This paper appears in the Workshop on Memory Systems Performance and Correctness (MSPC)
Seattle, WA June 2013

Sequential Consistency for Heterogeneous-Race-Free

Programmer-centric Memory Models for Heterogeneous Platforms

Derek R. Hower[†] Bradford M. Beckmann[†] Benedict R. Gaster[†] Blake A. Hechtman[†‡]. Mark D. Hill[*†] Steven K. Reinhardt[†] David A. Wood[*†]

[†]AMD Research [*]University of Wisconsin Dept. of Computer Sciences [‡]Duke University Dept. of Electrical and Computer Engineering

{derek.hower, brad.beckmann, benedict.gaster, steve.reinhardt}@amd.com {markhill, david}@cs.wisc.edu blake.hechtman@duke.edu

Abstract

Hardware vendors now provide heterogeneous platforms in commodity markets (e.g., integrated CPUs and GPUs), and are promising an integrated, shared memory address space for such platforms in future iterations. Because not all threads in a heterogeneous platform can communicate with the same latency, vendors are proposing synchronization mechanisms that allow threads to communicate with a subset of threads (called a scope). However, vendors have yet to define a comprehensive and portable memory model that programmers can use to reason about scopes. Moreover, existing CPU memory models, such as Sequential Consistency for Data-Race-Free (SC for DRF), are ill-suited, in part, because they define all synchronization operations globally and preclude low-energy, high-performance local coordination.

Towards this end, we embrace scoped synchronization with a new class of memory consistency models: Sequential Consistency for Heterogeneous-Race-Free (SC for HRF). Inspired by SC for DRF (C++, Java), the new models provide programmers with SC for programs with "sufficient" synchronization (no data races) of "sufficient" scope. We develop the first such model, called HRF0, show how it can be used to develop high-performance code, show example hardware support, and motivate future work.

1. Introduction

Heterogeneous systems, such as those containing graphics processing units (GPUs), are often organized in a hierarchy for performance reasons. This hierarchy is exposed directly to software by programming models like OpenCL™ [27] and CUDA [29], which bundle threads into tightly coupled groups called workgroups (OpenCL) or blocks (CUDA). On these systems, communication among threads in the same group is faster and more efficient than communication among threads in different groups [27, 29].

For this reason, high-performance code is written in a group-centric manner. For example, consider the code in Figure 1, which represents a common method of performing a stencil computation on a GPGPU [8, 26]. Rather than requiring inter-group communication at every timestep, as

```
01: while t < num_timesteps: /* Global loop */
02:    G times do: /* Local (group) loop */
03:       grid[id_x][id_y] = f(neighbors)
04:       barrier(threads in group)
05:    t += G
06:    barrier(all threads)
07:    read ghost zone values from neighbors
```

Figure 1. Pseudo-code for a single thread in a stencil "ghost zone" GPGPU application.

is common in CPU-based stencil implementations [22], this application performs a small amount of redundant computation so multiple timesteps can be computed within a group before global synchronization is needed.

In this method, called "ghost zones" [26] or "overlapped tiling" [22], a 2-D problem space is broken into MxM partitions that each group is responsible for in the final output. However, each group actually computes points on a (M+2G)x(M+2G) partition, where points outside the main MxM partition are redundant copies of points owned by another group and are said to lie in the ghost zone. By using the ghost zone, a group can perform G timesteps locally without needing inter-group communication. Beyond G timesteps, the points in the ghost zone are all invalid, requiring a global barrier and a refresh of the ghost zone values before proceeding.

How does a programmer ensure that the code in Figure 1 is synchronized correctly? In particular, how can a programmer be sure that the code in line 07 actually reads the updated ghost zone values (i.e., the neighbor's updates are not still in a private cache)? In current systems, the only option is to develop an understanding of the hardware-centric memory models used by languages like CUDA and OpenCL, or, if you enjoy the nuanced art of assembly programming, the similarly hardware-centric memory model of intermediate representations like PTX [30]. However, these hardware-centric models can be difficult to comprehend, ambiguous, or both, especially for software-oriented programmers.

To shield programmers from the complexities of hardware-centric memory models, many successful high-level languages adopt programmer-centric memory models. For example, both C++ [4] and Java [24] have adopted memory models from the class of Sequential Consistency for Data-Race-Free (SC for DRF) [2]. With SC for DRF, programmers can reason in terms of the intuitive sequential con-

Work by Hill and Wood was performed while consulting for AMD Research. Work by Hechtman was performed while on internship at AMD Research.

sistency model as long as they can ensure their code is correctly synchronized and free of data races.

Unfortunately, existing SC for DRF models are not a good match for heterogeneous systems like GPUs because they are defined in terms of a single, global order of synchronization. In an SC for DRF model, memory operations (logically) must become visible to all threads in the system at the same time. In a GPU, this means that *any* synchronization, even if it is local to a group, must by definition result in global visibility; in Figure 1, for example, all prior updates before the group barrier on Line 04 must be globally visible, even though it clear that full global visibility at that point is not required by the algorithm.

