Synchronization Using Remote-Scope Promotion

Marc S. Orr[†§], Shuai Che[§], Ayse Yilmazer[§], Bradford M. Beckmann[§], Mark D. Hill[†§], David A. Wood[†§]

[†]University of Wisconsin–Madison
Computer Sciences
{morr, markhill, david}@cs.wisc.edu

[§]AMD Research
{Shuai.Che, Ayse.Yilmazer, Brad.Beckmann}@amd.com

Abstract

Heterogeneous system architecture (HSA) and OpenCL™ define *scoped synchronization* to facilitate low overhead communication across a subset of threads. Scoped synchronization works well for static sharing patterns, where consumer threads are known *a priori*. It works poorly for dynamic sharing patterns (e.g., work stealing) where programmers cannot use a faster small scope due to the rare possibility that the work is stolen by a thread in a distant slower scope. This puts programmers in a conundrum: optimize the common case by synchronizing at a faster small scope or use work stealing at a slower large scope.

In this paper, we propose to extend scoped synchronization with *remote-scope promotion*. This allows the most frequent sharers to synchronize through a small scope. Infrequent sharers synchronize by promoting that remote small scope to a larger shared scope. Synchronization using remote-scope promotion provides performance robustness for dynamic workloads, where the benefits provided by scoped synchronization and work stealing are hard to anticipate. Compared to a naïve baseline, static scoped synchronization alone achieves a 1.07x speedup on average and dynamic work stealing alone achieves a 1.18x speedup on average. In contrast, synchronization using remote-scope promotion achieves a robust 1.25x speedup on average, across a diverse set of graph benchmarks and inputs.

Categories and Subject Descriptors C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming

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

ASPLOS '15, March 14 - 18, 2015, Istanbul, Turkey

http://dx.doi.org/10.1145/2694344.2694350

1. Introduction

As processors evolve to support more threads, synchronizing among those threads becomes increasingly expensive. This is particularly true for massively-threaded, throughput-oriented architectures, such as graphics processing units (GPUs), which do not support CPU-style "read-for-ownership" coherence protocols. Instead, these systems maintain coherence by "pushing" data to a common level of the memory hierarchy, accessible by all threads, which acts as the global coherence point. After synchronizing, threads must then ensure that they "pull" data from this common memory level (e.g., by invalidating their caches). For discrete and integrated GPU architectures, the global coherence point occurs at the last level cache (LLC) and memory controller, respectively, incurring very high latency. Many applications cannot amortize these high synchronization delays, which limits their performance on GPUs.

One approach to reducing synchronization latency is *scoped synchronization*, which partitions threads into subgroups called scopes. Threads in the same scope can synchronize with each other through a common, but non-global (i.e., scoped) coherence point. General-purpose GPU (GPGPU) languages like OpenCL 1.2 and CUDA have historically provided limited forms of scoped synchronization like work-group barriers [1][2]. Recently, heterogeneous system architecture (HSA) and OpenCL 2.0 (which closely follows HSA) introduced more general scoped synchronization primitives based on acquire/release semantics [3]. For example, in HSA, synchronization operations are tagged with one of the following scope modifiers: work-item (`wi`, a GPU thread), wavefront (`wv`), work-group (`wg`), component (`cmp`), or system (`sys`).

Scoped synchronization works well for static communication patterns, where producers and consumers have well-defined, stable relationships. Figure 1 (a) depicts an example of a *static local* sharing pattern, where two OpenCL work-groups, `wg0` and `wg1`, operate on separate data (`data0` and `data1`, respectively). Work-items within `wg0` use `wg`-scoped synchronization to coordinate operations on `data0`;

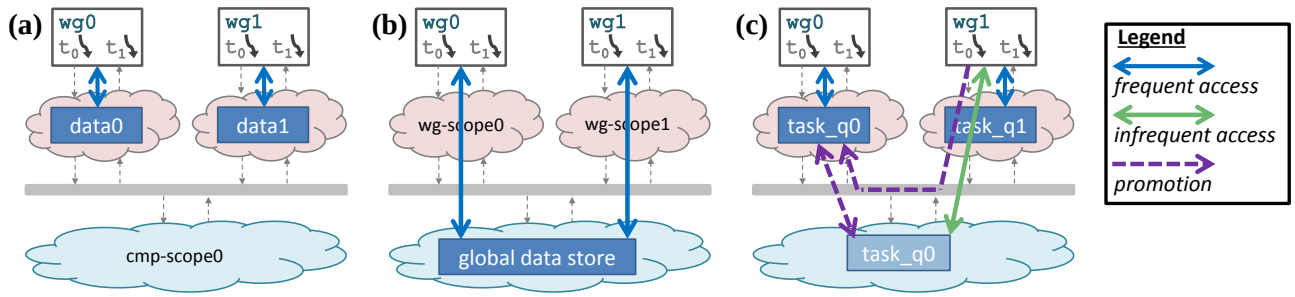

Figure 1. (a) Static local sharing. (b) Dynamic global sharing. (c) Dynamic local sharing.

wg-scoped operations are appropriate because only work-items from `wg0` operate on `data0`. Similarly, work-items in `wg1` use wg-scoped synchronization to operate on `data1`. Static local sharing is supported in OpenCL 2.0.

A larger scope (i.e., `cmp` scope) is required when data are shared by work-items in different work-groups. An example of this communication pattern, which we call *dynamic global* sharing, is shown in Figure 1 (b). In this scenario, work-items in `wg0` and `wg1` read and write a global data store. Component scoped synchronization guarantees that reads and writes to the global data store occur at a scope that is visible to all of the work-items. Dynamic global sharing can be expressed in both OpenCL and CUDA [4].

A third important sharing pattern is *dynamic local*, which occurs when a subset of work-items frequently access data within a smaller scope, but non-local work-items occasionally desire ad-hoc access. A common example of dynamic local sharing is work stealing, a scheduling policy that provides dynamic load balancing and is employed by several prominent CPU task runtimes including: Cilk, OpenMP, Intel's Threading Building Blocks (TBB), and Microsoft's Task Parallel Library (TPL) [5][6][7][8]. Figure 1 (c) illustrates work-groups `wg0` and `wg1` mostly accessing their local task queues: `task_q0` and `task_q1`, respectively. When a work-item finds its local task queue empty, it steals work from another task queue. Stealing mitigates load imbalance.

Ideally, accesses to the local task queue could be optimized with local synchronization. Unfortunately, scoped synchronization, as currently defined in HSA 1.0 and OpenCL 2.0, cannot efficiently support dynamic local sharing. To understand why, consider the work-stealing scenario depicted in Figure 1 (c). There are two scenarios to consider. First, when a work-item accesses its local task queue, it would like to use a smaller scope (e.g., `wg` scope) to reduce synchronization overheads. However, in the second scenario where a work-item steals work from another task queue, it must use a larger common scope (e.g., `cmp` scope). Scoped synchronization requires producers to synchronize at a scope that encompasses all of their consumers. Thus, because a stealer can be any work-item in the GPU, a work-stealing runtime requires all task queue synchronization to occur at the larger scope (i.e., `cmp` scope in this example). This means that smaller scopes cannot be used to optimize dynamic local sharing patterns like work stealing.

In this paper, we extend scoped synchronization to support dynamic local sharing using *remote-scope promotion*. Referring back to Figure 1 (c), work-items in `wg1` read from `task_q0` by promoting the scope (e.g., `wg` scope) of the most recent synchronization event on `task_q0` to a larger common scope (e.g., `cmp` scope) that encompasses work-items in `wg0` and `wg1`. Similarly, work-items in `wg1` modify `task_q0` by promoting the scope of the next future synchronization operation on `task_q0` to a common scope.

Remote-scope promotion naturally builds on hardware mechanisms prevalent in GPUs today and only requires a few minor modifications. Detailed simulation results show that remote-scope promotion leads to robust performance improvements across a diverse set of graph benchmarks and inputs.

To summarize, our contributions are:

- We identify that scoped synchronization, as defined in HSA 1.0 and OpenCL 2.0, limits dynamic local sharing.
- We propose a new synchronization semantic—remote-scope promotion— that allows a work-item to communicate outside of its scope hierarchy.
- We propose a hardware implementation for synchronization using remote-scope promotion that requires few changes to a state-of-the-art GPU.
- We evaluate our work for several emerging graph workloads and show that remote synchronization is 17% faster than scoped synchronization alone and 6% faster than work stealing alone.

