Lazy Release Consistency for GPUs

Johnathan Alsop[†] Marc S. Orr[‡§] Bradford M. Beckmann[§] David A. Wood[‡§]

[†]University of Illinois at Urbana–Champaign
alsop2@illinois.edu

[‡]University of Wisconsin–Madison
{morr,david}@cs.wisc.edu

[§]AMD Research
Brad.Beckmann@amd.com

***Abstract*—The heterogeneous-race-free (HRF) memory model has been embraced by the Heterogeneous System Architecture (HSA) Foundation and OpenCL™ because it clearly and precisely defines the behavior of current GPUs. However, compared to the simpler SC for DRF memory model, HRF has two shortcomings. The first is that HRF requires programmers to label atomic memory operations with the correct scope of synchronization. This explicit labeling can save significant coherence overhead when synchronization is local, but it is tedious and error-prone. The second shortcoming is that HRF restricts important dynamic data sharing patterns like work stealing. Prior work on remote scope promotion (RSP) attempted to resolve the second shortcoming. However, RSP further complicates the memory model and no scalable implementation of RSP has been proposed. For example, we found that the previously proposed RSP implementation actually results in slowdowns of up to 30% on large GPUs, compared to a naïve baseline system that forgoes work stealing and scopes. Meanwhile, DeNovo has been shown to offer efficient synchronization with an SC for DRF memory model, performing on average 21% better than our baseline system, but it introduces additional coherence traffic to maintain ownership of all modified data.**

To resolve these deficiencies, we propose to adapt lazy release consistency—previously only proposed for homogeneous CPU systems—to a heterogeneous system. Our approach, called hLRC, uses a DeNovo-like mechanism to track ownership of synchronization variables, lazily performing coherence actions only when a synchronization variable changes locations. hLRC allows GPU programmers to use the simpler SC for DRF memory model without tracking ownership for all modified data. Our evaluation shows that lazy release consistency provides robust performance improvement across a set of graph analysis applications—29% on average versus the baseline system.

Keywords—graphics processing unit (GPU); memory model; lazy release consistency; scope promotion; scoped synchronization; work stealing

I. Introduction

Architects must carefully consider a plethora of tradeoffs when specifying a new memory model and designing the hardware that implements it. With the emergence of heterogeneous computing and high-throughput accelerators, there is an increasing tension to keep both the memory model and hardware simple. In comparison, CPUs provide relatively simple memory models, but use complex and highly optimized cache coherence protocols that enforce the single-writer/multiple reader invariant [1]. Specifically, store operations invalidate the target address at every private cache other than the initiator's. This complicated CPU approach is a poor fit for GPUs for several reasons. First, a GPU core, called a compute unit (CU), has thousands of hardware threads, called work-items. Sending invalidations on every store miss would generate far too much invalidation traffic. Second, managing the invalidations requires sophisticated cache controllers that detract from the GPU's primary application: graphics. Finally, writer-initiated invalidations often employ inclusive caches, which are a poor fit for GPUs because their aggregate L1 cache capacity approaches the size of a typical GPU last-level cache.

For these reasons, GPUs take a different approach to synchronization. Specifically, they use simple bulk coherence actions, like cache flushes and invalidates, at the synchronization points in the program. This approach aligns with current memory models, like C++11 [2], where programmers clearly identify inter-thread communication by operating on atomic variables. At these synchronization points, coarse-grain coherence actions, like cache flushes and invalidates, are sufficient to implement memory models that guarantee sequential consistency for data race-free (SC for DRF) programs [3].

Unfortunately, bulk coherence actions negatively affect performance. Specifically, cache flushes incur long latencies because they require all of the dirty cache blocks in the initiator's private caches to be written through the memory hierarchy. Flash invalidations are fast, but degrade cache locality and cause excessive cache misses.

To solve these problems, modern GPUs support scoped synchronization [3][4][5]. Scopes takes advantage of the GPU's hierarchical execution model to limit the cost of bulk coherence actions. For example, work-items executing on the same CU can communicate through the L1 cache without incurring any cache flushes or invalidates. In contrast, work-items executing on different CUs are required to read and write from the GPU's monolithic last-level cache.

978-1-5090-3508-3/16/$31.00 ©2016 IEEE

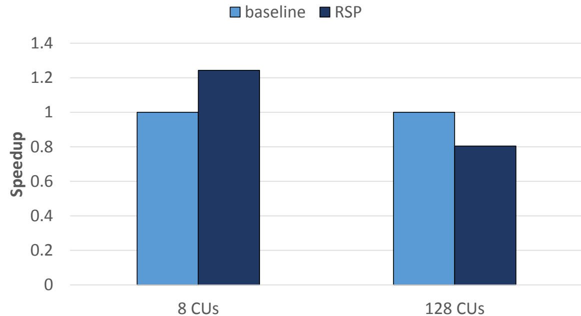

Figure 1. RSP scalability on a small and large GPU.

While scoped synchronization is successful in mitigating the cost of bulk coherence actions, it leads to a memory model (e.g., SC for HRF [6][7][8]) with two significant shortcomings. First, programmers are expected to explicitly label atomic memory operations with the correct scope in order to maximize performance, which is tedious and error-prone. Second, scoped synchronization does not use caches effectively for important dynamic data sharing patterns like work stealing.

To combat this second shortcoming, remote scope promotion [10] was recently proposed, but it is not a panacea. RSP further complicates the memory model and the initial implementations of RSP, while effective for relatively small GPUs, do not scale to large GPUs. Specifically, we found that RSP actually performs worse on a large 128-CU GPU when compared to a naïve baseline that forgoes work stealing and scopes (Figure 1).

Meanwhile, to combat the first shortcoming, the recent DeNovo proposal suggested that future GPUs should forgo scoped synchronization and support the simpler SC for DRF memory model [8]. However, DeNovo tracks ownership for all written data, requiring additional traffic to request and revoke ownership registration. Also, when compared to current GPU designs, DeNovo's benefits primarily arise from locality in written data, which is limited in existing GPU compute applications.

In this work, we introduce heterogeneous lazy release consistency (hLRC) for GPUs. Like DeNovo, our approach eliminates scopes and enables SC for DRF on GPUs, achieving scalable synchronization for data sharing patterns like work stealing. hLRC also uses atomic registration, as proposed by Sung and Adve [9], to track exclusive ownership of synchronization variables, but not all of stored data like DeNovo. hLRC also differs from DeNovo by performing coherence actions when synchronization variables change registration, thus implementing lazy releases and potentially reducing coherence traffic. hLRC achieves a speedup of on average 29% on a large GPU with 128 CUs, when compared to the naïve baseline, and 7% on average compared to DeNovo. Finally, our implementation of hLRC builds off of bulk synchronization flush and invalidate actions, which is consistent with the current approach to GPU synchronization.

Table 1. Simple GPU coherence actions.

Flush local L1	Coarse-grain flush of all dirty data in the local **L1** to the next level of the memory hierarchy.
Inv local L1	Coarse-grain invalidation of all valid data in the local **L1**.
LD/ST/RMW x L1/L2	Atomic memory access on location **x** performed at the **L1** or **L2** cache
Lock op/x	Block a specific operation (**op**) at a particular cache or all ops on address **x** within a cache.

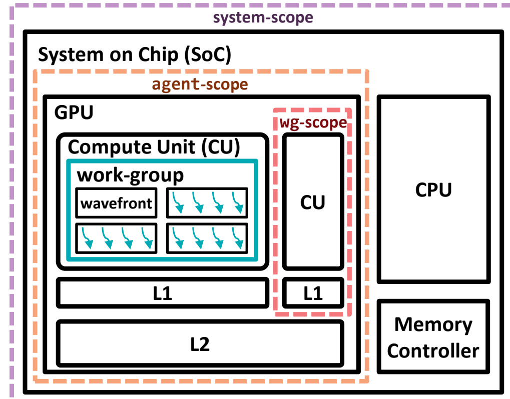

Figure 2. Baseline example GPU.