To resolve the conflict between high-performance synchronization in heterogeneous systems and programmer-centric memory models common in homogeneous systems, we propose a new class of memory models called Sequential Consistency for Heterogeneous-Race-Free (SC for HRF). In SC for HRF, all synchronization operations occur with respect to a subset of threads in an execution called a scope. When threads synchronize with a scope, they indicate that the synchronization effects -- including memory ordering -- can be limited to other threads in that scope. For example, we might say that the barrier on Line 04 in Figure 1 performs with respect to group scope and indicates that memory operations should be visible within, but not necessarily beyond, the local thread group.

Like the class of SC for DRF models from which we take inspiration, SC for HRF models allow programmers to reason in terms of sequential consistency so long as the programs they write are free of heterogeneous races. Intuitively, a heterogeneous race occurs if either (a) two conflicting (same address and at least one is a write) memory operations are not separated by any synchronization (*á la* a data race) or (b) the synchronization used is not performed with respect to sufficient scope, where sufficiency is determined by the specific SC for HRF model being used.

In this paper, we propose the first such model called HRF0. In HRF0, if two threads communicate, they must synchronize using operations of identical scope. We provide a formal definition of this first model, show how it can be used to write high-performance code, and show a basic hardware implementation. However, we also show how HRF0 may be limiting to software, thus hinting at the need for further investigation into alternative SC for HRF formalizations.

In summary, we make the following contributions:

- We observe that existing memory consistency models are ill-suited for future heterogeneous platforms that will use a single shared address space while still providing the ability to coordinate and synchronize locally.
- We propose a class of SC for HRF memory models that guarantee a sequentially consistent execution for any program that ensures all conflicting data accesses are coordinated with sufficient scoped synchronization.

We formally define the first SC for HRF model, HRF0, and show how it can be used and how hardware can support it, and motivate future work.

Like key SC for DRF papers [1, 4, 24], we focus on correctness and do not provide simulation results.

```
     t1              t2                    t3
ST X = 1
__syncthreads
                /* group sync */
                __syncthreads
                LD X (1)

                /* global sync */
                __threadfence_system
                ATOMIC_ST Y = 2
                                       LD Y (2)
                                       LD X (?)
```

Figure 2. Ambiguous behavior in CUDA. Assume all locations initially hold the value 0, threads t1 and t2 belong to the same block, and t3 is in a different block. Will the LD X on t3 see the value of ST X from t1? In our view, that is open to interpretation based on the published memory model. In current hardware we believe the answer is "yes," though that may change in future generations.

2. Background and Related Work

Although we believe SC for HRF models will be useful more generally for systems with heterogonous components, we focus on general-purpose GPUs (GPGPUs) in this paper due to their growing relevance and to make the discussion more concrete. This section discusses the state of the art in memory models for both GPGPUs and CPUs and describes why we believe none is completely appropriate for future systems.

2.1 Current GPGPU Models

Current GPGPU programming models group threads in several ways. First, a group of threads is bundled into a 64-thread wavefront (AMD) or 32-thread warp (NVIDIA) that execute together in lockstep on SIMD hardware. Second, wavefronts are grouped into workgroups (AMD) or blocks (NVIDIA). All threads in a workgroup execute concurrently and share resources, including a group memory (AMD) or shared memory (NVIDIA) that is accessible only by threads in the workgroup/block.

Both OpenCL and CUDA provide synchronization operations for threads communicating within a workgroup/block [3, 28] (i.e., scoped synchronization operations). In OpenCL 1.x, threads in a workgroup can synchronize via a barrier, which also acts as a workgroup memory fence such that all operations before the barrier are guaranteed to have completed before any thread leaves the barrier and any operation after the barrier is guaranteed not to be moved ahead of the barrier. Inter-workgroup communication is undefined behavior in OpenCL, though, as we will detail, that has not stopped programmers from writing code that uses workgroup communication based on knowledge of microarchitectural details.

2.1.1 Example of Ambiguity in Current Models

CUDA has a barrier operation for threads in the same block called `__syncthreads` that is similar to the OpenCL barrier. In addition, CUDA also provides memory-fence

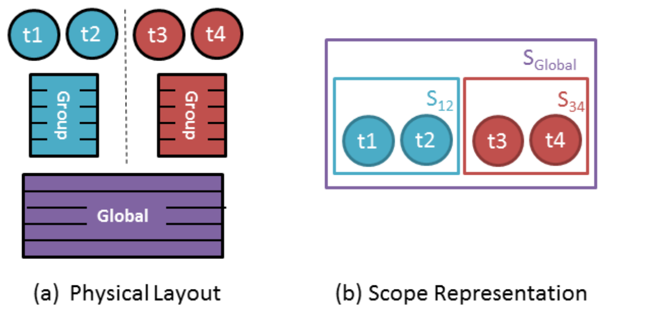

Figure 3. Simple baseline system.