2. GPU Architecture & Programming

This section reviews the GPU's hierarchical design and relates it to the GPU's relaxed memory model. Specifically, Section 2.1 describes how the HSA hierarchy maps to GPU hardware. Next, Section 2.2 explains how HSA extends acquire/release synchronization with scopes, corresponding to the HSA hierarchy, to minimize the cost of synchronization on GPUs. Finally, Section 2.3 illustrates how GPU synchronization is implemented through an example.

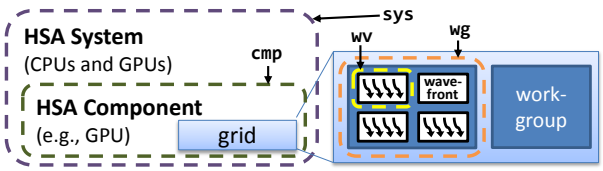

Figure 2. HSA's hierarchy of work-items.

2.1 The GPU Hierarchical Memory Model

GPUs attain high throughput by targeting massively parallel code that can be executed by thousands of threads. The GPU programming model incorporates hierarchy to help programmers organize threads, called work-items in HSA and OpenCL, in a GPU-efficient way. Figure 2 shows the HSA hierarchy of work-items [3]. An *HSA system* is composed of one or more *HSA components*, defined as a GPU for this paper. GPUs execute *grids*, which are sets of work-items that execute the same function at the same time. This organization, called single instruction multiple thread (SIMT), helps GPUs operate efficiently. The work-items within a grid are sub-divided into *work-groups*. Work-items within a work-group are scheduled together and can communicate with each other much faster than with work-items in other work-groups. Finally, an HSA component dynamically partitions work-groups into even smaller sets, called *wavefronts*, to match the GPU's execution width.

The example HSA-compatible GPU architecture illustrated in Figure 3 corresponds to the software hierarchy of work-items. A wavefront executes on single instruction multiple data (SIMD) units, which are sets of functional units that execute in lockstep. A work-group executes on a single compute unit (CU), which is composed of multiple SIMD units (four shown). Typically each CU also includes a private L1 data cache. CUs within a GPU share a common L2 cache that is used to communicate across a grid.

2.2 Scoped Synchronization Overview

HSA adopts acquire/release semantics from the C++ standard [9]. First, we describe how acquire/release synchronization works in C++. Next, we describe how HSA extends these semantics with scopes [10][11]. It is common to couple a synchronization operation with a memory instruction (e.g., *load acquire*, *store release*, etc.). Hardware implementations for acquire and release must do two things. First, they must correctly order memory operations around synchronization (i.e., acquire and release). This means that memory operations that occur after an acquire in program order cannot execute before that acquire (i.e., an acquire acts as a downward memory fence). Similarly, memory operations that occur before a release in program order cannot execute after that release (i.e., a release acts as an upward memory fence). In this work, GPU work-items execute in-order and block on loads; these two properties

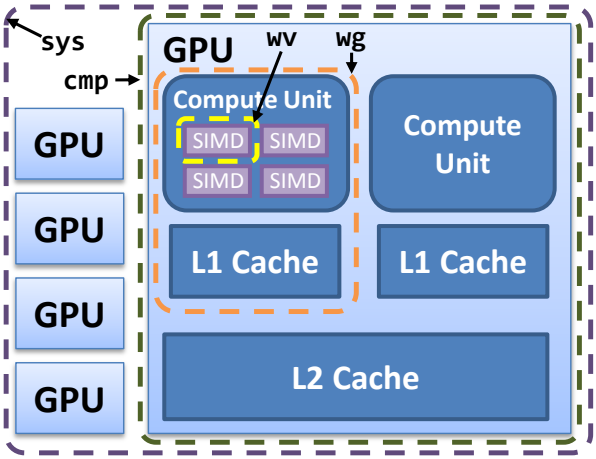

Figure 3. An example HSA cache hierarchy.

guarantee that a load will never be reordered around a synchronization operation. Stores are ordered by tracking their addresses in a hardware first-in/first-out (FIFO) buffer; this mechanism is described in detail in Section 2.3.

The second responsibility of hardware is to ensure that consumers see updates to memory made earlier by producers. Consumer work-items that execute an acquire "pull" the current global view of memory. The GPU implements a pull by invalidating the consumer's private caches, which prevents reading stale data. Producer work-items execute a release to globally "push" their updates to consumers. The GPU implements a push by flushing the producer's dirty cached data, to propagate it to consumers.

To limit synchronization penalties, HSA extends acquire/release semantics with scopes. Specifically, HSA defines the following scopes: work-item (`wi`), wavefront (`wv`), work-group (`wg`), component (`cmp`), and system (`sys`). Each scope corresponds to a different granularity of execution shown in Figure 2. Scopes can be applied to synchronization operations to limit the depth of the hierarchy that they synchronize to. For example, `wg`-scoped synchronization only needs to access the respective work-group's L1 cache since the entire work-group is scheduled on the same CU. Figure 3 shows how scopes are mapped to hardware.

Sequentially consistent for heterogeneous race free (SC for HRF) memory models were proposed to help programmers reason about scoped synchronization [10]. They extend sequentially consistent for data race free (SC for DRF) memory models with scoping rules. Two models were proposed. HRF-direct requires work-items to communicate through the same scope. HRF-indirect extends HRF-direct to allow transitive chains of communication through different scopes. HRF-Relaxed adds scope inclusion to both models [11]. We build on HRF-indirect-relaxed.

2.3 Scoped Synchronization Implementation

GPUs use simple write-through mechanisms to implement a relaxed memory consistency model. For example, many

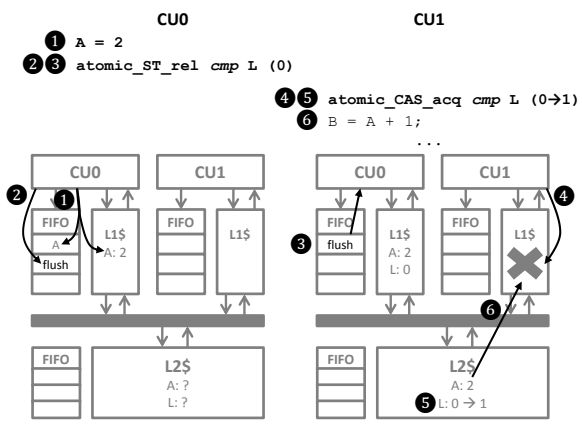

Figure 4. GPU release and acquire operation.

GPU architectures allows stores to complete without waiting for coherence operations, avoiding throughput bottlenecks caused by invalidation traffic [12]. In contrast, CPUs maintain the single-writer/multiple-reader invariant by stalling writes until all of their invalidations are acknowledged [13][14]. The GPU approach avoids invalidation traffic, but synchronization operations are expensive. In particular, release operations must wait until all prior writes reach the correct scope. Naïvely this can be achieved by tracking each outstanding write and waiting until all outstanding writes complete.

A better approach, proposed by Hechtman *et al*. [15], uses a FIFO at each cache level to track dirty addresses. This technique is demonstrated in Figure 4. Every write inserts its address at the FIFO's tail (time ❶), and when an address reaches the FIFO's head, its associated cache block is written to the next level of the hierarchy (if necessary). Release operations are handled by simply inserting flush markers into the FIFO (time ❷); the release completes when all prior addresses in the FIFO are written through to the appropriate scope (time ❸) and the flush marker reaches the FIFO's head. This FIFO approach for releases is assumed for this paper.

Hechtman *et al*. also proposed to use cache-block invalidations, but that adds considerable hardware complexity. Instead, we assume acquire operations simply invalidate all caches encompassed by the acquire's scope. Cache invalidates effectively "pull" the latest version of data from the distant levels of the memory hierarchy by removing closer cached data that have become stale. The invalidations themselves are cheap (i.e., a single-cycle flash-invalidate), but they decrease the effectiveness of caching and cause unnecessary inter-wavefront interference. Referring back to Figure 4, CU1 performs an acquire by invalidating its L1 cache (time ❹). The atomic compare-and-swap (`CAS`) operation is performed at the operation's respective scope. Specifically, the L2 cache controller performs the atomic read-modify-write (RMW) operation at time ❺ on address `L`. Finally, the value of `A` is returned at time ❻.

3. Extending Scoped Synchronization

Scoped synchronization is an exciting development in GPU programming models. Unfortunately, it currently suffers some severe programming limitations. Section 3.1 describes these limitations and Section 3.2 proposes our solution: synchronization using remote-scope promotion.