II. GPU Caches and Synchronization

A. GPU Architecture

The GPU's massively threaded architecture, depicted in Figure 2, targets highly concurrent applications. Specifically, each GPU core, called a compute unit (CU), executes thousands of threads, called work-items, simultaneously. For example, a CU in the AMD GCN architecture has hardware state for 2,560 work-items [11]. The GPU's CUs are connected to memory through a hierarchy of caches. Typically, each CU has a private L1 cache to optimize communication within a CU. The L1 caches tend to be small and optimize for throughput. For example, the L1 cache is 16 kB in AMD's GCN architecture [11] and up to 48 kB in Nvidia's Maxwell GPU [4]. To optimize communication between work-items on different CUs, it is common to connect the L1 caches to a GPU-wide non-inclusive L2 cache.

GPU work-items (`wi`) execute within an execution hierarchy that mirrors the GPU's hierarchical design. The first level of the execution hierarchy is a wavefront, which is a small group of work-items (e.g., 64 on AMD GPUs, 32 on NVIDIA GPUs, 4 on Intel GPUs, etc.) that execute in lockstep on the GPU's data-parallel execution units. Wavefronts then execute in small teams called work-groups. Wavefronts in the same work-group execute on the same CU, which enables them to synchronize through the L1 cache. Ultimately, a GPU executes a grid of work-groups. Thus, work-items in a grid can communicate through the GPU's L2. Finally, work-items in a grid can communicate externally (e.g., with CPU threads) through a common level of the memory hierarchy (e.g., the memory controller).

B. GPU Synchronization

Recall that each CU executes thousands of work-items concurrently. Thus, to avoid excessive invalidation traffic,

```
CU0            CU1
<guarded>
ST_rel x
               LD_acq x
               <guarded>
```
a. SC for DRF (no scopes)

```
CU0 (wi 0)   CU0 (wi 1)
<guarded>
ST_rel_wg x
             LD_acq_wg x
             <guarded>
```
b. SC for HRF (work-group scope)

```
CU0            CU1
<guarded>
ST_rel_agt x
               LD_acq_agt x
               <guarded>
```
c. SC for HRF (agent scope)

```
CU0            CU1
<guarded>
ST_rel_wg x
               LD_acq_rm_agt x
               <guarded>
```
d. RSP (remote agent scope)

Figure 3. Release to acquire synchronization in different memory models.

GPUs forgo CPU-like coherence protocols where each write obtains ownership to enforce the single-writer/multiple reader invariant [1]. Instead, GPUs only track whether a cache line is valid (V), dirty (D), or invalid (I) and rely on simple coherence actions, summarized in Table 1, at synchronization points (e.g., when accessing an atomic variable like a lock).

Coherence actions enable sequential consistency for data-race-free programs (SC for DRF) [3]. The C++11 memory model [12] provides atomic variables for communicating between threads. For example, consider the sequence of operations in Figure 3a, where a work-item on CU0 executes a store release (`ST_rel`) operation on the atomic variable `x` to broadcast data that it has written to other work-items on the GPU. A memory operation labeled as release naturally corresponds to the flush action (Table 1), which writes dirty data through the memory hierarchy. Referring back to Figure 3a, a work-item on CU1 then executes a load acquire memory operation (`LD_acq`) to read data written by another work-item. An acquire corresponds to the invalidate action in Table 1, which eliminates cached data that may have become stale.

C. Scoped Synchronization

Since GPUs use expensive coarse-grain coherence actions, current programmers are encouraged to use scoped synchronization to minimize their performance impact. In particular, cache flushes encounter long latency while waiting to write all dirty data through the cache hierarchy, and flash invalidates, while fast, significantly degrade cache locality. Scoped synchronization allows programmers to identify when these coarse-grain operations are necessary and conveniently matches the GPU's execution hierarchy. Currently, the most important scopes are work-group (`wg`), GPU-wide (`agent`), and SoC-wide (`system`). For example, consider Figure 3b where flushes and invalidates are avoided entirely because the atomic memory operations are labeled as `wg`-scoped, which is sufficient when work-items from the same work-group communicate. In contrast, when instructions are labeled as `agent`-scoped (Figure 3c), L1 cache flushes and invalidates are used to communicate data through the GPU-wide L2 cache and the atomic memory operations are directly performed at the L2 (denoted in row 3 of Table 1). Finally, while agent scope is sufficient when work-items from the same grid communicate, system scope is used to communicate between CPU and GPU threads.

Scoped synchronization has two major shortcomings. The first shortcoming is that it leads to a more complex memory model called SC for HRF, where atomic accesses must be statically labeled to indicate the scope of communication. If a programmer labels atomic memory operations with the wrong scope, a race can occur. A second problem with scopes is that it is difficult to optimize dynamic sharing patterns like work stealing.

D. Remote Scope Promotion

An inherent limitation of scoped synchronization is its inability to effectively utilize caches for dynamic sharing patterns like work stealing. For example, consider an application that allocates a task queue per work-group. In the common case, work-items within a work-group would like to coordinate their task queue accesses using `wg`-scoped atomic operations (e.g., Figure 3b). However, recall that work-items in different work-groups are required to synchronize through the agent scope (e.g., Figure 3c). Thus, accessing any queue with `wg`-scoped atomics disallows a work-item to steal a task from another work-group's task queue.

To solve this dilemma, RSP gives work-items the capability to dynamically promote the scope of atomic memory operations executed by other work-items. This is demonstrated in Figure 3d, where a work-item executing on CU1 promotes the `wg`-scoped store release operation to the GPU-wide agent scope.

While RSP resolves the limitation of using static scoped synchronization, the initial implementation of RSP relies on heavyweight broadcast operations and cache-level locks to preserve RMW atomicity. Table 2 describes these operations. To illustrate these overheads, we step through the required coherence actions for the example in Figure 3d. We focus on the implementation described (and verified) by Wickerson et al. [13], which is enumerated on the left side of Table 3.

In the example, the `wg`-scoped store release is performed locally in CU0's L1 cache (a_6) and does not trigger any coherence actions. To correctly enforce acquire semantics, the `rm_agent`-scoped RSP access needs to promote the scope of the last `wg`-scoped release. In the example, after performing a load on the atomic variable at the L2 cache (a_3), dirty data on CU0 must be written through to the GPU-wide L2 cache, since the `wg`-scoped release does not cause a flush. Since the RSP access does not know the location of past synchronizing releases, it must broadcast a flush command to all L1 caches

Table 2. Coherence actions added by RSP from Wickerson et al. [13].

Flush all L1s bcast	A *broadcasted* request to all remote L1 caches to flush their dirty data to the next level of the cache hierarchy.
Inv all L1s bcast	A *broadcasted* request to all remote L1 caches to invalidate their valid data.
Lock all RMWs	A *broadcasted* request to all remote L1 caches to block RMWs.
Unlock all RMWs	A *broadcasted* request to all remote L1 caches to unblock RMWs.

Table 3. Coherence actions for implementing the RSP (scoped) memory model and the DeNovo and hLRC (non-scoped) memory models.

Instruction	Scoped Memory Models			Non-scoped Memory Models			
	Order	***Scope***	***RSP Actions***	***Order***	***Prior Location of Registered Atomic***	***DeNovo Actions***	***hLRC Actions***
Atomic LD	Acquire	Work-group	a_0: LD x L1	Acquire	Local L1	b_0: LD x L1 b_1: Inv local L1	c_0: LD x L1
		Agent	a_1: LD x L2 a_2: Inv local L1		L2	b_2:R state & data to requesting L1 b_3: LD x L1 b_4: Inv local L1	c_1: R state & data to requesting L1 c_2: LD x L1 c_3: Inv local L1
	Remote Acquire	Agent	a_3: LD x L2 a_4: Flush all L1s bcast a_5: Inv local L1		Remote L1	b_5:R state & data to requesting L1 b_6: LD x L1 b_7: Inv local L1	c_4: Flush remote L1 c_5: R state & data to requesting L1 c_6: LD x L1 c_7: Inv local L1
Atomic ST	Release	Work-group	a_6: ST x L1	Release	Local L1	b_8: StReg local L1 b_9: ST x L1	c_8: ST x L1
		Agent	a_7: Flush local L1 a_8: ST x L2		L2	b_{10}: StReg local L1 b_{11}:R state & data to requesting L1 b_{12}: ST x L1	c_9: R state & data to requesting L1 c_{10}: ST x L1 c_{11}: Inv local L1
	Remote Release	Agent	a_9: LK all RMWs a_{10}: Flush all L1s bcast a_{11}: Inv all L1s bcast a_{12}: ST x L2 a_{13}: Inv all L1s bcast a_{14}: UL all RMWs		Remote L1	b_{13}: StReg local L1 b_{14}:R state & data to requesting L1 b_{15}: ST x L1	c_{12}: Flush remote L1 c_{13}: R state & data to requesting L1 c_{14}: ST x L1 c_{15}: Inv local L1
Atomic RMW	Acquire-Release	Work-group	a_{15}: RMW x L1	Acquire-Release	Local L1	b_{16}: StReg local L1 b_{17}: RMW x L1 b_{18}: Inv local L1	c_{16}: ST x L1
		Agent	a_{16}: Flush local L1 a_{17}: RMW x L2 a_{18}: Inv local L1		L2	b_{19}: StReg local L1 b_{20}:R state & data to requesting L1 b_{21}: RMW x L1 b_{22}: Inv local L1	c_{17}: R state & data to requesting L1 c_{18}: ST x L1 c_{19}: Inv local L1
	Remote Acquire-Release	Agent	a_{19}: LK all RMWs a_{20}: Flush all L1s bcast a_{21}: Inv all L1s bcast a_{22}: RMW x L2 a_{23}: Flush all L1s bcast a_{24}: Inv all L1s bcast a_{25}: UL all RMWs		Remote L1	b_{23}: StReg local L1 b_{24}:R state & data to requesting L1 b_{25}: RMW x L1 b_{26}: Inv local L1	c_{20}: Flush remote L1 c_{21}: R state & data to requesting L1 c_{22}: RMW x L1 c_{23}: Inv local L1