(called `__threadfence`) operations for the device (GPU) and system (GPU + CPU) scopes. CUDA, like OpenCL, defines the semantics of these operations in a hardware-centric manner. This can be difficult to understand for software-oriented programmers and also ambiguous in some corner cases.

We show an example of ambiguous CUDA behavior in Figure 2. In it, two threads in the same block (t1 and t2) first synchronize with each other using the block-barrier operation `__syncthreads`. Later, thread t2 synchronizes with the entire system using `__threadfence_system`. After t3, on a different block, observes the atomic store performed by t2 (which cannot bypass the threadfence), the question is: *Does t3 observe the original store to X by t1?*

We can find no clear answer based on the CUDA memory model. From the documentation, we know that before completing a `__threadfence_system`, a thread "waits until all global and shared memory accesses made by the calling thread prior to `__threadfence_system` are visible to … all threads in the device for global memory accesses." Does this mean that by performing a load of X before the threadfence, t2 makes the value produced by t1 visible to t3? We are confident, given knowledge of the current hardware implementation, that the answer is "yes," but have less confidence that is the intent of the memory model or that the answer will remain the same in future hardware generations.

2.1.2 Programmers Desire More

While the previous example may seem contrived, there is ample evidence that programmers are pushing the boundaries of GPGPU programming models despite the presence of undefined and/or ambiguous behavior. For example, the popular persistent threads programming paradigm [14] uses knowledge of how workgroups are scheduled together on a GPU to perform global communication among thread groups, even though (especially in OpenCL) global communication among thread groups is officially unsupported.

Because of this push for more general programming, we think it is important to develop a rigorously defined and easy to comprehend memory model that can serve as a guideline for writing portable code. We believe the class of SC for HRF models can fill that role.

2.2 Current and Future GPU Caches

Historically, GPUs contain at least two types of local memories (some have more, but for simplicity we omit them here). First, GPUs have a software-managed scratchpad that is addressable only by threads in a workgroup/block. Second, they contain hardware-managed caches that hold addresses in a global memory space that can be used by all threads on the device. As a result, threads manage data in multiple, disjoint address spaces, and, notably, must explicitly copy data to/from the group scratchpad space.

However, vendors have started to support more general memory models with a single address space. For example, NVIDIA now provides a software-managed L1 cache in the global address space that can be used by threads in a block [28] and AMD, in consortium with other companies in the Heterogeneous System Architecture (HSA) Foundation, has indicated support for shared virtual memory with a flat address space [17]. In addition, both have started to introduce features that will allow more fine-grained sharing among GPU threads [19]. Ultimately, these changes are aimed at making GPGPUs more usable for workloads that may not be as embarrassingly parallel as graphics.

Even with a shared address space and a move towards more general programming models, some remnants of the thread groups are likely to persist because they are useful for graphics (and at the end of the day, GPUs do graphics). In particular, we expect that the current thread-grouping into wavefront/warp and workgroup/block will persist. We also expect that caches, though they may be unified under a single address space, will not be invisible to software. In CPUs, software does not need to manage caches for correctness or performance reasons because they are managed by the hardware cache-coherence protocol. Providing similar read-for-ownership coherence on GPUs seems unlikely (see Keckler et al. [19] for a good overview of why), and so we expect that GPU caches will continue to behave like high-throughput write-through/write-coalescing caches. When synchronizing, these caches are typically flushed/invalidated to ensure that memory ordering constraints are met. Because these flushes/invalidates can be long-latency events, GPGPU models provide scoped synchronization primitives that permit synchronization to complete before those flushes reach all the way to main memory (and, for example, only need to reach a shared cache). Because it is likely this cache design will persist, we expect software will still have some responsibility for cache management (e.g., by selecting the appropriate synchronization scope).

2.3 Sequential Consistency for Data-Race-Free

Sequential consistency guarantees that the observed order of all memory operations is consistent with a theoretical execution in which each instruction is performed one at a time by a single processor [23]. SC preserves programmer sanity by allowing them to think about their parallel algorithm in sequential steps. Unfortunately, though true, SC can be difficult to implement effectively without sacrificing performance or requiring deeply speculative execution [13]. As a result, most commercially relevant architectures, run times, and languages use a model weaker than SC that allows certain operations to appear out of program order at the cost of increased programmer effort.

To bridge the gap between the programming simplicity of SC and the high performance of weak models, the SC for DRF class of models exist and guarantee an SC execution only in the absence of data races. Without data races, the system is free to perform any reordering that would not cause an observable violation of SC. Models in the class differ on the defined behavior in the presence of a data race, and vary from providing no guarantees [2, 4] to providing weak guarantees like write causality [24].