3.1 Limitations of Scoped Synchronization in HSA 1.0

When using scoped synchronization, programmers must manage both the level of visibility and the order of memory operations [10]. This is different than non-scoped synchronization, which only requires programmers to reason about order. To accommodate scopes, memory models apply scope modifiers to memory operations. In HSA, memory operations are tagged with one of the following scope modifiers: work-item (`wi`), wavefront (`wv`), work-group (`wg`), component (`cmp`), and system (`sys`).

A scope instance is a particular instantiation of a scope. For example, in Figure 5 `wg0`'s work-group scope instance is `wg-scope0` and `wg1`'s work-group scope instance is `wg-scope1`. HSA scope modifiers do not allow different scope instances to be distinguished. A work-item is associated with exactly one scope instance at each level of the scope hierarchy. This means that for a particular memory address to be visible to a work-item, that address must have been "pushed" (i.e., released) to a scope instance that is associated with (i.e., visible to) that work-item. This is because a work-item cannot "pull" (i.e., acquire) from a scope instance with which it is not associated. In other words, HSA defines push-pull semantics that require producers to push data to a scope that they share with their consumers.

These semantics make it difficult to optimize dynamic local sharing because consumers asynchronously read from producers. For example, in a work-stealing runtime, producers do not know when a subset of their consumers (i.e., the stealers) will read their data. Thus, work-items must conservatively push data to a scope that is visible to all consumers. Otherwise, a stealer may read stale data.

This conundrum is demonstrated in Figure 5 (a). Initially, at time ❶, a queue (`task_q0`) is instantiated for work-group 0 (`wg0`) in the component scope. In the common case, `wg0` is the only work-group that reads and writes `task_q0`. Thus the programmer would like to use work-group scoped synchronization to coordinate operations on `task_q0` (time ❷). Unfortunately, doing so would preclude work-items in a different work-group from stealing data from `task_q0`. For example, if a work-item in work-group 1 (`wg1`) were to read from `task_q0` (time ❸), it might find stale data in the component scope. Note that `wg1` cannot correctly use work-group scoped synchronization to operate on `wg0`'s scope instance.

It is important to note that the sequence of events in Figure 5 (a) would be correct in the absence of scopes. For

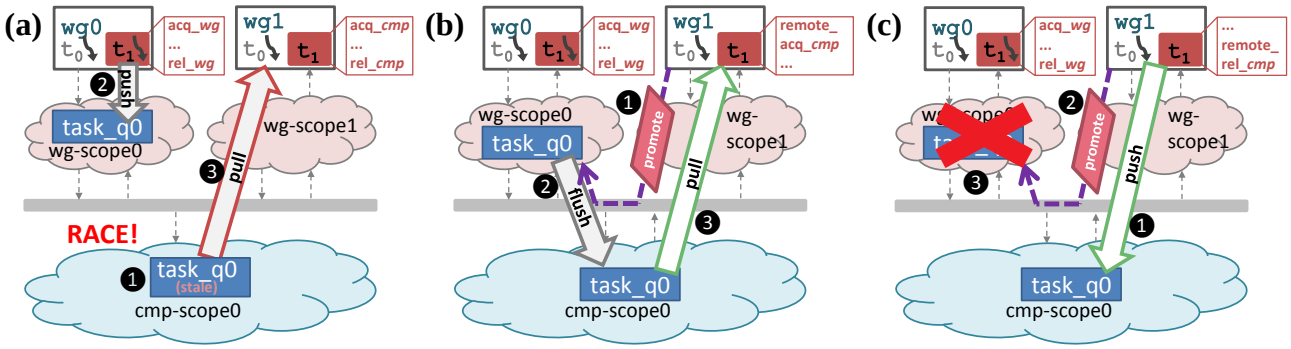

Figure 5. (a) Shortcomings of scoped synchronization. (b) Remote acquire. (c) Remote release.

example, in a traditional CPU, a release operation would make data visible to all threads in the system (i.e., all synchronization would occur at an implicit global scope).

3.2 Motivating Scope Promotion

Today, HSA requires work-items to synchronize through a scope that encompasses all sharers. This is because work-items are only able to synchronize with other work-items that are in the same scope hierarchy. To improve performance for dynamic-local sharing, it would be better if work-items could synchronize through a scope that encompasses the most frequent sharers instead of all possible sharers. Scoped synchronization, as defined in HSA and OpenCL, implicitly targets scope instances within the caller's scope hierarchy. A key insight is that extending this semantic to allow synchronization beyond the caller's scope hierarchy enables better use of scopes.

Towards this end, we propose synchronization using remote-scope promotion: a new semantic that allows a work-item to order memory accesses with a scope instance outside of its scope hierarchy. To access data in a remote scope, a work-item first promotes the remote scope to a larger common scope. After the promotion completes, the work-item synchronizes through the larger common scope. This sequence is shown in Figure 5 (b). At time ❶, `wg1` sends a message to promote `wg-scope0` to the shared component scope. Promotion is carried out by flushing dirty data in `wg-scope0` to `cmp-scope0` (time ❷). `wg1` can then pull the data with a `cmp`-scoped acquire (time ❸).

A similar procedure is used to modify data in a remote scope (e.g., `task_q0`'s head pointer). First, the data is written to a larger shared scope. Then the remote scope is promoted to that larger common scope. Figure 5 (c) illustrates how this works. At time ❶, `wg1` writes data through to the shared component scope; this is achieved by performing a `cmp`-scoped release. Next, `wg1` sends a message to promote `wg-scope0` to the component scope (time ❷). In this case, promotion is carried out by invalidating dirty data in `wg-scope0` (time ❸). This sequence guarantees that `wg0` will read updates written to the component scope by `wg1`.

Using scope promotion is surprisingly easy. Consider the tricky lock-free code in Figure 6 for stealing from a queue. This function is a part of an intricate work-stealing algorithm originally proposed by Arora *et al.* [16] and presented in the context of GPUs by Cederman and Tsigas [17]. We have augmented the code with remote synchronization highlighted in ***bold italics***. The primary difference is that the synchronization memory operations are trivially labeled with a remote memory ordering. Also, these operations must synchronize through the correct scope. The next section defines remote-scope promotion more precisely and proposes an implementation.

4. Remote-Scope Promotion

Synchronization using remote-scope promotion allows a work-item to pull and push data from a scope outside of its scope hierarchy without requiring a specific scope instance to be explicitly identified. Instead, programmers identify the common scope that a work-item shares with the target scope instance that the data resides in (e.g., the component scope encompasses `wg0` and `wg1` in the prior example).

In this section, we extend a current GPU instruction set to support remote synchronization and then we describe how these instructions can be implemented with only modest changes to current GPUs.

```
Task steal(Queue *q, Queue_Elem *val) {
 int tail = remoteRead(q->tail,
                       acq, wg cmp);
 int oldHead = q->head;
 if (tail <= oldHead)
    return NULL;
 Task task = q->contents[oldHead];
 int newHead = oldHead + 1;

 if (remoteCAS(&q->head, oldHead, newHead,
               ar, wg cmp))
    return task;

 return abort;
}
```

Figure 6. Steal with remote synchronization.

4.1 Defining Remote-Scope Promotion

We introduce three remote synchronization operations to carry out the promotion semantic described in Section 3.2:

Remote Acquire (rm_acq) ordering allows consumer work-items to acquire data produced at any scope instance. A remote acquire operation on address L comprises two steps: (1) promote the scope of the last release on address L to match the target scope of this remote acquire, (2) perform an acquire on address L at the target scope.

Remote Release (rm_rel) ordering allows work-items to propagate their updates to external scope instances. A remote release operation on address L comprises two steps: (1) perform a release on address L at the target scope, (2) promote the scope of the next acquire on address L to match to the target scope of this remote release.

Remote Acquire+Release (rm_ar) ordering combines the two remote operations above. To keep the presentation both minimal and complete, we label remote RMW operations with acquire+release ordering.

4.2 Implementing Remote-Scope Promotion

Implementing each remote order in a GPU can be thought of in terms of sub-operations. This sequence of sub-operations ensures that proper synchronization is maintained without directly involving the work-items in the remote scope instance.

