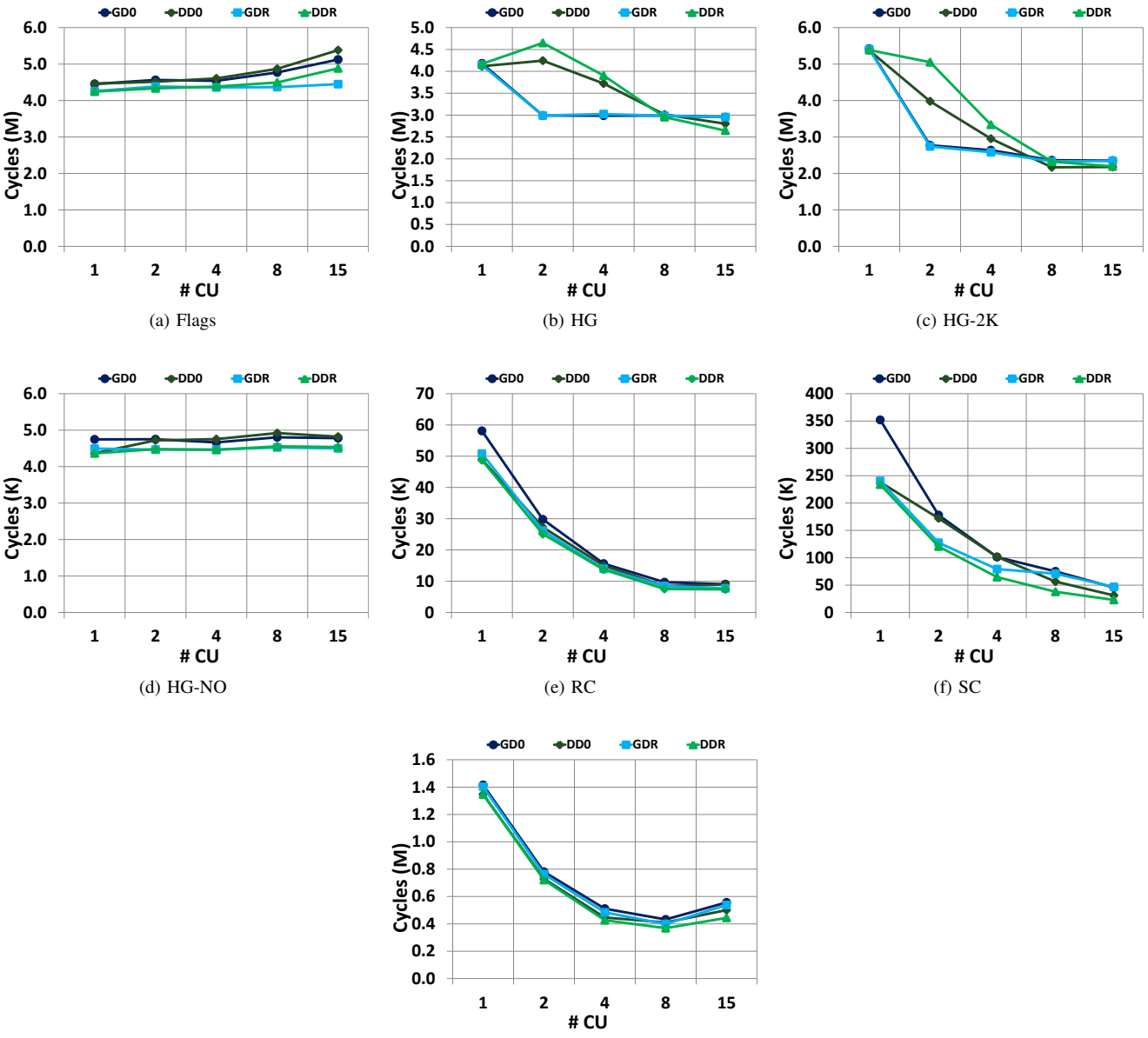
Mutexes: Since GD0 and GH provide the same performance when scope is global, we only show GH in Figure 3. As the number of CUs increase, as expected both GH and DD0 scale poorly. Nevertheless, DD0 always scales better than GH for all globally scoped mutexes (Figure 3a-3d) because it is better able to exploit reuse of written data and atomics, which prior work showed only for 15 CUs [4]. For example, for 8 CUs, DD0 outperforms GH by 24%, 29%, 33%, and 32%, respectively, for SPM_G, SPMBO_G, FAM_G, and SLM_G.