In SC for DRF models, memory accesses are grouped into one of two categories: data or synchronization. Synchronization accesses are ordered sequentially with respect to one another (i.e., they form a total global order). A data race occurs if two conflicting (same address, different threads, at least one is a write) data accesses perform and are not separated by a synchronization access. Said another way, a program contains a data race if, in some sequentially consistent execution, it is possible for two conflicting data accesses to appear next to each other in the total memory access order.

One downside of the SC for DRF models is that racey software has undefined behavior. This can be especially problematic in codes that use intentional (benign) data races or in codes containing unintentional bugs. To address this, Marino et al. proposed the DRFx model that, in addition to guaranteeing sequential consistency for data race-free programs, will raise a memory model exception when a racey execution violates sequential consistency [25]. To do so, the authors propose adding SC violation detection hardware similar to conflict detection mechanisms in hardware transactional memory proposals.

3. HRF0: The First Sequential Consistency for Heterogeneous Race-free Model

The existing SC for DRF models define a data race in terms of a single, global synchronization order; as a result, they are unable to take advantage of synchronization scopes available on heterogeneous platforms. To address this gap, we propose a new class of memory consistency models called sequential consistency for heterogeneous race-free. In an SC for HRF model, all synchronization occurs with respect to a subset of threads in an execution called a scope. Practically, scopes can be defined to reflect the capabilities of a system. In a GPGPU, there will likely be one scope for each thread group sharing a memory.

Scopes can (and will) overlap with one another. For instance, the scope containing a GPU thread group will overlap with at least one other scope representing all threads in the execution. As such, threads will have to decide which scope to use when synchronizing. Generally, they will want to choose the smallest scope possible that includes all threads involved in the communication. More specifically, the choice of scope will depend on properties of the specific SC for HRF model that regulate how synchronization from different scopes can interact with one another.

3.1 Baseline System and Architecture

Before diving into the specifics of the HRF0 model, we will first lay down the basic assumptions about the hardware and language/ISA of our target system. We describe HRF0 in terms of a simple system first and leave generalization to more complex systems to future work.

```
while t < num_timesteps: /* Global loop */
    acquire(Global)
    read ghost zone values from neighbors
    G times do: /* Local (group) loop */
        acquire(Group)
        grid[id_x][id_y] = f(neighbors)
        release(Group)
        barrier(threads in group)
    t += G
    release(Global)
    barrier(all threads)
```

Figure 4. Pseudo-code for a single thread in a stencil "ghost zone" GPGPU application.

Our basic target system contains four threads and two thread groups, as shown in Figure 3(a). Threads in a group (t1/t2 or t3/t4) share a local cache that is backed up by a larger, globally visible cache. All caches share the same address space but are incoherent. Throughout this paper, we will represent the system in Figure 3(a) using the scope representation in Figure 3(b), which illustrates that threads t1 and t2 (t3 and t4) belong to both a group scope S_{12} (S_{34}) and a global scope S_{Global}.

For synchronization, we assume that the language and/or ISA (the HRF0 model could represent either) provides acquire and release operations with semantics similar to the operations of the same name in systems implementing release consistency [11]. We chose acquire/release synchronization, rather than fence operations found in current GPGPU models, for two reasons. First, as Adve and Hill have shown [1], acquire/release can be generalized to other synchronization primitives, making our analysis compatible with other synchronization methods. Second, the HSA foundation has indicated that future HSA-compliant devices will use acquire/release synchronization [17]. Like the HSA operations, we assume that acquire and release always perform with respect to a particular scope and may not always result in global visibility.

3.2 HRF0 Synchronization Model

We define the HRF0 synchronization model based on the constraint that if two threads communicate, they synchronize using operations of the *exact same* scope. If -- in some sequentially consistent execution of a program -- two conflicting data accesses are performed without being separated by paired synchronization of identical scope, then those accesses form a heterogeneous race. HRF0 provides considerable hardware implementation flexibility, as we discuss in Section 4.

To synchronize correctly in HRF0 while still getting good performance, programmers can follow a simple rule: *Always use the smallest scope available that includes all threads that may see the values that will be written*. One important caveat to this rule is that all threads must use the same scope. Thus, if a group of threads is cooperating to perform work locally, when that work needs to be communicated to threads outside the group, *all* of the group threads must perform a larger scope synchronization. Luckily, in existing GPGPU programming models in which threads execute in lockstep, this constraint may not be as limiting as it first appears.

Figure 4 shows the code from Figure 1 correctly synchronized for HRF0 implementation. We show the acquire/release actions asfl explicit calls, though in an actual implementation they may be combined with the semantics of the barriers.

When iterating locally in the inner loop, threads avoid slow global communication by synchronizing with respect to their group scope and update only group memory. At the end of a timestep, all the threads perform a global synchronization to communicate boundary conditions with other groups and (logically) flush all updates to global memory. In the example, the program still would have been correct if the local synchronization performed conservatively with respect to the global scope. This important property of HRF0 makes it possible for users to adopt the model incrementally by starting with an SC for DRF-compatible program and later using scopes to improve performance.

