Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models

Matthew D. Sinclair† Johnathan Alsop† Sarita V. Adve†‡
† University of Illinois at Urbana-Champaign
‡ École Polytechnique Fédérale de Lausanne
hetero@cs.illinois.edu

ABSTRACT

As GPUs have become increasingly general purpose, applications with more general sharing patterns and fine-grained synchronization have started to emerge. Unfortunately, conventional GPU coherence protocols are fairly simplistic, with heavyweight requirements for synchronization accesses. Prior work has tried to resolve these inefficiencies by adding scoped synchronization to conventional GPU coherence protocols, but the resulting memory consistency model, heterogeneous-race-free (HRF), is more complex than the common data-race-free (DRF) model. This work applies the DeNovo coherence protocol to GPUs and compares it with conventional GPU coherence under the DRF and HRF consistency models. The results show that the complexity of the HRF model is neither necessary nor sufficient to obtain high performance. DeNovo with DRF provides a sweet spot in performance, energy, overhead, and memory consistency model complexity.

Specifically, for benchmarks with globally scoped fine-grained synchronization, compared to conventional GPU with HRF (GPU+HRF), DeNovo+DRF provides 28% lower execution time and 51% lower energy on average. For benchmarks with mostly locally scoped fine-grained synchronization, GPU+HRF is slightly better – however, this advantage requires a more complex consistency model and is eliminated with a modest enhancement to DeNovo+DRF. Further, if HRF's complexity is deemed acceptable, then DeNovo+HRF is the best protocol.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures – Cache memories; Shared memory; C.1.2 [**Processor Architectures**]: Single-instruction-stream, multiple-data-stream processors (SIMD); I.3.1 [**Computer Graphics**]: Graphics processors

Keywords

GPGPU, cache coherence, memory consistency models, data-race-free models, synchronization

DOI: http://dx.doi.org/10.1145/2830772.2830821

1. INTRODUCTION

GPUs are highly multithreaded processors optimized for data-parallel execution. Although initially used for graphics applications, GPUs have become more general-purpose and are increasingly used for a wider range of applications. In an ongoing effort to make GPU programming easier, industry has integrated CPUs and GPUs into a single, unified address space [1, 2]. This allows GPU data to be accessed on the CPU and vice versa without an explicit copy. While the ability to access data simultaneously on the CPU and GPU has the potential to make programming easier, GPUs need better support for issues such as coherence, synchronization, and memory consistency.

Previously, GPUs focused on data-parallel, mostly streaming, programs which had little or no sharing or data reuse between Compute Units (CUs). Thus, GPUs used very simple software-driven coherence protocols that assume data-race-freedom, regular data accesses, and mostly coarse-grained synchronization (typically at GPU kernel boundaries). These protocols invalidate the cache at acquires (typically the start of the kernel) and flush (writethrough) all dirty data before the next release (typically the end of the kernel) [3]. The dirty data flushes go to the next level of the memory hierarchy shared between all participating cores and CUs (e.g., a shared L2 cache). Fine-grained synchronization (implemented with *atomics*) was expected to be infrequent and executed at the next shared level of the hierarchy (i.e., bypassing private caches).

Thus, unlike conventional multicore CPU coherence protocols, conventional GPU-style coherence protocols are very simple, without need for writer-initiated invalidations, ownership requests, downgrade requests, protocol state bits, or directories. Further, although GPU memory consistency models have been slow to be clearly defined [4, 5], GPU coherence implementations were amenable to the familiar data-race-free model widely adopted for multicores today.

However, the rise of general-purpose GPU (GPGPU) computing has made GPUs desirable for applications with more general sharing patterns and fine-grained synchronization [6, 7, 8, 9, 10, 11]. Unfortunately, conventional GPU-style coherence schemes involving full cache invalidates, dirty data flushes, and remote execution at synchronizations are inefficient for these emerging workloads. To overcome these inefficiencies, recent work has proposed associating synchronization accesses with a scope that indicates the level of the memory hi-

erarchy where the synchronization should occur [8, 12]. For example, a synchronization access with a local scope indicates that it synchronizes only the data accessed by the thread blocks within its own CU (which share the L1 cache). As a result, the synchronization can execute at the CU's L1 cache, without invalidating or flushing data to lower levels of the memory hierarchy (since no other CUs are intended to synchronize through this access). For synchronizations that can be identified as having local scope, this technique can significantly improve performance by eliminating virtually all sources of synchronization overhead.