in the system (a_4), conservatively expanding the scope of all past release accesses to agent. When the heavy-weight broadcast flush is complete, a local cache invalidation is triggered at CU1's L1 (a_5) to make any flushed writes visible at CU1.

A broadcast flush command can add significant overhead to an RSP synchronization access because all dirty data in all remote CUs must be flushed before the RSP access is complete. Performing a flush at all CUs also reduces the write combining potential of all L1 caches in the system and increases traffic in the network.

The example focused on synchronizing a `wg`-scoped release to a `rm_agent`-scoped acquire. A similar approach is used to synchronize a `rm_agent`-scoped release (a_9-a_{14}) to a `wg`-scoped acquire. Specifically, because the `rm_agent`-scoped release does not know the origin of the next acquire, it must broadcast an invalidation command to all L1 caches in the system, conservatively expanding the scope of all future acquires to the agent scope.

Any RSP operation that involves a write (e.g., store, a_9–a_{14}, or RMW, a_{19}–a_{25}) must also enforce RMW atomicity with concurrent `wg`-scoped RMW accesses (a_{15}) to preserve a consistent final state of memory [13]. Specifically, an RSP store blocks RMW operations at all CUs using a broadcast lock command (a_9). It then ensures all caches have the target variable in a consistent state using broadcast flush (a_{10}) and invalidate commands (note a_{11} and a_{13} are both necessary to avoid all races). Finally, it broadcasts an unlock command, allowing CUs to resume processing local RMW requests (a_{14}). Broadcast lock and unlock commands add overhead to the RSP atomic writes, increase network traffic, and delay `wg`-scoped RMW accesses on all CUs for the duration of the RSP access.

In the end, RSP enables dynamic sharing by adding a new synchronization order. However, the proposed implementation of RSP triggers broadcast operations for every remote-scoped access, which complicates the memory system and significantly limits scalability.

E. DeNovo

Sinclair et al. recently proposed applying the DeNovo coherence protocol to GPUs as a means to achieve efficient synchronization without the need for scopes [8]. The column labeled *DeNovo Actions* in Table 3 describes the implementation and highlights that rather than using scopes to avoid coherence actions, DeNovo uses exclusive registration to reduce the impact of synchronization. Specifically, a cache must have registered ownership for all written data by the time it reaches a release point, and immediately for each atomic access. We refer to this store registration action as `StReg` in Table 3 (b_8, b_{10}, b_{13}, b_{16}, b_{19}, and b_{23}). The action is conceptually similar to the `Flush` action described in Table 2. However, instead of writing through dirty data, DeNovo requests registration for each dirty address from the L2 cache and the dirty data remains in the local L1 cache. As long as data obtains exclusive registration, it does not need to be flushed or invalidated on a subsequent acquire or release operation.

DeNovo implements exclusive registration by adding a Registered state to the L1 and L2 cache. If data is in the Registered state at an L1 cache, then that cache has the only registered, up-to-date copy of the variable in the system. When in the Registered state, the L2 cache tracks the ID of the current registered L1 cache using its empty L2 data entry, avoiding the need for a separate pointer storage structure. This requires L2 inclusivity for registered data.

When registration is transferred (either to the L2 on an L1 eviction or to a new requesting L1 cache), the previously registered cache forwards the up-to-date value to the newly registered cache. If the registration is to another L1, the L2 cache updates its registered ID. DeNovo registration thus guarantees that there is always one up-to-date location for written data, and its location can be determined by querying the L2 cache.

By obtaining local registration for written data and atomic accesses, DeNovo is able to exploit locality even in the presence of frequent synchronization. Specifically, only non-registered data needs to be invalidated or flushed on a synchronizing atomic, and the actual atomic update is performed locally. However unlike RSP, every synchronizing atomic triggers an invalidation or flush action. As a result, DeNovo can perform wasteful coherence actions when synchronization is local, but this also allows DeNovo to exploit data locality even when synchronization locality is absent. Additionally, DeNovo's registration of all written data can incur significant overhead, since the L2 must be kept inclusive for this data and an additional level of indirection is required when remotely owned data is requested.

III. Lazy Release Consistency for GPUs

We propose heterogeneous lazy release consistency (hLRC) as a new GPU implementation to efficiently support dynamic sharing and the SC for DRF memory model. hLRC is based on the principles of lazy release consistency, which has previously been used to reduce wasteful communication in distributed CPU shared memory systems [14]. Similar to DeNovo (and unlike RSP), hLRC offers efficient local synchronization and scalable global synchronization without the need for scopes, thus enabling an SC for DRF memory model. In addition, similar to RSP (and unlike DeNovo), hLRC entirely avoids coherence actions when synchronization is local.

Similar to the lazy release consistency for CPUs, hLRC associates each atomic variable with the location it was last accessed. Coherence actions are then performed only when the location of the atomic variable changes (including when the variable is first brought into a cache), because this indicates a possible inter-core synchronization. In doing so, the caches are able to exploit greater efficiency, and heavy-weight coherence actions only need to occur in a targeted manner on the one or two CUs that may be involved in remote synchronization.

A. hLRC Atomic Tracking

In order to trigger the appropriate coherence actions when an atomic variable changes location, hLRC must track and serialize updates to each atomic variable. This is accomplished by obtaining exclusive local registration for every atomic access. The registration mechanism used for hLRC is based on DeNovo registration, described in Section II.E. As with DeNovo, a variable may be registered in only one location at any time, and once obtained, registration is not revoked until a remote CU requests the registration or until the data is evicted.

While DeNovo uses registration for both atomic accesses and normal stores, a key distinction of hLRC is that only atomic accesses require registration. As a result, DeNovo experiences greater L2 cache pressure because every write requires registration at the L2. By only registering atomics, hLRC significantly reduces the amount of registered data that must be tracked at the L2, effectively increasing L2 capacity. In addition, hLRC significantly reduces the additional traffic, probe bandwidth, and latency incurred by requests for remotely registered data.

B. Implementing Synchronization Semantics

hLRC implements release consistency in a scalable fashion appropriate for GPUs and uses atomic registration to automatically detect and exploit synchronization locality without relying on scopes. By delaying coherence actions until the location of a synchronization variable changes, locality for all data (not just written data) is improved, coherence actions are more targeted, and unnecessary data communication is reduced.

Just as traditional lazy release consistency decouples the coherence actions from a release operation and performs the

release only when a synchronizing acquire is detected, hLRC only performs the necessary actions when potential inter-core synchronization is detected: when the location of an atomic variable changes. This section describes how hLRC implements the relevant release consistency semantics. Our implementation is specified completely in the right side of Table 3, which is laid out to show how registration replaces scopes. The coherence actions required by any atomic operation are dependent on the prior registered location of the targeted variable.