3.3 Happens-before Order

While many programmers can successfully use their intuition to write HRF0 programs correctly, experience shows that complicated corner cases need robust formalism. Towards that goal, in this subsection we describe a happens-before relation that can be used to reason rigorously about heterogeneous races.

In HRF0, whether or not an operation happens before another -- and, consequently, whether it forms a heterogeneous race -- depends on the how operations are related through scope synchronization. Abstractly, one can think of there being a separate order of operations for each scope in an HRF0 execution (in contrast to a single global order in SC for DRF). Operations are ordered in a scope if they are ordered by the transitive closure of (a) program order, and (b) synchronization order of acquire/release operations *in the scope in question*.

Ultimately, given the observed scope orders for an execution, we can say that an operation A happens before an operation B if A appears before B in *any* scope order (i.e., are ordered by synchronization within any single scope).

We show an example of this abstract model in Figure 5. On the left is an execution order observed during a run of the program; on the right are the three happens-before scope orders relating instructions from different threads as they appear when the program completes. In the example, we observe that the release ② provides its value to release ③ in scope S_{12}'s order and release ⑥ provides its value to release ⑦ in scope S_{Global}'s order. When transitively combined with program order dependencies, all other instructions are ordered as shown in Figure 5.

Based on those scope orders, we show examples of both correctly synchronized conflicts and a conflict that forms a heterogeneous race. First, in this execution, both the conflicting pairs ❶-❹ and ❺-❽ are synchronized properly because they are ordered by scope orders S_{12} and S_{Global}, respectively. On the other hand, the conflicting pair ❶-❾ forms a heterogeneous race in HRF0 because no scope order contains both operations. Thus, in the execution, the value observed by the load in instruction ❾ is undefined.

3.4 Formal Definitions

We formally define the SC for HRF0 model using the set relational notation first adopted by Adve and Hill [2]:

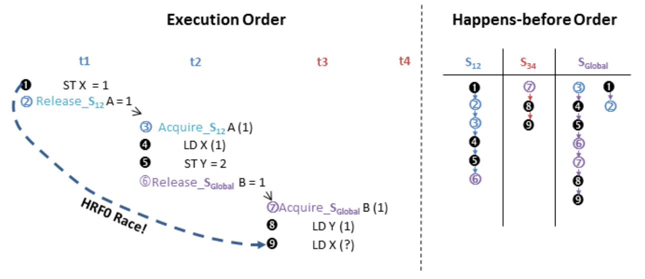

Figure 5. Example of correct synchronization and a race in HRF0. Note that happens-before is a partial order.

Program Order $(\overrightarrow{po})$***:*** op1 $\overrightarrow{po}$ op2 iff op1 and op2 are from the same thread and op1 completes before op2.

Scope Synchronization Order $(\overrightarrow{so_S})$***:*** Release op1 appears before release or acquire op2 in $\overrightarrow{so_S}$ iff both are performed with respect to scope S, access the same location, and op1 occurs before op2.

Heterogeneous Happens-before 0 (hhb0): The union of the irreflexive transitive closures of all scope synchronization orders with program order:

$$\bigcup_{\forall \mathbb{S}} (\overrightarrow{po} \cup \overrightarrow{so_S})^{+}$$

In this equation, $\mathbb{S}$ represents the set of all scopes in an execution. In this equation, the closure applies only to the inner union and is not applied to the outer union. Heterogeneous Happens-before forms a partial order of execution.

Conflicting Operations: Two operations op1 and op2 conflict iff both are to the same address and at least one is an ordinary store or a release.

Heterogeneous Race: A pair of conflicting operations op1 and op2 forms a heterogeneous data race iff they are not ordered by hhb0.

Heterogeneous-Race-Free-0 (HRF0): An execution is Heterogeneous Race-free-0 iff there are no heterogeneous races. A program is HRF0 iff all possible sequentially consistent executions of the program are HRF0.

Sequential Consistency for Heterogeneous-Race-Free-0 (SC for HRF0): A system implementation obeys the SC for HRF0 memory model iff all executions of an HRF0 program on the system are sequentially consistent.

3.5 Analysis

Relationship to SC for DRF: When defining an SC for HRF model, we do not aim to enable new functionality or programming idioms beyond what is possible with existing DRF models. We do, however, aim to open new possibilities for increased performance in systems in which threads can synchronize with each other with unequal effort while giving programmers the tools they need to create correct code. Towards this end, we believe all SC for HRF models should obey two guidelines, which are met in HRF0:

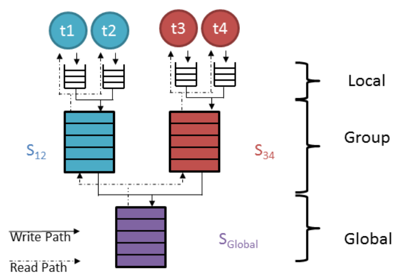

Figure 6. Hardware in the example implementation.