Remote Acquire ordering ensures that writes performed at a remote scope instance will be seen by the requestor. Three sub-operations are required. (rm_acq.1) The first step is to promote the scope of the last release to match the target scope of this acquire. This is achieved by flushing all remote scope instances that are inclusive with the target scope. For example, if the target scope is cmp, then all wg, wv, and wi scope instances encompassed by the requestor's cmp-scope instance will flush their dirty data out to the requestor's cmp-scope instance. These flushes propagate local updates to the target scope. (rm_acq.2) Once these flushes complete, an acquire is carried out by invalidating the requestor's scope instances up to the target scope; this guarantees that the requestor will see the latest data. (rm_acq.3) Finally, the requestor performs the actual memory operation, such as a load, at the target scope.

Remote Release ordering ensures that the writes performed by the requestor are guaranteed to be seen by the remote scope instances. Three sub-operations are required. (rm_rel.1) First, the requestor performs a release by flushing to the target scope. (rm_rel.2) After the flush completes, the associated memory operation, such as a store, proceeds at the target scope[1]. (rm_rel.3) Finally, once the memory operation completes, such as the store becomes visible at the target scope, all remote scope instances encompassed by the target scope instance are promoted to the target scope. This is achieved by invalidating them up to the target scope. For example, if the target scope is cmp, then all wg, wv, and wi scope instances encompassed by the requestor's cmp-scope instance are invalidated, which guarantees that remote work-items will see the latest data.

Remote Acquire+Release ordering combines the sequencing of the other two remote orderings to ensure sequentially consistent execution. Logically, this operation promotes the remote data to the target scope, performs the atomic RMW, and then performs remote invalidations to promote the scope of next acquire to the target scope. Four sub-operations are required. (rm_ar.1) The first step is to flush all of the remote scope instances encompassed by the target scope. The requestor also flushes its local data to the target scope to ensure correct operation when the remote scope instance is the same as the requestor's scope instance. (rm_ar.2) The requestor also invalidates its local data so that it will execute the RMW at the target scope (this can occur in parallel with the first sub-operation). (rm_ar.3) Next, the actual RMW operation is performed at the target scope. (rm_ar.4) Finally the instruction completes by invalidating all of the remote scope instances that are encompassed by the target scope, which guarantees that remote work-items will see the update performed by the RMW operation. This implementation ensures that all RMWs to a particular location are globally ordered. Section 4.4.1 discusses this requirement in further detail.

Formalizing Scope Promotion helps to disambiguate corner cases and is discussed in the Appendix. Because we leverage hierarchical memory that provides transitivity, a simple extension of HRF-indirect suffices [10]. Specifically, we build on an extended version of HRF-indirect called HRF-indirect-relaxed [11]. HRF-indirect-relaxed requires each synchronizing release-acquire pair in a transitive sequence to occur at a compatible scope. Our work leverages scope promotion, which allows a work-item to promote the scope of another work-item's acquire/release operations. Thus, the formal model extends HRF-indirect-relaxed to require each synchronizing pair in a transitive sequence to either: (1) have both operations occur at a compatible scope (as in HRF-indirect-relaxed), or (2) one operation must be remote-scoped to promote the pair to a common scope.

4.3 An Example of Remote-Scope Promotion

Figure 7 provides an example of synchronization using remote-scope promotion in a GPU composed of two compute units (CUs). We assume the GPU memory system presented in Section 2, which includes FIFOs at each level of the cache hierarchy to track dirty addresses. The example illustrates how a critical section protected by a lock,

[1] Potential races between the executing the memory operation (step 2) and remote invalidations (step 3) are avoided by stalling accesses to the cache line at the target scope until the remote operation completes.

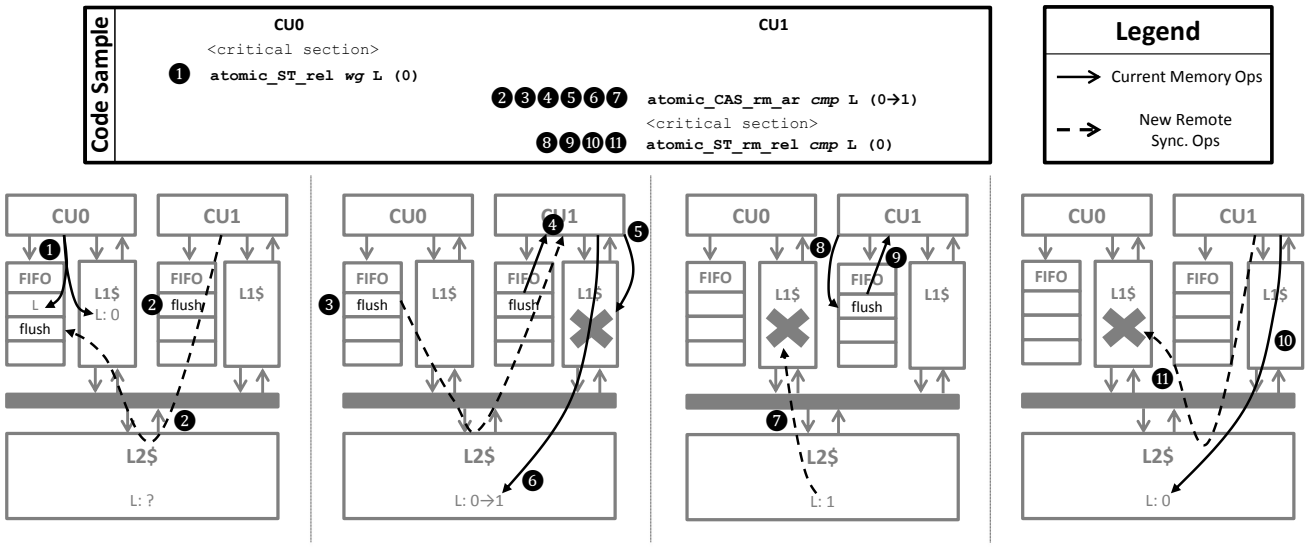

Figure 7. An example of synchronization using remote-scope promotion in a GPU.

which is frequently accessed by a single work-group, can be occasionally accessed in an ad-hoc manner by work-items outside of that work-group. The example assumes that the single work-group frequently accessing the critical section is executing on CU0 and the programmer has specified work-group scope for those synchronizing instructions. The work-items outside of that work-group execute on CU1 and use remote synchronization operations to properly synchronize with the work-group accesses. The example highlights that the work-items performing the remote synchronization operations encounter the primary performance overhead. Meanwhile, the work-group executing on CU0 encounters minimal overhead using work-group scoped instructions in the common case. The text that follows is labeled to clarify the corresponding event.

atomic_ST_rel (store-release on CU0 at wg-scope): The example in Figure 7 begins with a work-item executing on CU0 reaching the end of a critical section. The work-item issues a `wg`-scoped store release to notify other work-items that the critical section is open. Since the store release is at the work-group scope, no flush marker is inserted into the local FIFO. Instead, the store operation is immediately performed at the local L1 cache and the store's address `L` is inserted into the local FIFO (time ❶).

atomic_CAS_rm_ar (remote-acquire+release on CU1 at cmp-scope): Next, the work-group on CU1 wants to enter the critical section. It attempts to do so by using a remote synchronization operation to pull the valid data from the work-group executing on CU0. Specifically, at time ❷, a work-item on CU1 executes a `CAS` at `cmp`-scope with remote acquire+release ordering. The sub-operations are carried out as follows. (`rm_ar`.1) First, CU0's `wg`-scope instance must be promoted (i.e., flushed) to the component scope. This is achieved by broadcasting (via the on-chip network) a `cmp`-scope flush marker into all FIFOs in the component (including the FIFO of the requestor[2]). At time ❸, the remote flush marker reaches the head of its FIFO indicating that all prior updates by CU0 have propagated to the component scope and CU1 is notified at time ❹. (`rm_ar`.2) At time ❺, CU1 invalidates its local cache[3]; this is required for the RMW to execute at component scope. The promotion is complete.

An important correctness issue is that all RMW operations must be properly ordered. The remote acquire+release operation has temporarily promoted the `wg`-scope ordering point to the component scope. Thus, to maintain a single ordering point, all L1 cache controllers connected to the encompassing scope (i.e., component in this example) stall acquires, releases, and RMWs when they begin processing the flush marker sent from CU1.

(`rm_ar`.3) At time ❻, CU1 proceeds with the RMW (i.e., `CAS` reads 0 → writes 1) at the L2 cache and the work-item on CU1 gains access to the critical section. Synchronization is re-enabled at CU1's L1 cache. (`rm_ar`.4) Finally, the operation completes by promoting the scope of the next acquire to component scope. An invalidation message is sent to all remote L1 caches. CU0 receives the message and invalidates its L1 caches at time ❼. This ensures that CU0 will see the latest data at the component scope. The invalidation message also re-enables synchronization at the respective `wg`-scope. It is important to note that this final remote invalidation does not stall the remote acquire+release instruction. Rather, it just needs to be performed before the next acquire operation executed by each remote CU.