1) Acquire Semantic:

With hLRC, an acquire memory operation does not require knowledge of the scope of the last release. Instead, the location of the last release is determined through hardware registration. If the target variable is not already registered at the requesting L1, registration must be obtained. There are two registration scenarios: (1) the atomic variable is resident at the L2 cache or deeper; (2) the atomic variable is registered at another CU's L1 cache. In the latter case, registration is revoked and the last (remote) L1 cache holding the registered data is required to flush its dirty data to the GPU-wide L2 (Table 3, c_4). Once the atomic is unregistered from the last owner, the registration process can proceed (Table 3, c_1-c_3 or c_5-c_7). Transferring registration to an L1 cache always triggers an L1 invalidate (Table 3, c_3, c_7) following the data access. As a result, future acquires to that atomic variable require no coherence actions (Table 3, c_0) as long as the variable remains registered in the L1. To summarize, invalidations are limited to changes in registration (for synchronization variable only). Acquires to local variables avoid the bulk invalidate actions used by DeNovo, and all acquires avoid the costly broadcast invalidate actions used by the RSP implementation.

2) Release Semantic

The operation of a release (Table 3, c_8;c_9-c_{11};c_{12}-c_{15}) mirrors the operation of an acquire. Notably, no coherence actions occur when a release finds the atomic registered at the L1 (Table 3, c_8). This approach delays the flush associated with a release until a registration change. Registration can change in two ways: (1) an atomic access on another CU revokes registration (Table 3, c_4-c_7); (2) the atomic variable is evicted from the L1 cache. When the remote CU loses registration, its L1 cache is flushed (Table 3, c_{12}) to propagate dirty data associated with the atomic and the atomic variable may not be read by any core until the flush is complete. A second subtlety in obtaining registration for a release access is that the local CU executing the release, requires an L1 cache invalidation (Table 3, c_{11};c_{15}). This is to ensure that any subsequent acquire operations to atomic variables located on the same cache line synchronize correctly.

In summary, flushes only occur when an L1 loses registration. Therefore local releases are able avoid most of the coherence actions incurred by DeNovo, and the costly broadcasted flushes used by RSP are eliminated. By delaying the coherence actions associated with a release, hLRC improves store coalescing and reduces write-through traffic when synchronization locality is high.

3) RMW Atomicity

In hLRC, registration acts as a token for exclusive permission, naturally preserving RMW atomicity. Therefore, unlike RSP, atomic memory operations do not require global RMW locks to prevent racing local RMW accesses from generating an inconsistent state.

C. Discussion

Tracking atomic variables enables targeted and efficient coherence actions for most types of synchronization, however there are complexities and costs associated with hLRC. In this subsection, we discuss three unique issues associated with hLRC.

1) Multi-word Cache Block Issues

False sharing occurs when two synchronization variables lie in the same cache block. If these variables are accessed regularly at different CUs, then hLRC registration transfers may be frequent. Every time registration is transferred, coherence actions are triggered to enforce data consistency between the old CU and new CU even though there is no synchronization between them.

In addition to false sharing between atomic variables, hLRC may degrade the performance of non-atomic data accesses to a registered cache line. When a cache line is registered in an L1, it is unaffected by flush or invalidation coherence actions. However, if there are non-atomic data variables on the same line, then preventing flushes and invalidations of this data could cause the cached data to become inconsistent. Therefore data loads and stores to a registered cache line must bypass the L1 cache and be performed at the L2 cache. Servicing data accesses at the L2 to registered cache lines is possible because under SC for DRF the data access may not conflict with the registered portion of the L2 data field.

Code should be optimized to avoid the problems arising from multiple atomic variables on the same cache line or atomic and data variables on the same cache line. Specifically, atomic data may be padded where feasible to prevent colocation in the same cache line with unrelated atomics, or with frequently accessed data.

2) Other Unnecessary Coherence Actions

A second problem is that unnecessary coherence actions can be triggered when registration transfer occurs in the absence of a synchronizing acquire-release pair. For example, frequent registration transfers can occur if there is high contention for an atomic variable (e.g., a global lock). In such a scenario, each synchronization access from a different CU will trigger a serialized store buffer flush, a registration transfer, and a cache invalidation on the critical path of the requesting thread, even if that thread is one of many spinning waiting for the lock. These disadvantages can be mitigated by using synchronization scopes as a performance optimization. This is discussed further in Section V.C.

In addition, cold misses, cache evictions, and false sharing cause registration transfer as well. A cold miss on a

Table 4. Simulation configuration.

128 Compute Units, each configured as described below:	
Clock	1GHz, 4 SIMD units
Wavefronts (#/scheduler)	40 (each 64 lanes)/oldest-job first
Data cache	16kB, 64B line, 16-way, 4 cycles, delivers one line every cycle
Memory Hierarchy	
L2 cache	4MB, 64B line, 16-way, 24 cycles write-through (write-back for R data)
1 Instr. cache/4 CUs	32kB, 64B line, 8-way, 4 cycles
DRAM	DDR3, 32 Channels, 500 MHz
Task Runtime	
128 task queues	1 work-group/queue, 2 wavefronts/work-group
Additional DeNovo/hLRC Storage	
L1 / L2 cache state	R state / R state

synchronization variable clearly does not require any coherence actions. However, since the system cannot differentiate a cold miss from data that has been evicted, an invalidation must be triggered at the requesting L1.

Eviction of registered L1 data causes a flush of dirty data at that L1. Furthermore, since registered data must be kept inclusive in the L2, an eviction of a registered synchronization variable at the L2 must also trigger an eviction and flush in the L1. As a result if the variable is accessed next by the same CU, the miss will trigger an additional unnecessary cache invalidation.

To minimize the overhead associated with unnecessary registration actions, we optimized the L1 and L2 cache replacement policy to prefer non-registered data for eviction first and then by last use (older data is preferred for eviction).

3) *Supporting fences and relaxed atomics*

hLRC has been designed so far to support the simpler SC for DRF memory model, but most modern programming languages, such as C++ [15], provide relaxed atomics and fences that violate SC for DRF. While not explored in detail, we believe hLRC can implement these operations in a straightforward manner. Specifically, relaxed atomics could avoid coherence actions and operate at the L1 cache on an L1 hit, at the L2 cache on an L2 hit, or in a remote cache if remotely registered. Meanwhile, fence operations can translate to acquire-release operations without a memory access (or to a dummy variable).

IV. Methodology

We simulate a CPU-GPU system using an extended version of the publicly available AMD gem5 APU simulator [16]. Figure 2 represents the high-level organization of the evaluated GPU that contains 128 CUs in total. Each CU has four SIMD units with 40 hardware wavefront contexts that are scheduled using the oldest-job-first policy. Each CU has a private L1 data cache. Each instruction cache is shared by four CUs. All L1 data caches and instruction caches are connected to a unified L2 cache that is then connected to system memory through a memory controller shared by an on-chip CPU. These components are distributed evenly across an 8x16 mesh network. Table 4 describes the detailed parameters of the simulated system.

Table 5. Workloads and inputs.

Benchmarks	Graph Inputs	Graph Sizes
Single Source Shortest Path (SSSP)	1: USA-road-d.BAY 2: USA-road-d.COL 3: c-68	7.86 MB 21.3 MB 4.27 MB
Graph Coloring (color)	1: ecology1 2: coAuthorsDBLP 3: dictionary28	35.9 MB 18 MB 1.64 MB
PageRank (PR)	1: USA-road-d.BAY 2: c-68 3: OPF_10000	7.86 MB 4.27 MB 3.57 MB

The baseline protocol uses a write-through, write-allocate policy at the L1 and L2 for all data. An acquire operation triggers a single-cycle flash invalidation of the L1 cache, and a release operation triggers a flush of the L1 store buffer, which is implemented as a FIFO.

To support DeNovo and hLRC we added a Registered state to the L1 and L2 caches. When using hLRC, or DeNovo L1 and L2 caches are write-back and write-allocate for registered data. When registration is transferred to an L1, an invalidation is triggered at that L1. When registration is transferred out of an L1, a flush is triggered at that L1. Registered data is not invalidated on a flash invalidation or written back on a flush.

A. *Workloads*

We select three graph processing applications from the Pannotia benchmark suite [17] to evaluate our system changes. The applications selected are:

Single-source shortest path (SSSP): Calculates the shortest distance between a source node and all other nodes in the graph.