1. An SC for HRF model should be equivalent to DRF models in the degenerate case of only one global scope.
2. A system should be free to synchronize with larger (more inclusive) scopes than is specified by software at any time.

In other words, we believe that a DRF program should run correctly on HRF hardware and an HRF program should always run correctly on DRF hardware (on which all synchronization implicitly is promoted to global scope). This ensures an easy adoption path to HRF; existing DRF software will continue to work, and programmers can introduce scoped synchronization as needed for performance.

Synchronization Races: In HRF0, it possible for two synchronization accesses (i.e., an acquire and release) to be unordered and thus form a heterogeneous race *with each other*. This is a foreign concept in SC for DRF models because there is by definition a single order of all synchronization accesses. Synchronization races can happen in HRF0 because the model allows synchronization accesses to perform locally in a scope without having to wait to be ordered with other scopes. Thus, if pairs of threads from different scopes are synchronizing with each other using the same location but different scopes, then those synchronization accesses race with each other.

Luckily, software can avoid this complexity by following a simple best practice: *When using HRF0, software should associate a single scope with each synchronization variable*. For example, software could be constructed so variable A is always used with scope S_{12}, B with S_{34} etc. If this practice is followed, synchronization operations will never race with each other (though races between data accesses can certainly still occur).

4. Example Implementation

In this section we describe a reference GPU memory implementation that is compatible with HRF0. We design the implementation for a baseline like the one in Figure 3. In this system, groups of threads share a fast cache memory through which they can communicate through with each other. Further, there is a slower global memory that can be used to hold data that does not fit in the group memory and communicate between threads of different groups. For simplicity, assume the global memory is DRAM, though it could internally also contain other caches.

One can think of each of the memories in the baseline system as belonging to exactly one of the scopes in Figure 3. In particular, the memory on the bottom belongs to the global scope and the cache memories above that belong to a group (i.e., S_{12} or S_{34}) scope. Additionally, in Figure 6, we show write buffers (commonly called write-combining buffers in GPUs) between the threads and group memory. These write buffers can be considered, for purposes of the following description, part of an implicit local scope that is smaller than group scope and that contains only one thread. In an actual implementation, there may also be write buffers between group and global memory, but because they would logically belong to the group memory anyway, we omit them for simplicity of exposition.

4.1 Invariants

The implementation obeys the HRF0 model by maintaining three invariants:

1. Before a release completes, all prior writes have been written into a memory belonging to the scope of the release.
2. Before an acquire completes, any memory that both belongs to a scope containing the thread and is part of a scope smaller than the scope of the acquire has been invalidated.
3. An acquire or release completes when it reads or writes to a memory belonging to the target scope.

Together, these invariants ensure the HRF0 requirement that if any load and store are separated by a paired acquire/release to the same scope, the store will be visible to the load. This is because the release will ensure that the stored value is visible in the scope of the synchronization by flushing dirty values. Likewise, the acquire will ensure that the thread performing the load will fetch that value from the scope of the synchronization by invalidating all lower scope memories.

4.2 Basic Operation

The system maintains the invariants listed in Section 4.1 by taking the following actions on memory operations:

Load: Read directly from group memory unless the address is present in the local write buffer, in which case bypass from the local write buffer. If the location is not present in group memory, fetch it from global memory.

Store: Insert the write into the local write buffer.

Group Release: Empty the local write buffer, then complete the store in group memory.

Group Acquire: Complete the load in group memory and empty the local write buffer before the next load or store performs.

Global Release: Flush all dirty data in both the local write buffer and group memory to global memory. When those are completed, perform the store in global memory.

Global Acquire: Flush all dirty data and invalidate all valid data in both the thread's write buffer and the group memory. Perform the load in global memory.

An implementation could support the release/acquire actions in a variety of ways. In its most basic form, controllers could walk the caches to locate all dirty and/or valid data. More sophisticated methods are possible, including those that use hardware support to intelligently flush/invalidate caches [15, 31] and those that use software/compiler support to get the same result [9].

4.3 Possible Optimizations

While the basic operation outlined in Section 4.2 is correct and good for developing a high-level understanding, it leaves some possible optimizations on the table.

For example, on a global acquire, the system does not have to wait for group memory to flush/invalidate before proceeding; rather, it needs only to ensure that any subsequent load or store in program order will not perform ahead of any flush/invalidate caused by the acquire (as previously observed for single-scope models [11]). Also, if local memories are managed as write-through caches (as is typical in GPU streaming caches), then the flush actions only have to ensure that the write buffer has emptied.

In HRF0, an implementation also can selectively flush/invalidate memories on acquire and release operations rather than using the blunt-force method in the example. If an implementation had the ability to distinguish reads and writes from different threads (e.g., using thread ID tags), the flush/invalidate actions could be filtered to affect only the locations touched by the synchronizing thread.