[2] In this example, both work-items reside on different CUs. Flushing the dirty data in the requestor's L1 cache is required when they reside on the same CU because the RMW operates directly at the L2 cache.

[3] The invalidation can execute in parallel with the flushes.

atomic_ST_rm_rel (remote-release on CU1 at cmp-scope): After exiting the critical section, CU1 issues a store with remote release ordering to release the lock. The sub-operations are carried out as follows. (`rm_rel`.1) At time ❽, a flush marker is inserted into CU1's FIFO to release updates, local to CU1, to the component scope. At time ❾, the flush marker reaches the head of the queue, indicating that all prior writes by CU1 are visible at the component scope. (`rm_rel`.2) Now assured that all prior writes are visible, at time ❿, CU1 performs the store operation, writing value '0' to address `L`. (`rm_rel`.3) Finally, at time ⓫, CU1 completes the remote release by invalidating all remote L1 caches encompassed by the component scope, which promotes the next acquire to the component scope.

This example illustrates that synchronization using remote-scope promotion defers overheads to the infrequent case (e.g., stealing) rather than the frequent case (e.g., accessing the local task queue). In addition, the example highlights that remote synchronization seamlessly builds on top of current state-of-the-art GPU memory systems (Section 2) with only a few additional mechanisms.

4.4 Hardware Support for Remote Synchronization

Assuming that GPUs already use per-CU FIFOs to track the partial order of writes and releases, remote-scope promotion requires three additional mechanisms.

The first allows work-items to send commands (e.g., flush and invalidate) and acknowledgments (acks) to other (non-local) CUs. These messages are distinguished in Figure 7 by dashed lines. For example, at time ❷, CU1 inserts a flush marker into CU0's FIFO, and at time ❹, CU1 receives an ack that the remote flush is complete. Similarly, at time ⓫, CU1 invalidates CU0's L1 cache.

The second hardware mechanism is the ability to lock cache lines at the targeted scope for remote-release, as described in Section 4.2. Finally, the third hardware mechanism is enabling remote-acquire+release to block issuing new synchronization operations within the target scope. In particular, Section 4.4.1 describes how remote RMW operations establish a valid *coherence order* using this third mechanism. Alternative solutions may be more efficient, but will require more hardware complexity to handle the longer duration of remote RMWs and merging multiple versions of the cache line within the target scope.

4.4.1 Digging Deep: Ordering Remote RMWs

Correctly ordering remote RMWs is particularly dubious and requires careful consideration. For instance, Hower *et al*. [10] previously identified that race-free code using scoped synchronization must observe a total order of RMWs (referred to as atomics in that work). More recently, Gaster *et al*. [11] and the HSA memory model [3] formally defined this requirement by stating that a GPU implementation must provide a "coherent order" for all accesses to a single memory location. In addition, they state that "the read and write components of an atomic RMW must be adjacent in coherent order." Thus it is necessary for our proposed hardware to ensure that all race-free RMW (both relaxed and non-relaxed) operations are totally ordered.

Our implementation adheres to the principles of the HRF memory models, including the total order of all RMWs. The challenge in ordering RMWs is ensuring that they can immediately observe their place in a location's coherent order (for example, atomic increment instructions that read the previous value). Therefore, remote RMWs must immediately be ordered across all scope instances.

Our solution is to utilize the previously described remote acquire+release ordering rules for all remote RMWs regardless of what ordering the instruction specifies. When a remote RMW is being performed, we stall all RMW operations at L1 caches within the target scope to avoid possible conflicts (i.e., remote scope instance modification vs. modifications at the promoted scope). While heavyweight, this approach ensures correctness and should not impact performance as long as remote RMWs are used sparingly.

5. Methodology

We used gem5 [18] coupled with a proprietary GPU model to evaluate synchronization using remote-scope promotion. The evaluated GPU is based on the design previously shown in Figure 3. It has eight CUs and each CU has four SIMD units. 40 hardware wavefront contexts are time-multiplexed across the four SIMD units using an oldest-job-first scheduling policy. Each CU has a private L1 data cache. There are two instruction caches and 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. Details about the baseline GPU configuration can be found in Table 1.

The L1 data caches and L2 cache are write-through and write-no-allocate. Acquire and release operations are implemented as described in Section 2. Specifically, an ac-

Table 1. Simulation configuration

8 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	512kB, 64B line, 16-way, 24 cycles
1 Instr. cache/4 CUs	32kB, 64B line, 8-way, 4 cycles
DRAM	DDR3, 8 Channels, 500 MHz
Protocol	Write-through, write-no-allocate, acquires trigger scoped flash invalidate, releases trigger scoped FIFO flush
Task Runtime	
8 task queues	1 work-group/queue, 4 wavefronts/work-group

quire operation invalidates the caches between the acquire's wavefront up to the scope of the acquire. Every cache is paired with a FIFO and a release operation flushes the FIFOs from the release's wavefront up to the scope of the release.

We extended the GPU model to support remote-scope promotion. This required adding the new remote synchronization orders proposed in Section 4.1 to the model and modifying the L1-data and L2-cache controllers to carry out remote synchronization, as described in Section 4.2, whenever they encounter a remote operation.

5.1 Workloads

We selected three applications from the Pannotia benchmark suite [19] and evaluated them across a diversity of inputs. Pannotia is a collection of graph workloads implemented in OpenCL. These applications suffer load imbalance when individual work-items process vertices with different numbers of neighbors. It is a challenge to predict the load distributions before running the applications because the resulting load imbalance is dependent on the graph input. The graph applications we evaluated are:

Single-source shortest path (SSSP): Given a source vertex, finds the shortest distances of all other vertices to the source by gradually expanding vertex frontiers.

Graph coloring (color): Iteratively labels a graph with different colors. Individual vertices determine if they should be assigned a new color by evaluating their neighboring vertices each iteration.

PageRank (PR): Estimates the importance of web pages by repeatedly transmitting values among vertices through edges that represent pages and links, respectively.

The graph inputs are chosen from the 9th and 10th DIMACS implementation challenges [20] and a graph for interacting proteins [21]. The inputs that we evaluated for each workload are listed in Table 2. We modified the OpenCL code to use work stealing. Our implementation follows the lock-free algorithm described by Tsigas and Cederman [17]. We allocated and associated a task queue to a work-group composed of four wavefronts. We instantiate eight task queues across the eight CUs. A queue element represents the work consumed by all of the work-items in a work-group. Important operations include: (1) pop, which dequeues an element from the *tail* of the local queue, and (2) steal, which dequeues an element from the *head* of a remote queue. Popping and stealing from different ends of the queues ensures that these operations conflict at most once per queue. Pushing to the tail of the local queue is not necessary for this particular set of workloads.

Table 2. Workloads and inputs

Benchmarks	Graph Inputs
Single Source Shortest Path (SSSP)	dip20090126_MAX (1), USA-road-d.BAY(2), USA-road-d.COL(3)
Graph Coloring (color)	dip20090126_MAX(1), USA-road-d.NY(2), ecology1(3), G3_circuit(4)
PageRank (PR)	cond-mat-2003(1), small-world(2), coAuthorsDBLP(3)

Initially, each task queue is statically assigned work elements (i.e., the graphs' vertices are evenly distributed across the queues). On behalf of the entire work-group, one work-item pops a queue element from its local queue, and application-specific information (e.g., parameters and pointers to the actual work) is provided to all of the work-items in the work-group. When the local queue is empty a work-group attempts to steal an element from a remote queue. The kernel terminates when all of the queues are empty. During the execution, contention may happen, for example, when a pop and steal collide and try to dequeue the same element. The algorithm [17] uses a lock-free solution to ensure these cases are handled efficiently.

5.2 Evaluation Scenarios

Four scenarios are used to evaluate remote synchronization:

Baseline: In the first scenario, called *baseline*, stealing is disabled and global synchronization is used to dequeue elements. Technically, synchronization can be avoided in the baseline scenario because our applications don't enqueue work, but we are interested in how our technique applies beyond this particular set of graph applications.

Scope-only: The second algorithm, called *scope-only*, improves on baseline by replacing global synchronization with scoped synchronization. The scope-only scenario uses optimizations available in OpenCL 2.0 today.

Steal-only: The third algorithm, called *steal-only*, improves on baseline by replacing its static scheduling algorithm with work stealing. Again, this can be achieved using component scoped synchronization in OpenCL 2.0 today.