Although the introduction of scopes is an efficient solution to the problem of fine-grained GPU synchronization, it comes at the cost of programming complexity. Data-race-free is no longer a viable memory consistency model since locally scoped synchronization accesses potentially lead to "synchronization races" that can violate sequential consistency in non-intuitive ways (even for programs deemed to be well synchronized by the data-race-free memory model). Recently, Hower et al. addressed this problem by formalizing a new memory model, heterogeneous-race-free (HRF), to handle scoped synchronization. The Heterogeneous System Architecture (HSA) Foundation [1], a consortium of several industry vendors, and OpenCL 2.0 [13] have recently adopted a model similar to HRF with scoped synchronization.

Although HRF is a very well-defined model, it cannot hide the inherent complexity of using scopes. Intrinsically, scopes are a hardware-inspired mechanism that expose the memory hierarchy to the programmer. Using memory models to expose a hardware feature is consistent with the past evolution of memory models (e.g., the IBM 370 and total store order (TSO) models essentially expose hardware store buffers), but is discouraging when considering the past confusion generated by such an evolution. Previously, researchers have argued against such a hardware-centric view and proposed more software-centric models such as data-race-free [14]. Although data-race-free is widely adopted, it is still a source of much confusion [15]. Viewing the subtleties and complexities associated even with the so-called simplest models, we argue GPU consistency models should not be even more complex than the CPU models.

We therefore ask the question – can we develop coherence protocols for GPUs that are close to the simplicity of conventional GPU protocols but give the performance benefits of scoped synchronization, while enabling a memory model no more complex than data-race-free?

We show that this is possible by considering a recent hardware-software hybrid protocol, DeNovo [16, 17, 18], originally proposed for CPUs. DeNovo does not require writer-initiated invalidations or directories, but does obtain ownership for written data. The key additional overhead for DeNovo over GPU-style coherence with HRF is 1 bit per word at the caches (previous work on DeNovo uses software regions [16] for selective invalidations; our baseline DeNovo protocol does not use these to minimize overhead). Specifically, this paper makes the following contributions:

- We identify DeNovo (without regions) as a viable coherence protocol for GPUs. Due to its use of ownership on writes, DeNovo is able to exploit reuse of written data and synchronization variables across synchronization boundaries, without the additional complexity of scopes.
- We compare DeNovo with the DRF consistency model (i.e., no scoped synchronization) to GPU-style coherence with the DRF and HRF consistency models (i.e., without and with scoped synchronization). As expected, GPU+DRF performs poorly for applications with fine-grained synchronization. However, DeNovo+DRF provides a sweet spot in terms of performance, energy, implementation overhead, and memory model complexity.

  Specifically, we find the following when comparing the performance and energy for DeNovo+DRF and GPU+HRF. For benchmarks with no fine-grained synchronization, DeNovo is comparable to GPU. For microbenchmarks with globally scoped fine-grained synchronization, DeNovo is better than GPU (on average 28% lower execution time and 51% lower energy). For microbenchmarks with mostly locally scoped synchronization, GPU does better than DeNovo – on average 6% lower execution time, with a maximum reduction of 13% (4% and 10% lower, respectively, for energy). However, GPU+HRF's modest benefit for locally scoped synchronization must be weighed against HRF's higher complexity and GPU's much lower performance for globally scoped synchronization.
- For completeness, we also enhance DeNovo+DRF with selective invalidations to avoid invalidating valid, read-only data regions at acquires. The addition of a single software (read-only) region does not add any additional state overhead, but does require the software to convey the read-only region information. This enhanced DeNovo+DRF protocol provides the same performance and energy as GPU+HRF on average.
- For cases where HRF's complexity is deemed acceptable, we develop a version of DeNovo for the HRF memory model. We find that DeNovo+HRF is the best performing protocol – it is either comparable to or better than GPU+HRF for all cases and significantly better for applications with globally scoped synchronization.

Although conventional hardware protocols such as MESI support fine-grained synchronization with DRF, we do not compare to them here because prior research has observed that they incur significant complexity (e.g., writer-initiated invalidations, directory overhead, and many transient states leading to cache state overhead) and are a poor fit for conventional GPU applications [3, 19] (also evidenced by HSA's adoption of HRF). Additionally, the DeNovo project has shown that for CPUs,

DeNovo provides comparable or better performance than MESI at much less complexity [16, 17, 18].