Graph coloring (color): Assigns colors to nodes in a graph such that each node is a different color than its neighbors.

PageRank (PR): Generates a ranking of importance for each node in a graph based on its connectivity and the ranks of its neighbors.

Each of these applications converges on a solution by iteratively processing all nodes in a graph. Processing a node involves visiting each of the node's neighbors, so load imbalance is introduced through variation in the degrees of assigned nodes.

To mitigate load imbalance and demonstrate the value of dynamic sharing, each application has been modified to use per-CU task queues and work stealing. Graph nodes are initially evenly distributed across all 128 task queues. In the presence of load imbalance, underutilized CUs may steal from remote task queues when stealing is enabled. Work stealing is implemented similar to Orr et al [10][18]. Lock-free pop and steal functions are used to consume nodes from the local task queue and from a remote task queue, respectively.

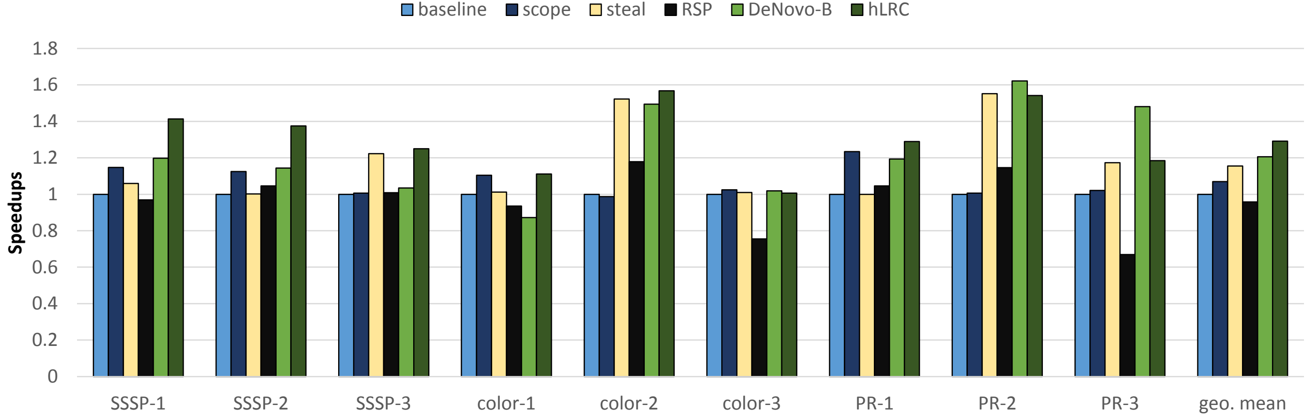

Figure 4. Performance (speedup relative to baseline).

Graph inputs are chosen from the Florida sparse matrix collection [19]. The inputs that we evaluated for each workload are listed in Table 5. These inputs were selected to fully utilize all 128 CUs.

B. Scenarios

We use 6 execution scenarios to evaluate hLRC: **baseline**, **scope-only**, **steal-only**, **RSP**, **DeNovo-B**, and **hLRC**. The **baseline** configuration uses neither scoped synchronization nor work stealing. Work-groups can only pull from their own work list and use agent scope for all synchronization. The **scope-only** configuration isolates the efficiency benefits of scoped synchronization by only allowing work-groups to pull from their own statically assigned work list and the work-groups use wg-scope for all synchronization. The **steal-only** configuration isolates the load balancing benefits of work stealing by allowing work stealing from remote CUs and using agent scope for all synchronization accesses. **RSP** uses the RSP implementation described in Section II.D to take advantage of both scoped synchronization and work stealing. Work-items use wg-scoped synchronization when pulling from their local work lists, and they use RSP synchronization when stealing from a remote work list. **DeNovo-B** represents a simplified implementation of the DeNovo protocol (described in section II.E) because it tracks registration at cache *block* granularity rather than at word granularity. It also differs from the prior implementation [8] because coherence regions are not implemented for read-only data. Although these simplifications reduce hardware and software complexity, they can lead to increased false sharing and more wasteful invalidations than would occur in the optimized protocol. **hLRC** is implemented as described in Section IIII; it can perform work stealing and exploit synchronization locality automatically through local registration.

V. Results

A. Performance of 128-CU GPU

Figure 4 compares the speedup of the 5 configurations described in Section IV.B relative to baseline. On average, the scope-only configuration improves performance by 7% relative to baseline, the steal-only configuration improves performance by 16%, the RSP implementation causes a 4% decrease in performance, DeNovo-B improves performance by 21%, and hLRC improves performance by 29%.

Each evaluated scenario triggers coherence actions at different rates for different reasons. To illustrate this, Figure 5 breaks down the L1 invalidations and Figure 6 breaks down the L1 flush and store registration actions. All scenarios trigger an equal amount of invalidation and flush actions at the start and end of a kernel, labeled *kernel start* and *kernel end*.

In addition, accessing the task queue can cause invalidation and flush actions. Specifically, in the baseline scenario popping a task causes the following coherence actions: an L1 invalidation occurs for every acquire operation and an L1 store buffer flush occurs for every release operation. These are labeled *acquire* in Figure 5 and *release* in Figure 6, respectively. The figures demonstrate that stealing causes the steal-only and DeNovo-B scenarios to experience more invalidate and flush actions in some cases. At the same time, RSP can experience fewer invalidate and flush actions because wg-scoped operations do not require L1 coherence actions. Instead, RSP additionally triggers 127 (one for every remote core) L1 self-invalidations for every remote release and 127 L1 store buffer flushes for every remote acquire (i.e., for every steal attempt). These are labeled *remote inval* and *remote flush* respectively. Finally, hLRC triggers an L1 invalidation whenever an L1 obtains exclusive ownership for an atomic variable, and an L1 flush whenever an L1 loses synchronization (i.e., for a synchronization miss or eviction). These are labeled *atomic in* and *atomic out* respectively.

Figure 7 shows the combined latency of all non-atomic accesses, broken down by load and store operations and normalized to baseline. Since all scenarios load and store approximately the same amount of data, this graph helps explain how data access latency is affected by each protocol. Figure 8 shows the combined latency of all acquire operations, release operations, and atomic accesses, broken down by operation type and normalized to baseline. This helps explain how synchronization operations, which are less numerous than data accesses but are often on the program's critical path, are affected by each protocol. *Acquire* includes latency from the

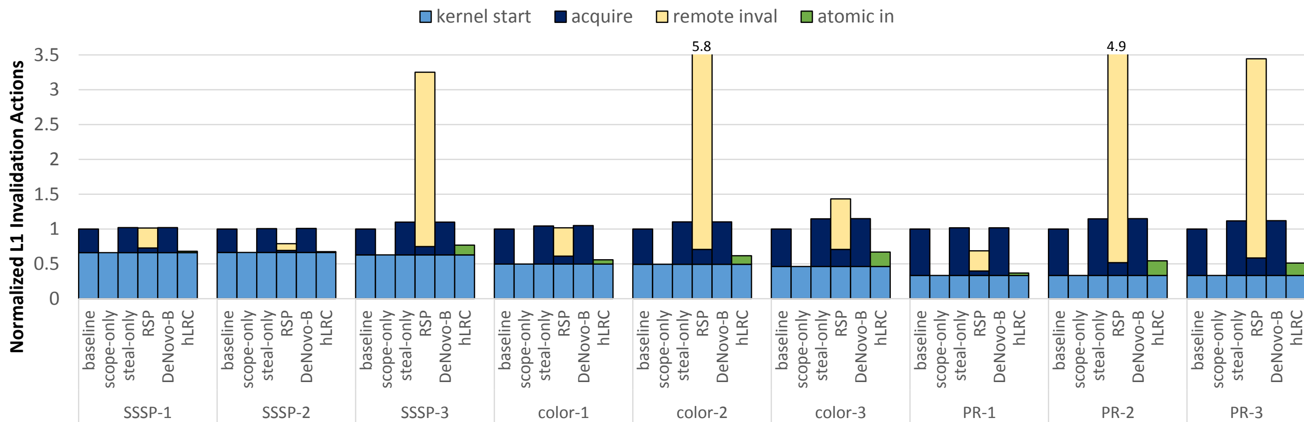

Figure 5. Total counts of L1 cache invalidation coherence actions, normalized to baseline.