Rem-sync: Finally, the *rem-sync* scenario applies both scopes and work-stealing to the baseline. This is possible because of the remote-scope promotion technique proposed in this paper.

6. Results

We find that synchronization using remote-scope promotion is able to achieve the benefits of both work stealing and scoped synchronization without compromise. Figure 8 compares remote synchronization (rem-sync) to the other three evaluation scenarios described in Section 5.2. Figure 8 shows that remote synchronization achieves speedups of on average 1.25x over baseline. In contrast, the average scope-only speedup is only 1.07x and the average steal-only speedup is 1.18x. Remote synchronization achieves higher speedup than both the scope-only and steal-only scenarios because is able to optimize for both scopes and dynamic load balancing. As a result, remote synchronization always achieves the maximum or better than the scope-only and steal-only algorithms.

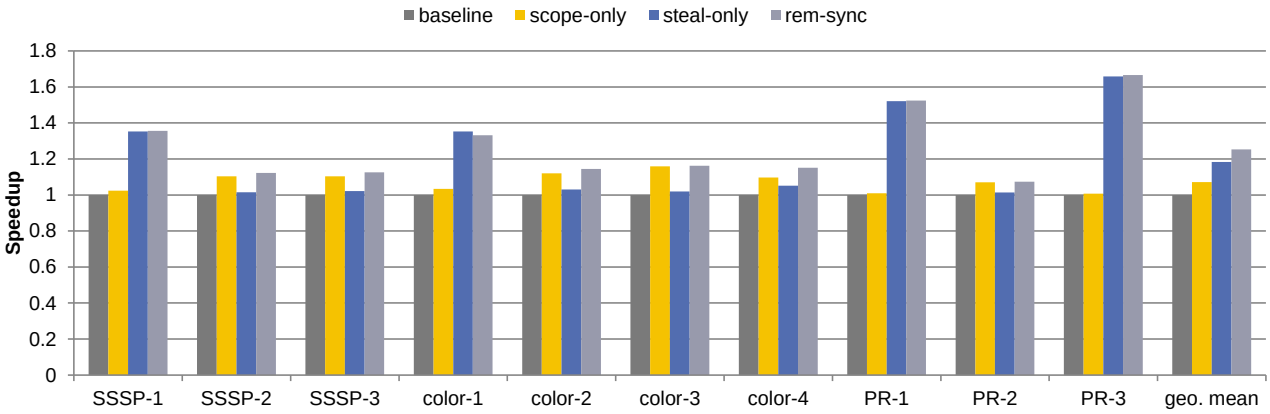

Figure 8. Synchronization using remote-scope promotion quantified.

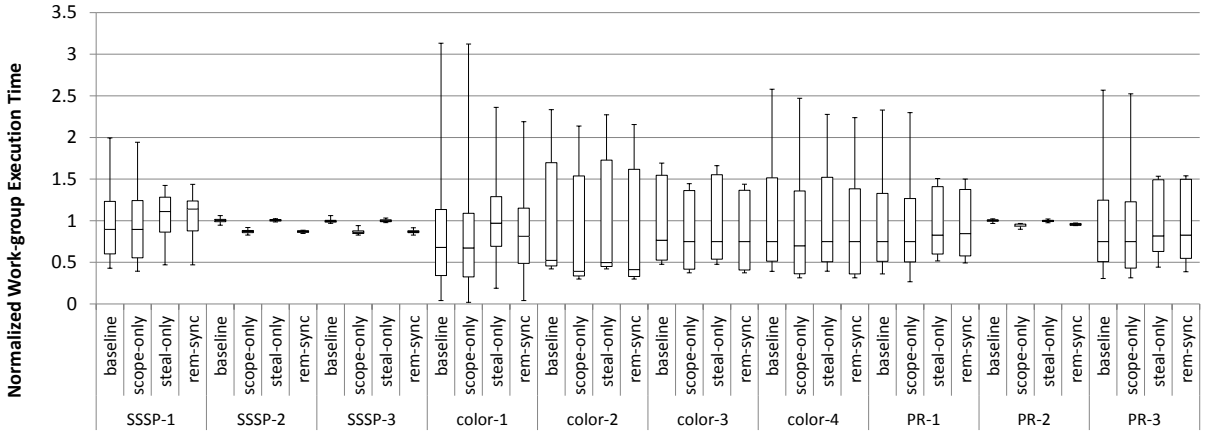

Figure 9. Load imbalance.

The workload/input combinations in Figure 8 clearly demonstrate a variety of behaviors. Some cases benefit from scopes, but are not affected by work stealing. For example, color-3 realizes a 1.16x speedup from scoped synchronization but is hardly impacted by work stealing. At the other extreme is PR-3 which runs more than 1.65x faster with dynamic load balancing, but is not helped by scoped synchronization. Workloads, like color-4, that benefit from both work stealing and scoped synchronization are able to use remote synchronization to optimize both cases at the same time. The key takeaway of Figure 8 is that the dynamic behavior of real workloads cannot be predicted and optimized statically. This fact puts programmers in a dilemma: whether to optimize for scopes or load balance. Remote synchronization eliminates this dilemma.

An important question is: why do some workloads do better with scoped synchronization while others do better with work stealing? In the following sections, we correlate the underlying data movement patterns across the different workload/input combinations to their application behavior.

6.1 Load Balance

Dynamic load balancing is crucial for workloads that have lop-sided data partitions to process. How evenly work is distributed across work-groups largely depends on a workload's input set. We would expect work stealing to accelerate inputs that lead to an uneven distribution of work. Figure 9 shows a box plot that quantifies the load imbalance for each workload/input combination. The box plot shows work-group execution times normalized to the baseline. The boxes in the box plot represent the 25th to 75th percentiles of work-group execution times. The horizontal line inside the box marks the median. Each box also has two tails. The bottom tail marks the minimum work-group execution time and the top tail marks the maximum.

Pannotia workloads execute in iterations. Different iterations may exhibit different load distributions. We find that many workloads with long top tails correspond to inputs that suffer load imbalance within an iteration. For example, SSSP-1, color-1, PR-1, and PR-3 all exhibit long top tails that are effectively reduced by work stealing. This reduction in the maximum work-group execution time ultimately reduces execution time. Interestingly, many of the color benchmarks exhibit long top tails because one iteration takes significantly longer than the others, but the intra-iteration variance is rather small except for color-1, which does benefit from work stealing. In general, the workloads with short boxes and tails correspond to inputs that are

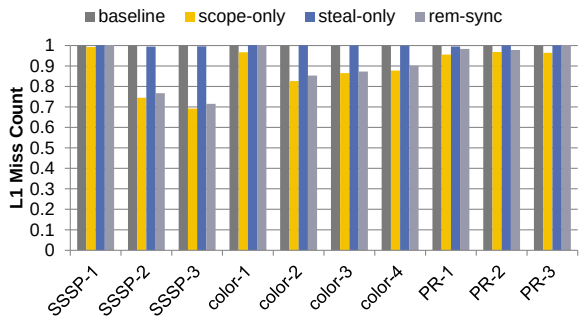

Figure 10. L1 miss rate.

unable to benefit from work stealing because of little variation in work-group execution time within an iteration.

6.2 Cache Behavior

Scoped synchronization is important for workloads that are able to take advantage of the combining provided by the GPU's L1 caches and FIFOs. When global synchronization is used frequently, addresses are unable to reside in the L1 caches long enough to achieve effective combining. In turn, more requests are issued to the L2 cache, which becomes a bandwidth bottleneck. Figure 10 shows the number of L1 cache misses for each algorithm, normalized to the baseline. The baseline, which uses `cmp`-scoped synchronization, suffers many more L1 cache misses than the scope-only algorithm, which uses `wg`-scoped synchronization. The difference in L1 cache misses indicates which workloads will benefit from scoped synchronization. Workload/input combinations that have less than 10% difference in L1 cache misses between the baseline and scope-only scenarios, which includes SSSP-1, color-1, PR-1, PR-2, and PR-3 benefit from scoped synchronization less than workloads with more than 10% difference.