SPMBO_G again does not reduce execution time compared to SPM_G, despite increased contention for the globally shared mutexes. Thus, even though SPMBO_G reduces contention for the shared mutex, it does not reduce execution time. However, for higher levels of contention (e.g., 15 CUs) SLM_G's decentralized ticket lock is more scalable than FAM_G for DD0 (but not GH) because DeNovo allows threads to spin locally in their L1 caches.

Semaphores: Execution time also increases for the globally scoped semaphores (Figure 3e and 3f) as contention increases. For SPS_G, increasing contention hurts DD0 more than GH because more ownership requests must be sent to remote L1s. For 15 CUs, DD0 is 8% worse than GH. Introducing backoff significantly reduces execution time (63% for DD0, 55% for GH with 15 CUs) and DD0 has 11% less execution time than GH.

C. Relaxed Atomics

We use strong scaling for the relaxed atomics microbenchmarks. SC, RC, and SL (Figure 4e-4g) see significant reductions in execution time when the number of CUs increase. This shows that they are able to effectively partition the work across the CUs, especially through 8 CUs. Using relaxed atomics also reduces execution time for SC, RC, and SL, although the gains are sometimes small. In the best case, with 4 CUs, DDR (GDR) reduces execution time by 36% (22%) over DD0 (GD0) for SC. Moreover, DeNovo reduces execution time compared to GPU coherence regardless of the number of CUs used, by reusing written data and atomics. For example, for 15 CUs, DDR reduces execution time by 51%



(a) Flags (b) HG (c) HG-2K

(d) HG-NO (e) RC (f) SC

(g) SL

Figure 4: Strong scaling results for relaxed atomic benchmarks.

over GDR for SC. In general, relaxed atomics improve the scalability of these algorithms, with 2 notable exceptions (SC, for 8 and 15 CUs with GDR and SL, for 15 CUs, for GDR and DDR) where the extra contention caused hurts scalability.

Flags (Figure 4a) execution time scales similarly to LFTB_LG, because the lock-free tree barrier makes up a significant portion of Flags' execution time. Nevertheless, as the number of CUs increases, the benefits from using relaxed atomics increase (for 15 CUs, 9% less execution time for DDR vs. DD0 and GDR vs. GD0) and GPU coherence reduces execution time more than DeNovo because of the overhead of remote L1 ownership requests.

HG-NO (Figure 4d) is relatively unaffected by increasing the number of CUs. This is not surprising, because HG-NO only uses a single TB. Thus, increasing the number of CUs only affects which CU may own the data from the previous phase of the application (HG). Relaxed atomics reduce execution time, but the gains are small ($\leq$ 6% in all cases) and are unaffected by varying the number of CUs.

Increasing the number of CUs shows mixed results for HG and HG-2K (Figure 4b and 4c). For example, for HG, increasing the number of CUs from 1 to 2 increases execution

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

VI. Related Work

There have been many studies on synchronization benchmarks for CPUs (e.g., Synchrobench [41]), but we focus on those for GPUs. Previous work has created suites of GPU microbenchmarks that use various synchronization primitives for discrete GPUs. Several papers have explored the design of concurrent queues [20] and lock-free data structures [21]. Others have benchmarks that use fine-grained locking in a similar manner to the SyncPrims mutex locks [22], [23], but use them to explore transactional memory in discrete GPUs. More recent work has explored optimizing performance for lock-free applications on GPUs through optimized assembly code and rollback for cases where deadlock occurs [24], [25]. None of these papers examine how their algorithms perform on tightly coupled systems with DRF or HRF consistency models though. Chai and Hetero-Mark introduce benchmarks for collaborative CPU-GPU computing, but they focus on applications that do not use fine-grained synchronization [26], [27]. Some notable exceptions are Chai's *Image Histogram - Input Partitioning* and Hetero-Mark's *Color Histogram*, which are similar to H and HG, and Chai's *Padding*, which is similar to Flags.

VII. Conclusion

In this work, we present HeteroSync, a set of GPU microbenchmarks that use various kinds of synchronization, that combines microbenchmarks from several previous papers into a single suite. HeteroSync allow researchers to explore the differences between various fine-grained synchronization algorithms, coherence protocols, and consistency models. Our results demonstrate how HeteroSync can be used to compare algorithm scalability, coherence protocols, and consistency models for heterogeneous systems.

Acknowledgments

This work was supported in part by a Qualcomm Innovation Fellowship for Sinclair, the National Science Foundation under grants CCF 13-02641 and CCF 16-19245, and by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] HSA Foundation, "HSA Platform System Architecture Specification," http://www.hsafoundation.com/?ddownload=4944, 2017.