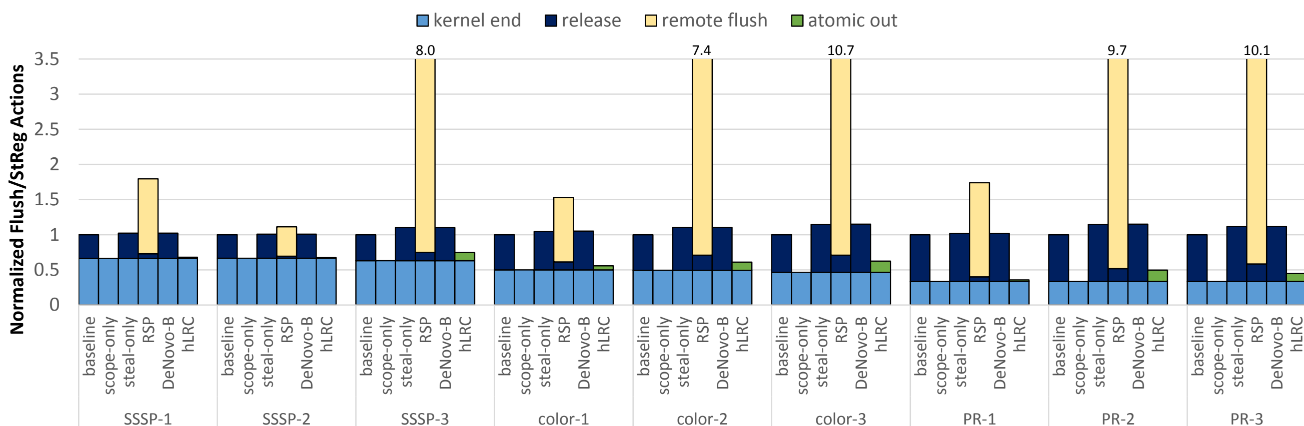

Figure 6. Total counts of L1 store flush/store registration coherence actions, normalized to baseline.

kernel start and *acquire* actions in Figure 5, and *Release* includes latency from the kernel end and release actions in Figure 6. Latency from the atomic in and atomic out invalidate/flush actions is included in *Atomic LD/ST/RMW*.

Using these detailed graphs, we next compare the performance of the RSP, DeNovo-B, and hLRC scenarios.

RSP: Although past work has shown RSP can benefit from simultaneously providing both scoped synchronization and work stealing when the number of CUs is small (e.g., eight) [10], it is clear from these results that the initially proposed RSP implementation does not scale. On 128 CUs, the broadcast lock, invalidate, and flush commands of the RSP implementation greatly increase coherence actions, data access latency, and synchronization latency. This ultimately degrades performance relative to the baseline configuration by up to 33%.

DeNovo-B: The DeNovo-B configuration scales well to 128 cores and can exploit significant cache locality in the presence of work stealing, delivering the best performance for multiple workloads. However, it does not provide the benefits of both scope-only and steal-only for multiple other workloads. DeNovo-B differs from RSP and hLRC in that it obtains local registration for all stored data, and it performs coherence actions for every atomic. DeNovo-B's L2 inclusivity for all writes causes increased contention and evictions at the L2 cache, which in turn increases release latency and fills up the store buffer, stalling subsequent accesses (evident in the high release latency, acquire latency, and store latency in SSSP-3 and color-1). However, our implementation of store registration also means the L2 functions as a write-back cache for dirty data, which enables improved reuse of written data through the L2. While other protocols maintain consistency with the CPU by invalidating dirty L2 cache lines as soon as the write-through to the backing cache completes, the DeNovo implementation keeps dirty data registered until the data is evicted, even across kernel boundaries. Therefore, DeNovo is better able to exploit reuse at the L2 between CUs and across kernel boundaries, which is evident in the decreased release latency and load latency for PR-2 and PR-3.

Performing coherence actions at every synchronization reduces L1 reuse for some workloads (evident in increased data access latency of SSSP-1, SSSP-2, and color-1), but it also means DeNovo-B is less affected than RSP and hLRC by low synchronization locality (e.g., color-3, PR-2, and PR-3).

Overall, DeNovo-B outperforms hLRC where reuse of written data at the L2 is possible and synchronization locality is low, but performs relatively poorly when synchronization is

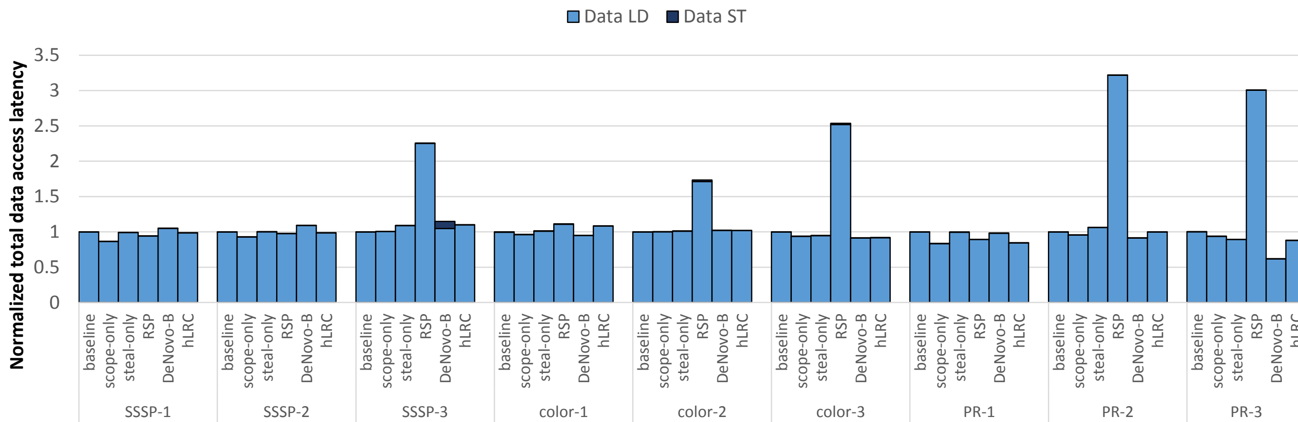

Figure 7. Total latency for data operations (load and store) normalized to baseline.

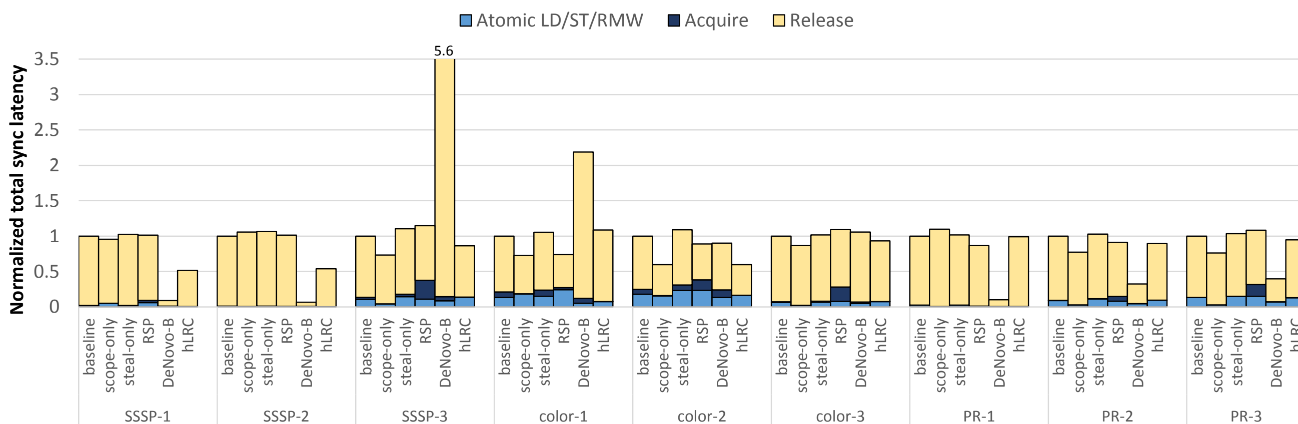

Figure 8. Total latency for synchronization operations (atomic access, acquire, and release) normalized to baseline.

primarily local and load-load reuse at the L1 is frequent, or when L2 cache pressure is high.

hLRC: By automatically avoiding coherence actions when synchronization is local, hLRC is able to exploit the benefits of both work stealing and cache reuse. As Figure 5 and Figure 6 show, hLRC decouples flush and invalidate operations from acquire and release accesses, reducing load, acquire, and release latency when synchronization locality is high. However, since hLRC's benefits rely on synchronization locality, its gains are less pronounced when stealing is frequent, and it performs roughly the same as the steal-only configuration when steal-only is dominant (SSSP-3, color-2, PR-2, PR-3). This is because for every acquire-release pair between remote CUs, hLRC incurs the latency of serially executing flush and invalidate actions as part of the acquiring atomic access, rather than preemptively flushing at the release. Although the increase in atomic latency is in most cases outweighed by a decrease in release latency (Figure 8), atomic access latency is more likely to be on the critical path than a release operation, so it can have a larger impact on performance.