6.3 Limiting Remote Synchronization Overheads

In the dynamic local sharing patterns that remote synchronization is intended for, remote operations should be rare. For example, in a work stealing runtime, the common case is to dequeue work from the local queue. Steals only occur when a local queue becomes empty. The first two columns in Table 3 show the percentage of queue elements that are stolen over each workload's duration. For most workloads, stealing is negligible, which helps explain why many cases don't benefit from work stealing. In contrast, the workloads with the highest amount of load imbalance—SSSP-1, color-1, PR-1, and PR-3—execute the largest number of steals. The surprisingly high number of steals for these workloads is due to our work-stealing algorithm, which steals one task queue element at a time. This policy works best when stolen tasks generate new local tasks, but this is not the case for our workloads. Thus, single-element steals occur frequently when there is significant load imbalance. Table 3 also shows the percentages of failed steal operations. Too many failed steal operations could degrade performance, but fortunately we do not observe significant failures.

Table 3. Steal behavior

	% of queue elements stolen		*% of steals that fail*	
	steal-only	*rem-sync*	*steal-only*	*rem-sync*
SSSP-1	9.0	9.7	1.6	0.7
SSSP-2	0.5	0.6	0.8	3.6
SSSP-3	0.6	0.7	0.6	2.2
color-1	9.6	10.3	0	2.7
color-2	1.2	1.0	0.9	1.1
color-3	0.7	0.4	0	1.2
color-4	1.5	1.4	0.1	0.4
PR-1	5.5	5.5	0	0
PR-2	0.8	0.6	0	0
PR-3	5.1	5.1	0	0

Recall that there are two costs to remote synchronization. The first cost is that caches become less effective (e.g., a remote release will invalidate all non-local caches). The second cost is the up-front latency to carry out a remote operation. For example, a remote acquire has to invalidate the local scope, flush caches in the remote scope, etc. These operations can take hundreds of cycles. Surprisingly, we found that remote synchronization is ~10 times faster than `cmp`-scoped synchronization on average. This is because remote synchronization allows all pop operations to be performed at work-group scope. Replacing `cmp`-scoped synchronization with `wg`-scoped synchronization reduces traffic at the cache controllers because there are less evictions, flushes, and synchronization events. So, in reality, while remote synchronization operations might take more time to service, they take less time overall because they spend less time queued at the cache controllers.

7. Related Work

Prior work studied scoped synchronization in the context of software distributed shared memory for CPUs [22][23]. More recently, Hower *et al.* developed the sequentially consistent for heterogeneous-race-free (SC for HRF) memory model [10] by incorporating scopes into the sequentially consistent for data-race-free (SC for DRF) memory model. Correctly placing synchronization operations in SC for DRF leads to well-defined behavior. In SC for HRF, this is not enough. All synchronization operations must also be labeled with the correct scope. Our work allows a work-item to label a synchronization operation so that it applies to a scope instance other than its own.

In one embodiment of SC for HRF, called HRF-indirect, work-items can transitively push values on behalf of other work-items in their scope instance. HRF-indirect alone is not sufficient to enable dynamic local sharing because the transitive push only applies to a work-item's scope instance hierarchy. In contrast, remote synchronization allows work-items to perform synchronization operations in a scope instance outside of their scope instance hierarchy.

work-item A	work-item B	work-item C
❶ atomic_CAS_acq_wg &L, 0, 1 ❷ enqueue(task_A); ❸ atomic_ST_rel_wg &L, 0	❹ atomic_CAS_acq_wg &L, 0, 1 ❺ enqueue(task_B); ❻ atomic_ST_rel_cmp &L, 0	❼ atomic_CAS_acq_cmp &L, 0, 1 ❽ enqueue(task_C); ❾ atomic_ST_rel_cmp &L, 0

Figure 11. Transitive synchronization. A and B are in the same work-group. A, B, and C are in the same component.

Quickly reacquirable locks (QRL) optimize for dynamic local sharing patterns on CPUs [24]. QRLs can either be in a *biased* state, which reduces the overhead of acquire and release operations, or in a *default* state, which treats the lock normally. To move the QRL from the biased state to the default state (analogous to a *remote acquire* in this paper), a non-biased thread simply updates the state of the lock using the CPU's invalidation-based coherence protocol. This approach cannot be applied to GPUs because their caches do not use an invalidation-based protocol.

Other work on GPU synchronization has not considered scopes. For example, transactional memory (TM) was extended to GPUs to improve the performance of and simplify the use of synchronization [25][26]. One could envision scoped transactions. Singh *et al*. proposed temporal coherence, which is a time-based self-invalidation coherence protocol for GPUs [27]. Scopes could potentially be applied to temporal coherence to reduce self-invalidations.

Previous work surveyed dynamic load-balancing policies for GPUs. Hower *et al*. evaluated work sharing with scoped synchronization [10], but did not consider work stealing due to limitations in SC for HRF. They found that it performs well, but our results are even more impressive. Tseng *et al*. compared work donation and work stealing on a GPU without scoped synchronization [28]. Both scheduling policies showed similar performance, but they preferred work donation because it enables fixed-size queues. However, neither work donation nor work stealing can use scopes without support for remote synchronization.

8. Conclusion

This paper proposed a new synchronization semantic: synchronization using remote-scope promotion. The basic idea is that the work-items that most frequently access shared data do so at the smallest scope possible. Work-items that access that shared data less frequently do so by promoting the small remote scope to a larger compatible scope.

Remote synchronization optimizes scoped synchronization for dynamic local sharing, which occurs when a data structure is shared by many threads, but a subset of those threads desire infrequent, ad-choc access. An important example of this is work stealing. Remote synchronization robustly supports both scoped synchronization and work stealing across a diverse set of graph workloads and inputs.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. We also thank Derek Hower for answering our questions about HRF-Relaxed. Finally, we thank Tony Tye, Jason Power, and Michael Scott for thoughtful discussions. 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.

Appendix: Formalization

We add the notion of scope promotion to HRF-indirect-relaxed. Before presenting these additions, we give some background on the existing HRF models and discuss why we build on HRF-indirect-relaxed in particular.

Prior work proposed two memory models for scoped synchronization: HRF-direct and HRF-indirect [10]. The scoping rules are a defining part of these models. Both models say that a synchronization pair (i.e., a release on an address, executed by one work-item, followed by an acquire on the same address, executed by another work-item) must occur at the same scope. HRF-Relaxed extends the original HRF models by adding scope inclusion, which occurs when: (1) both scopes in the pair are specified such that one is a subset of the other, and (2) both scopes include each work-item in the pair [11]. The HRF-Relaxed paper uses the following notation for scope inclusion: $Rel_{S,A}[\ell] \approx_{\text{incl}} Acq_{S',A'}[\ell]$. This expression says that the scope S of the release Rel on location ℓ executed by work-item, or agent, A is inclusive with the scope S' of the acquire Acq on location ℓ executed by work-item, or agent, A'.

What distinguishes HRF-indirect is that it allows different synchronization pairs to the same location to occur at different scopes; this property is called transitive synchronization. For example, consider the sequence shown in Figure 11. Work-items A and B first synchronize on location L at work-group scope. Then, work-items B and C synchronize on location L at component scope. This sequence, invalid in HRF-direct, is allowed by HRF-indirect.

Transitivity is useful for scope promotion because it defines the case where synchronization first occurs at the remote scope and then occurs at the promoted scope. Note that without some sort of scope promotion, there is no way for an arbitrary work-item (e.g., a stealer) to guarantee that

Scoped Synchronization Order ($\overrightarrow{so_a}$): Given a release memory action, $Rel_{S,A}[\ell]$, and an acquire memory action, $Acq_{S',A'}[\ell]$, $Rel_{S,A}[\ell]\ \overrightarrow{so_a}\ Acq_{S',A'}[\ell]$ iff $a \in S$, $a \in S'$, $Rel_{S,A}[\ell] \approx_{\text{incl}} Acq_{S',A'}[\ell]$, and $Rel_{S,A}[\ell]\ \overrightarrow{coh_\ell}\ Acq_{S',A'}[\ell]$. Scoped synchronization order captures the synchronization operations visible to a single agent a.

Figure 12. Definition of Scoped Synchronization Order.

the last release occurred at a compatible scope. Thus, we add scope promotion to HRF-indirect-relaxed.

Adding Scope-Promotion to HRF-indirect-relaxed

An important definition in HRF-indirect-relaxed is scoped synchronization order, shown in Figure 12, which describes the order of acquires and releases. We identify where synchronization using remote-scope promotion can violate scoped synchronization order and then propose an addendum to HRF-indirect-relaxed (Figure 13) to fix the issue.