4.4 Analysis

Recall that if the system had to support a single-scope DRF model for the memory layout like the baseline in Figure 6, the implementation would presumably either have to perform expensive local memory flush/invalidates on every acquire or release, or -- to avoid the flush/invalidates -- implement a read-for-ownership coherence protocol. The performance of the first option is likely to be poor enough to discourage fine-grained synchronization. On the other hand, while the second option may perform better, some believe that implementing a coherence protocol on a GPU would be prohibitively complex and/or expensive [19, 21]. With those considerations, the HRF0 implementation seems to be a reasonable alternative.

5. Discussion

Hierarchical Scoping: In HRF0, threads must synchronize with each other using identical scope. While this is supported by our vision of future GPU hardware, it may also be overly conservative. The caches in future GPUs will likely be hierarchical, as they are in our example implementation. With this hierarchy, hardware could support an SC for HRF model that permits thread synchronization using different, but overlapping, scopes. For example, the hardware in Figure 6 would support synchronization between one thread in a group that releases to global scope and another thread in the same group that acquires from group scope. Synchronization among different scopes may not generalize, however, and requires careful consideration.
Dynamic Scoping: So far, we have assumed that scopes are defined statically by the system to reflect an implementation's memory hierarchy. In this case, the specific scopes available to a thread will depend on where it executes (i.e., what thread group it belongs to). This decision works well when the programming model reflects the physical resources, as in current GPGPU languages.

However, if an SC for HRF model were applied to a more general programming model like C++, using static scoping could be too restrictive. In that case, it is possible to allow scopes to be defined dynamically by binding threads that communicate frequently together at run-time. If this is allowed, the application will likely need a run-time layer than maps software-defined scopes to physical resources in the system. For example, a run-time thread scheduler could map threads in the same dynamically defined scope to the same thread group on a GPU.
DRF + Scope -- Another View: While we have presented SC for HRF as a new class of memory models, an alternative view is the SC for HRF models are SC for DRF with races defined to include scope. Both are valid; we choose to present this work as a new model to emphasize the fact that synchronization scopes introduce an entirely new class of races that do not exist in prior synchronization models.
Working with CPU Threads: In our descriptions so far, we have ignored the presence of cache-coherent CPU threads that may share a global scope with heterogeneous threads. The CPU threads could complicate the system because, in most cases, accesses emanating from CPU threads will need to respect the semantics of a legacy memory model (e.g., x86-TSO). Luckily, though, the SC for HRF models are weak enough that most existing CPU memory models will be strictly stronger and the two can safely co-exist.
Dealing with Races: The HRF0 model does not define semantics in the presence of a heterogeneous race. There are several alternate options a model could choose to handle behavior in racey programs. First, any SC for HRF model can support a memory model exception that is raised if an implementation detects that sequential consistency is violated, *á la* DRFx [25]. To ease implementation complexity, we advocate making the exception semantics best-effort, so doing nothing is a valid implementation. Second, like the Java memory model, an SC for HRF could provide some basic guarantees like write causality for racey code. Any such guarantees should be thoughtfully considered, however, because previous efforts have proven more difficult than first imagined [32].

6. Related Work

Hechtman and Sorin have recently posited that GPU systems should implement sequential consistency, and have shown that the performance of an SC GPU is comparable to one implementing a weak model [16]. Their analysis, however, assumes a baseline GPU that implements read-for-ownership coherence and does not take interaction with a CPU core into account. In this paper, we make different assumptions, and propose SC for HRF as a class of models to reason about consistency in current and future GPUs.

Scoped synchronization is not limited to just GPU systems. The Power7 CPU system uses scoped broadcasts in its coherence protocol [18]. Though this scoping is not exposed to the programmers, it nonetheless represents the trend toward scoped synchronization in modern hardware.

One could argue that message-passing models like MPI provide scoped consistency by allowing threads to explicitly specify senders and receivers [12]. Of course, message-passing models do not use shared memory, and as a result are difficult to use with algorithms involving pointer-based data structures like linked lists. In addition, shared memory programs are easier for compilers to optimize because in

MPI, compilers must have semantic knowledge of the API to perform effective operation reordering [10].

Recently, there has been an effort in the high-performance community to push programming models that make use of a partitioned global address space (PGAS). These include languages like x10 [7], UPC [5], and Chapel [6]. Like SC for HRF models, these PGAS languages present a single shared address space to all threads. However, not all addresses in that space are treated equally. Some addresses can be accessed only locally while others can be accessed globally but have an affinity or home node. As a result, PGAS programs still explicitly copy data between memory regions for high performance. In contrast, in an SC for HRF model, a particular address is not bound to a home node and there is no need for application threads to copy data explicitly between memory regions.

There has recently been work to on coherence alternatives for shared memory in GPU architectures. Cohesion is a system for distinguishing coherent and incoherent data on GPU accelerators [20]. The incoherent data has to be managed by software with explicit hardware actions like cache flushes. SC for HRF models, on the other hand, abstract away hardware details for programmers and rely on an implementation to manage memory resources.