However, the common case is local synchronization. Table 6 gives the proportion of hLRC synchronization accesses that are satisfied in the local L1 cache (L1 hits), in the L2 cache or memory (L2/Mem hits), and in a remote L1 cache (Remote L1 hits). Note that an L1 hit does not require a coherence action, a L2/Mem hit triggers an L1 invalidation, and a Remote L1 hit triggers an L1 flush and an L1 invalidation. For completeness, evictions that result in the loss of L1 registration are also shown (Synch Evicts), normalized to the total number of synchronization accesses. These evictions also result in an L1 flush, but since non-registered data is prioritized for eviction in hLRC, this count is near 0 for all inputs. Overall, atomic locality is high for the work stealing applications studied, and in every workload a significant majority of atomic accesses require no coherence actions. As a result, hLRC is able to outperform all other configurations on average.

In summary, RSP, DeNovo-B and hLRC all attempt to simultaneously optimize for cache locality and work stealing, but only hLRC is able to consistently match or exceed the performance of the best of scope-only and steal-only for the workloads studied. RSP's broadcast operations significantly degrade performance at 128 CUs, causing RSP to perform up to 33% worse than the baseline. DeNovo-B improves upon RSP, providing an average performance improvement of 21% over the baseline. With a write-back L2 and a lack of reliance on synchronization locality, the DeNovo-B implementation is even able to outperform hLRC when locality in written data is

Table 6. Synchronization hit proportions.

Bench-mark	L1 hits	L2/Mem hits	Remote L1 hits	Synch Evicts
SSSP-1	95.7%	0.8%	3.6%	0%
SSSP-2	97.8%	0.3%	1.9%	0%
SSSP-3	76.0%	4.6%	19.4%	0.67%
color-1	92.3%	0.32%	7.4%	0%
color-2	85.3%	1.0%	13.7%	0%
color-3	74.3%	5.6%	20.0%	0%
PR-1	96.6%	1.0%	2.4%	0%
PR-2	80.7%	4.3%	15.0%	0%
PR-3	82.4%	6.4%	11.2%	0%

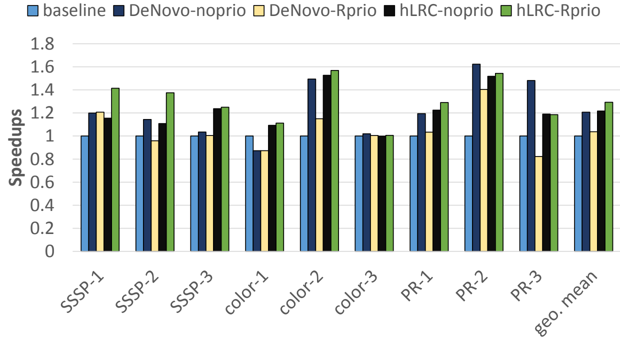

Figure 9. Impact of owned prioritization on performance.

available and stealing is frequent. However, the overheads of registration for data accesses and increased coherence actions outweigh these benefits in most cases, and hLRC performs on average 7% better than DeNovo-B for the work stealing workloads studied.

B. *Analyzing the optimized replacement policy*

We next analyze the impact of an optimized L1 and L2 replacement policy that avoids evicting inclusively tracked cache lines containing atomic variables for DeNovo-B and hLRC. Figure 9 shows the speedup of four configurations relative to the baseline: DeNovo-noprio, DeNovo-Rprio, hLRC-noprio, and hLRC-Rprio. DeNovo-Rprio and hLRC-Rprio use a state-aware replacement policy which ranks cache lines first by state (non-Registered data is preferred for eviction), and then by last use. DeNovo-noprio and hLRC-noprio uses an unbiased least-recently-used (LRU) replacement policy, which does not take into account the Registered state of the cache line. The DeNovo-B and hLRC policies evaluated in Section V.A are the same as the DeNovo-noprio and hLRC-ownprio configurations here.

hLRC-noprio achieves on average 22% speedup relative to baseline while hLRC-Rprio achieves on average 29% speedup. This difference in performance demonstrates the importance of preventing unnecessary Registration state evictions and their subsequent coherence actions when possible. In the evaluated workloads, atomic operations only touch 24 KB of cache blocks. Thus by biasing the replacement policy to hold onto these blocks, the 4 MB L2 still provides a significant caching of non-atomic data and avoids unnecessary coherence actions.

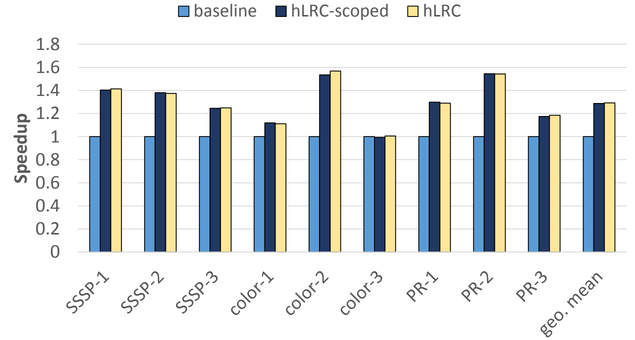

Figure 10. Speedup of hLRC-scoped vs. hLRC.

DeNovo, on the other hand, requires far more registration than hLRC. For the applications studied, multiple megabytes of stored data accumulate in DeNovo's L2 cache, depending on the size of the graph. Prioritizing these registered cache lines actually degrades performance in most cases, causing DeNovo-Rprio to perform only 3% faster than baseline on average, compared with 21% speedup for DeNovo-noprio.

C. *Using scopes with hLRC*

hLRC does not require scopes and simply acquires L1 registration for all atomic accesses, relying on implicit coherence actions to keep data consistent. However, if synchronization locality is low, then it may be preferable to perform atomic accesses at the L2 and perform explicit coherence actions as necessary. For example, if a release operation is known to only synchronize with an acquire operation from a remote CU, the CU should not cache the variable locally after the release. Obtaining local Registration only triggers a wasteful invalidation at the releasing CU and delays the release until the critical path for the acquire. Instead, the release should perform the atomic access at the L2 and preemptively flush any data that needs to be synchronized. Thus, the subsequent acquire operation does not need to look up the local Registration owner of the variable or trigger coherence actions on a remote CU. In essence, scopes could be used by expert programmers to optimize an hLRC system and exploit knowledge of communication locality. However, unlike HRF, incorrect assumptions about communication locality can only affect performance and not cause scope races and correctness bugs.

hLRC can support scoped synchronization in a straightforward manner. Specifically `wg`-scoped atomic accesses are handled the same as the non-scoped atomic accesses described in Section III. For an `agent`-scoped atomic access, an L1 flush (release) or invalidation (acquire) must be explicitly performed before or after registering the data at the L2 (which may require revoking registration from a remote L1), respectively.

We demonstrate the potential performance effects of using scopes to optimize hLRC by comparing hLRC, as described in the prior sections, to a configuration that uses scopes, called hLRC-scoped. If steals are rare, the scoped implementation should perform better because the stealing threads do not acquire local Registration and avoid needless coherence

actions related to Registration transfer. In reality Figure 10 shows that on average, both hLRC and hLRC-scoped achieve 29% speedup over baseline. hLRC-scoped does not perform consistently better than hLRC, and in multiple cases (SSSP-1, SSSP-3, Color-2, PR-3) performs slightly worse because stealing does exhibit some locality. When multiple steals from the same work-group occur, hLRC-scoped repeatedly pushes the atomic variable back to the L2 cache whereas in hLRC steals hit in the L1 cache, which amortizes the cost of local L1 Registration.