[2] IntelPR, "Intel Discloses Newest Microarchitecture and 14 Nanometer Manufacturing Process Technical Details," *Intel Newsroom*, 2014.

[3] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs," in *IEEE 20th International Symposium on High Performance Computer Architecture*, 2014.

[4] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, December 2015.

[5] ——, "Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[6] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU concurrency Weak behaviours and programming assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[7] T. Sorensen, J. Alglave, G. Gopalakrishnan, and V. Grover, "ICS: U: Towards Shared Memory Consistency Models for GPUs," in *Towards Shared Memory Consistency Models for GPUs*, 2013, pp. 489–490.

[8] M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling SC Atomics in C11 and OpenCL," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 634–648.

[9] "SPEC ACCEL," https://www.spec.org/accel/, 2017.

[10] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IEEE International Symposium on Workload Characterization*, 2012, pp. 141–151.

[11] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," in *IEEE International Symposium on Workload Characterization*, Sept 2013, pp. 185–195.

[12] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-Race-Free Memory Models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 427–440.

[13] J. Y. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[14] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, "Synchronization Using Remote-Scope Promotion," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[15] J. A. Stuart and J. D. Owens, "Efficient Synchronization Primitives for GPUs," *CoRR*, vol. abs/1110.4623, 2011. [Online]. Available: http://arxiv.org/abs/1110.4623

[16] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *19th International Symposium on High Performance Computer Architecture*, 2013.

[17] J. Alsop, B. Beckmann, M. Orr, and D. Wood, "Lazy Release Consistency for GPUs," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[18] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016.

[19] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *ACM Transactions on Architecture and Code Optimizations*, vol. 12, no. 1, pp. 7:1–7:26, April 2015.

[20] D. Cederman, B. Chatterjee, and P. Tsigas, "Understanding the Performance of Concurrent Data Structures on Graphics Processors," in *Proceeedings of 18th International Euro-Par Conference on Parallel Processing*, 2012, pp. 883–894.

[21] P. Misra and M. Chaudhuri, "Performance Evaluation of Concurrent Lock-free Data Structures on GPUs," in *IEEE 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 53–60.

[22] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO, 2011, pp. 296–307.

[23] W. W. L. Fung and T. M. Aamodt, "Energy Efficient GPU Transactional Memory via Space-time Optimizations," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 408–420.

[24] Y. Xu, L. Gao, R. Wang, Z. Luan, W. Wu, and D. Qian, "Lock-based Synchronization for GPU Architectures," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2016, pp. 205–213.

[25] A. Li, G.-J. van den Braak, H. Corporaal, and A. Kumar, "Fine-Grained Synchronizations and Dataflow Programming on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 109–118.

[26] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. García-Flores, S. G. de Gonzalo, T. B. Jablin, A. J. Peña, and W.-m. Hwu, "Chai: Collaborative Heterogeneous Applications for Integrated-architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS, April 2017, pp. 43–54.

[27] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli, "Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing," in *IEEE International Symposium on Workload Characterization*, ser. IISWC, September 2016, pp. 1–10.

[28] S. V. Adve and H.-J. Boehm, "Memory Models: A Case for Rethinking Parallel Languages and Hardware," *Communications of the ACM*, pp. 90–101, August 2010.

[29] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 68–78.

[30] H. Sutter, "Atomic Weapons: The C++ Memory Model and Modern Hardware," in *C++ and Beyond*, 2012.

[31] T. Sorensen and A. F. Donaldson, "Exposing Errors Related to Weak Memory in GPU Applications," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 100–113.

[32] NVIDIA, "CUDA SDK 3.1," http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[33] M. D. Sinclair, "Efficient Coherence and Consistency for Specialized Memory Hierarchies," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2017.

[34] V. Podlozhnyuk, "Histogram calculation in CUDA," http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf, 2007.

[35] H.-J. Boehm, "Can Seqlocks Get Along with Programming Language Memory Models?" in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2012.

[36] P. McKenney, "Some Examples of Kernel-Hacker Informal Correctness Reasoning," in *Proceedings of the Dagstuhl Workshop on Compositional Verification Methods for Next-Generation Concurrency*, 2015.

[37] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, P. Srivastava, M. Kotsifakou, S. V. Adve, and V. S. Adve, "Stash: Have Your Scratchpad and Cache it Too," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 707–719.

[38] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, 2005.

[39] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[40] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[41] V. Gramoli, "More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP, 2015, pp. 1–10.