The definition gives three conditions to establish that a release on location ℓ, $Rel_{S,A}[\ell]$, occurs before an acquire on ℓ, $Acq_{S'A'}[\ell]$ (i.e., $Rel_{S,A}[\ell]\ \overrightarrow{so_a}\ Acq_{S',A'}[\ell]$). First, the two scopes S and S' must include agent a. Note, there is a synchronization order for each agent (i.e., work-item) in the execution (e.g., the producer work-item that executes the release and the consumer work-item that executes the acquire). This is depicted in Figure 14. Referring to the figure, the two scopes are `wg0` (i.e., the release executed by agent 0) and `cmp0` (the remote acquire executed by agent 1).

Case 1 shows when agent 0 and agent 1 both execute in the same work-group, the two scopes include both agents. For this case, remote synchronization adheres to HRF-indirect-relaxed.

Case 2 shows agent 0 and agent 1 executing in different work-groups. Both scopes include agent 0, but `wg0` does not include agent 1. To fix this issue, agent 1 *promotes* the scope of agent 0's release. The promotion is signified by back arrow in the figure, which transforms the scoped synchronization order graph. Specifically, the node corresponding to the most recent release on the same location as the remote acquire is updated with a promotion semantic.

The second condition to establish scoped synchronization order is that both scopes are inclusive. Using properly

I. New Operators for Scope Promotion

Most Recent Release on ℓ: Given an acquire memory action, $Acq_{S,A}[\ell]$, return the most recent release memory action, $Rel_{S'A'}[\ell]$, in $\overrightarrow{coh_\ell}$:

$lastRelease(Acq_{S,A}[\ell]) = Rel_{S'A'}[\ell]$:
$(Rel_{S'A'}[\ell]\ \overrightarrow{coh_\ell}\ Acq_{S,A}[\ell]) \wedge (\nexists Rel'_{S''A''}[\ell]$:
$Rel_{S'A'}[\ell]\ \overrightarrow{coh_\ell}\ Rel'_{S''A''}[\ell]\ \overrightarrow{coh_\ell}\ Acq_{S,A}[\ell])$

Next Acquire on ℓ: Given a release memory action, $Rel_{S,A}[\ell]$, return the next future acquire memory action, $Acq_{S'A'}[\ell]$, in $\overrightarrow{coh_\ell}$:

$nextAcquire(Rel_{S,A}[\ell]) = Acq_{S'A'}[\ell]$:
$(Rel_{S,A}[\ell]\ \overrightarrow{coh_\ell}\ Acq_{S'A'}[\ell]) \wedge (\nexists Acq'_{S''A''}[\ell]$:
$Rel_{S,A}[\ell]\ \overrightarrow{coh_\ell}\ Acq'_{S''A''}[\ell]\ \overrightarrow{coh_\ell}\ Acq_{S'A'}[\ell])$

Promote a Scope: Given scopes S' and S:

$$\begin{cases} promote(S',S) = S, & if\ S' \subseteq S \\ promote(S',S) = S', & if\ S \subseteq S' or\ S \cap S' = \emptyset \end{cases}$$

II. New Definitions for Scope Promotion

Remote Acquire: Given a remote acquire memory action, $RmAcq_{S,A}[\ell]$, and the most recent release memory action on ℓ, $Rel_{S'A'}[\ell] = lastRelease(RmAcq_{S,A}[\ell])$, replace $Rel_{S'A'}[\ell]$ with $Rel_{promote(S',S),A'}[\ell]$.

Remote Release: Given a remote release memory action, $RmRel_{S,A}[\ell]$, and the next future acquire memory action on ℓ, $Acq_{S'A'}[\ell] = nextAcquire(RmRel_{S,A}[\ell])$, replace $Acq_{S'A'}[\ell]$ with $Acq_{promote(S',S),A'}[\ell]$.

Figure 13. Scope promotion for HRF-indirect-relaxed.

synchronized scope promotion, this is the case. Remote operations (e.g., $RmAcq_{S',A'}[\ell]$) are scoped to encompass the releasing scope, S.

Finally, the third condition says that the release, $Rel_{S,A}[\ell]$, must execute before the acquire, $Acq_{S'A'}[\ell]$, in the total order of all operations on location ℓ (called the coherence order, or $\overrightarrow{coh_\ell}$). This means that a work-item must release a location before another work-item can acquire it. The same is true for scope promotion—a work-item must release a location before another work-item can remotely acquire it.

Thus, we add promotion to HRF-indirect-relaxed (Figure 13) to formalize remote synchronization.

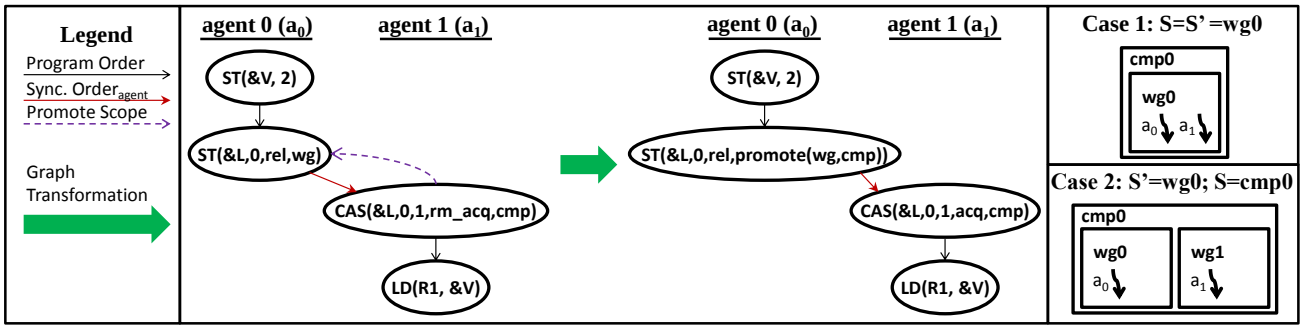

Figure 14. Example synchronization orders.

References

[1] "OpenCL 2.0 Reference Pages." [Online]. Available: http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/.

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

[3] "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.

[4] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," In *Proceedings of the Conference on High Performance Graphics*, New York, N.Y., USA, 2009, pp. 145–149.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, New York, N.Y., USA, 1998, pp. 212–223.

[6] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.0," [Online]. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[7] "Intel Threading Building Blocks." [Online]. Available: http://www.threadingbuildingblocks.org/.

[8] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 227-242, 2009.

[9] 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

[10] 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.

[11] 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.

[12] AMD, "Southern Islands Series Instruction Set Architecture," 2012.

[13] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: x86-TSO," In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, 2009.

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

[15] 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," *presented at the 20th IEEE International Symposium On High Performance Computer Architecture (HPCA-2014)*.

[16] N.S. Arora, R.D. Blumofe, and C. Greg Plaxton, "Thread scheduling for multiprogrammed multiprocessors," In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, ACM, Puerto Vallarta, Mexico, 1998, pp. 119–129.

[17] 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.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," In *SIGARCH Computer Arch. News*, vol. 39, no. 2, pp. 1-7, Aug. 2011.

[19] 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*, Sept. 2013.

[20] DIMACS Implementation Challenges. http://dimacs.rutgers.edu/Challenges/

[21] Web resource: http://www.sommer.jp/graphs/

[22] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway distributed shared memory system," In *Proc. 38th IEEE Computer Society Int. Conf.*, pp. 528-537, 1993.

[23] L. Iftode, J. P. Singh, and K. Li, "Scope consistency: a bridge between release consistency and entry consistency," In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, p.277-287, June 24-26, 1996, Padua, Italy.

[24] D. Dice, M.S. Moir, and W.N. Scherer III, "Quickly reacquirable locks," *US Patent 7,814,488*, 2010.

[25] W.W.L. Fung and T.M. Aamodt, "Energy Efficient GPU Transactional Memory via Space-Time Optimizations," In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, pp. 408-420, Davis, CA, Dec. 7-11, 2013.

[26] D. Cederman, P. Tsigas, and M.T. Chaudhry, "Towards a Software Transactional Memory for Graphics Processors," In *Proceedings of the 10th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV 2010)*.

[27] I. Singh, A. Shriraman, W.W.L. Fung, M. O'Connor, and T.M. Aamodt, "Cache Coherence for GPU Architectures," In *Proceedings of the 19th IEEE International Symposium on High-Performance Computer Architecture (HPCA-19)*, pp. 578-590, Shenzhen, China, Feb. 23-27, 2013.

[28] S. Tzeng, A. Patney, and J.D. Owens, "Task Management for Irregular-Parallel Workloads on the GPU," In *Proceedings of High Performance Graphics 2010*, pp. 29–37. June 2010.