VI. RELATED WORK

Many existing coherence protocols exhibit some degree of laziness with regard to making written data visible to remote cores. Such strategies often rely on weak memory models and software-defined synchronization semantics to delay the point of stale data invalidation until the time at which it may actually be read [6][9][20][21], although the principle generalizes to models such as TSO as well [22][23]. However, in all of these protocols the propagation of dirty data to a shared cache level is still triggered at the writing core. hLRC takes laziness a step further by delaying the propagation of written data until a potential remote synchronization is detected. In this respect, hLRC more closely resembles Lazy Release Consistency, which has been proposed for distributed CPU systems [14]. By tracking synchronization variables and delaying all write propagation actions until a synchronization variable is accessed by a new core, reduced communication and improved reuse is possible.

Other work has evaluated multiple memory consistency models for GPUs and concluded that sequential consistency can be implemented in GPUs just as efficiently as weaker memory models and should be preferred [24][28]. However, their evaluations only assessed hardware coherence protocols, such as MESI, which require writer-initiated invalidation. The overheads required for such protocols are not attractive for current GPUs, and these prior studies lack comparisons to more realistic data points.

There have been multiple attempts to limit the overheads of hardware coherence protocols for GPUs. Heterogeneous System Coherence uses a hierarchical directory to track coherence state at coarse-grain regions [25]. This work focused on reducing coherence traffic between the CPU and the GPU, which is orthogonal to hLRC's focus on intra-GPU communication. QuickRelease [26] is a write-through protocol designed for CPU-GPU systems. It can support high memory bandwidth, but it uses writer-initiated invalidation and broadcast invalidates for coherence, limiting scalability. Temporal coherence has also been proposed for GPU coherence [27]. This implementation avoids writer-initiated invalidation, but performance depends greatly on the lifetime of data in the cache: too short, and locality cannot be exploited; too long, and communication suffers.

Hower et al. defined the SC for HRF memory model to establish clear semantics for the synchronization operations that exist in current GPUs [6][7]. HRF enables programmers to exploit knowledge of locality by specifying the visibility of synchronization operations. In this way, synchronization between local threads can be performed cheaply. However, without scopes, synchronization is still expensive. Using scopes requires static knowledge of the locations of synchronizing threads, and it introduces the notion of a heterogeneous data race, which occurs when two threads synchronize at different scope instances. Work stealing is possible with HRF, but it requires that all synchronization occurs through an encompassing scope (e.g., agent scope). Thus, dynamic sharing patterns like work stealing are not able to fully utilize the caches in the SC for HRF memory model.

As already discussed, Remote Scope Promotion [10][13] extends HRF with additional memory access semantics in order to support dynamic sharing between remote threads. This enables programming models such as work stealing, but it adds overhead to remote accesses and further complicates the SC for HRF memory model.

VII. CONCLUSION

Scoped synchronization is currently used by modern GPUs, but it relies on a complex memory model and is unable to exploit locality in many types of sharing patterns. Remote scope promotion (RSP) has been proposed to enable more flexible communication patterns in GPUs, but RSP as originally proposed uses broadcast invalidate, flush, and lock commands which scale poorly.

In comparison, DeNovo coherence implements efficient dynamic sharing without the use of scopes. Rather than relying on software-specified synchronization locality to reduce the cost of coherence actions, DeNovo uses exclusive registration to exploit locality in written data, even in the absence of synchronization locality. However, it still must perform coarse-grain coherence actions on every acquire and release, and the overhead of registration can outweigh its benefits when written data exhibits minimal locality.

This work introduced hLRC, a novel method for GPU synchronization based on lazy release consistency. Similar to existing methods for GPU synchronization, hLRC uses acquire-release semantics and coarse-grain coherence actions to enforce consistency. Like RSP, hLRC relies on synchronization locality to avoid performing these actions when synchronization is local. However, hLRC does not rely on scopes and instead uses an atomic tracking hardware mechanism to trigger targeted coherence actions when necessary. By automatically exploiting available synchronization locality and performing targeted coherence actions only when needed, hLRC achieves efficient cache utilization and scalable communication. Specifically while RSP degrades the performance of a 128-CU GPU and DeNovo improves performance by on average 21%, hLRC achieves a 29% speedup on average compared with our baseline. Like DeNovo, hLRC it is able to achieve this efficiency without any knowledge of communication locality, thus enabling support for the simpler SC for DRF memory model.

Acknowledgments

We thank Steve Reinhardt and the anonymous reviewers for their helpful feedback. We also thank Sarita Adve for her insightful suggestions that significantly improved the DeNovo comparison in the final version of the paper. Finally, we thank Tony Tye, Brian Sumner, and Paul Blinzer for thoughtful discussions. This work was performed while John Alsop interned at AMD Research. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos.

References

[1] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011.

[2] International Organization for Standardization, "Working Draft, Standard for Programming Language C++," [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[3] S. Adve and M. Hill, "Weak Ordering -- A New Definition," in Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990."OpenCL 2.1 Reference Pages." [Online]. Available: https://www.khronos.org/registry/cl/sdk/2.1/docs/man/xhtml/.

[4] "CUDA C Programming Guide." [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[5] "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG) Version 1.0 Provisional," HSA Foundation, Spring 2013.

[6] 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 The 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-19), 2014.

[7] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models," In Transactions on Architecture and Code Optimization (TACO), 2015.

[8] 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 International Symposium on Microarchitecture, 2015.

[9] H. Sung and S. V. Adve, "DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations," In The 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-20), 2015.

[10] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, "Synchronization using remote-scope promotion," In The 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-20), 2015.

[11] Mike Mantor, "AMD Radeon™ HD 7970 with Graphics Core Next (GCN) Architecture," In HOT Chips, A Symposium on High Performance Chips, 2012.

[12] H. J. Boehm and S. Adve, "Foundations of the C++ Concurrency Memory Model," In PLDI, 2008

[13] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, "Remote-scope promotion: Clarified, rectified, and verified." In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2015.

[14] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," In Proceedings of the 19th Annual Symposium on Computer Architecture, 1992.

[15] International Organization for Standardization, "Working Draft, Standard for Programming Language C++," [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[16] AMD Research, "AMD's GEM5 APU simulator" [Online]. Available: http://www.gem5.org/wiki/images/7/7a/2015_ws_03_amd-apu-model.pdf

[17] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding Irregular GPGPU Graph Applications," In Proceedings of the International Symposium on Workload Characterizations, 2013

[18] D. Cederman and P. Tsigas, "Dynamic Load-Balancing Using Work-Stealing," In GPU Computing Gems Jade Edition, Wen-Mei Hwu (Editor-in-Chief), Morgan Kaufmann

[19] The University of Florida Sparse Matrix Collection, T. A. Davis and Y. Hu, ACM Transactions on Mathematical Software, Vol 38, Issue 1, 2011, pp 1:1 - 1:25. http://www.cise.ufl.edu/research/sparse/matrices

[20] A. Lebeck and D. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," In The 22nd International Symposium on Computer Architecture (ISCA), 1995.

[21] A. Ros and S. Kaxiras. "Complexity-effective multicore coherence," In The International Conference on Parallel Architecture and Compilation (PACT), 2012.

[22] M. Elver and V. Nagarajan. "TSO-CC: Consistency directed cache coherence for TSO," In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, 2014.

[23] M. Elver and V. Nagarajan. "RC3: Consistency directed cache coherence for x86-64 with RC extensions," In The International Conference on Parallel Architecture and Compilation (PACT), 2015.

[24] B. A. Hechtman and D. J. Sorin, "Exploring Memory Consistency for Massively-threaded Throughput-oriented Processors," in Proceedings of the 40th Annual International Symposium on Computer Architecture, 2013.

[25] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

[26] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt and D. A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, 2014.

[27] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor M, and T. M. Aamodt TM, "Cache coherence for GPU architectures," In The19th International Symposium on High Performance Computer Architecture (HPCA2013), 2013.

[28] A. Singh, S. Aga, and S. Narayanasamy, "Efficiently enforcing strong memory ordering in GPUs," In Proceedings of the 48th International Symposium on Microarchitecture, 2015.