7. Conclusions

In this paper, we present a new class of consistency models called sequentially consistent for heterogeneous race-free that allow programmers to reason about scoped synchronization present in heterogeneous systems. We present the first SC for HRF model, called HRF0, that requires correct synchronization to use identical scope. We have shown how programmers can use HRF0 to build correct high-performance software and how designers can build hardware that supports the model. Our preliminary analysis shows that HRF0 may constrain software unnecessarily, and thus future work will investigate more permissive models and their effects on system designs.

References

[1] Adve, S.V. and Hill, M.D. 1993. A unified formalization of four shared-memory models. *Parallel and Distributed Systems, IEEE Transactions on*. 4, 6 (1993), 613–624.

[2] Adve, S.V. and Hill, M.D. 1990. Weak ordering—a new definition. *Proceedings of the 17th annual international symposium on Computer Architecture* (New York, NY, USA, 1990), 2–14.

[3] AMD 2012. *AMD Accelerated Parallel Processing OpenCL Programming Guide*.

[4] Boehm, H.-J. and Adve, S.V. 2008. Foundations of the C++ concurrency memory model. *PLDI* (Tuscon, AZ, Jun. 2008), 68–78.

[5] Carlson, W.W. et al. 1999. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses.

[6] Chamberlain, B.L. et al. 2007. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*. 21, 3 (2007), 291–312.

[7] Charles, P. et al. 2005. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices* (2005), 519–538.

[8] Che, S. et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009* (Oct. 2009), 44 –54.

[9] Choi, L. et al. 1996. Techniques for compiler-directed cache coherence. *IEEE Parallel Distributed Technology: Systems Applications*. 4, 4 (Dec. 1996), 23 –34.

[10] Danalis, A. et al. 2009. MPI-aware compiler optimizations for improving communication-computation overlap. *Proceedings of the 23rd international conference on Supercomputing* (2009), 316–325.

[11] Gharachorloo, K. et al. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proceedings of the 17th annual International Symposium on Computer Architecture* (1990), 376–387.

[12] Gropp, W. et al. 1999. *Using MPI: portable parallel programming with the message passing interface*. MIT press.

[13] Guiady, C. et al. 1999. Is SC+ILP=RC? *Proceedings of the 26th International Symposium on Computer Architecture, 1999* (1999), 162 –171.

[14] Gupta, K. et al. 2012. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. *Proceedings of Innovative Parallel Computing (InPar '12)* (May. 2012).

[15] Hammond, L. et al. 2004. Transactional memory coherence and consistency. *Proceedings of the 31st annual international symposium on Computer architecture* (2004), 102.

[16] Hechtman, B.A. and Sorin, D.J. 2013. Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors. *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)* (Tel Aviv, Israel, Jun. 2013).

[17] HSA Foundation 2012. *Heterogeneous System Architecture: A Technical Review*.

[18] Kalla, R. et al. 2010. Power7: IBM's next-generation server processor. *Micro, IEEE*. 30, 2 (2010), 7–15.

[19] Keckler, S.W. et al. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro*. 31, 5 (Oct. 2011), 7 –17.

[20] Kelm, J.H. et al. 2010. Cohesion: a hybrid memory model for accelerators. *Proceedings of the 37th annual international symposium on Computer architecture* (New York, NY, USA, 2010), 429–440.

[21] Kelm, J.H. et al. 2009. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *Proceedings of the 36th annual international symposium on Computer architecture Pages* (Austin, TX, Jun. 2009), 140–151.

[22] Krishnamoorthy, S. et al. 2007. Effective automatic parallelization of stencil computations. *ACM Sigplan Notices* (2007), 235–244.

[23] Lamport, L. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*. C-28, 9 (Sep. 1979), 690 –691.

[24] Manson, J. et al. 2005. The Java memory model. *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), 378–391.

[25] Marino, D. et al. 2010. DRFX: a simple and efficient memory model for concurrent programming languages. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), 351–362.

[26] Meng, J. and Skadron, K. 2009. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. *Proceedings of the 23rd international conference on Supercomputing* (2009), 256–265.

[27] Munshi, A. et al. 2011. *OpenCL programming guide*. Addison-Wesley Professional.

[28] Nickolls, J. et al. 2008. Scalable parallel programming with CUDA. *Queue*. 6, 2 (2008), 40–53.

[29] NVIDIA Corporation *CUDA 4.3 C Programming Guide*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[30] NVIDIA Corporation 2012. *Parallel Thread Execution ISA Version 3.1*.

[31] Pfister, G.F. et al. 1985. The IBM research parallel processor prototype (RP3): Introduction and architecture. *Proc. Int. Conf. Parallel Processing* (1985), 764–771.

[32] Pugh, W. 1999. Fixing the Java memory model. *Proceedings of the ACM 1999 conference on Java Grande* (1999), 89–98.