This work is the first to show that GPUs can support fine-grained synchronization efficiently without resorting to the complexity of the HRF consistency model at modest hardware overhead. DeNovo with DRF provides a sweet spot for performance, energy, overhead, and memory model complexity, questioning the recent move towards memory models for GPUs that are more complex than those for CPUs.

2. BACKGROUND: MEMORY CONSISTENCY MODELS

Depending on whether the coherence protocol uses scoped synchronization or not, we assume either data-race-free (DRF) [14] or heterogeneous-race-free (HRF) [8] as our memory consistency model.

DRF ensures sequential consistency (SC) to data-race-free programs. A program is data-race-free if its memory accesses are distinguished as data or synchronization, and, for all its SC executions, all pairs of conflicting data accesses are ordered by DRF's happens-before relation. The happens-before relation is the irreflexive, transitive closure of program order and synchronization order, where the latter orders a synchronization write (release) before a synchronization read (acquire) if the write occurs before the read.

HRF is defined similar to DRF except that each synchronization access has a scope attribute and HRF's synchronization order only orders synchronization accesses with the same scope. There are two variants of HRF: HRF-Direct, which requires all threads that synchronize to use the same scope, and HRF-Indirect, which builds on HRF-Direct by providing extra support for transitive synchronization between different scopes. One key issue is the prospect of synchronization races – conflicting synchronization accesses to different scopes that are not ordered by HRF's happens-before. Such races are not allowed by the model and cannot be used to order data accesses.

Common implementations of DRF and HRF enforce a *program order requirement:* an access X must *complete* before an access Y if X is program ordered before Y and either (1) X is an acquire and Y is a data access, (2) X is a data access and Y is a release, or (3) X and Y are both synchronization. For systems with caches, the underlying coherence protocol governs the program order requirement by defining what it means for an access to *complete*, as discussed in the next section.

3. A CLASSIFICATION OF COHERENCE PROTOCOLS

The end-goal of a coherence protocol is to ensure that a read returns the correct value from the cache. For the DRF and HRF models, this is the value from the last conflicting write as ordered by the happens-before relation for the model. Following the observations made for the DeNovo protocol [16, 17], we divide the task of a coherence protocol into the following:

	Invalidation Initiator	**Tracking up-to-date copy**	**Different scopes?**
Conv HW	writer	ownership	yes
SW	reader	writethrough	yes
Hybrid	reader	ownership	yes

Table 1: Classification of protocols covering conventional HW (e.g., MESI), SW (e.g., GPU), and Hybrid (e.g., DeNovo) coherence protocols.

(1) *No stale data*: A load hit in a private cache should never see stale data.

(2) *Locatable up-to-date data*: A load miss in a private cache(s) must know where to get the up-to-date copy.

Table 1 classifies three classes of hardware coherence protocols in terms of how they enforce these requirements. Modern coherence protocols accomplish the first task through invalidation operations, which may be initiated by the writer or the reader of the data. The responsibility for the second task is usually handled by the writer, which either registers its ownership (e.g., at a directory) or uses writethroughs to keep a shared cache up-to-date. The HRF consistency model adds an additional dimension of whether a protocol can be enhanced with scoped synchronization.

Although our taxonomy is by no means comprehensive, it covers the space of protocols commonly used in CPUs and GPUs as well as recent work on hybrid software-hardware protocols. We next describe example implementations from each class. Without loss of generality, we assume a two level cache hierarchy with private L1 caches and a shared last-level L2 cache. In a GPU, the private L1 caches are shared by thread blocks [20] executing on the corresponding GPU CU.

Conventional Hardware Protocols used in CPUs

CPUs conventionally use pure hardware coherence protocols (e.g., MESI) that rely on writer-initiated invalidations and ownership tracking. They typically use a directory to maintain the list of (clean) sharers or the current owner of (dirty) data (at the granularity of a cache line). If a core issues a write to a line that it does not own, then it requests ownership from the directory, sending invalidations to any sharers or the previous owner of the line. For the purpose of invalidations and ownership, data and synchronization accesses are typically treated uniformly. For the program order constraint described in Section 2, a write is complete when its invalidations reach all sharers or the previous owner of the line. A read completes when it returns its value and that value is globally visible.

Although such protocols have not been explored with the HRF memory model, it is possible to exploit scoped synchronization with them. However, the added benefits, are unclear. Furthermore, as discussed in Section 1, conventional CPU protocols are a poor fit for GPUs and are included here primarily for completeness.

Software Protocols used in GPUs

GPUs use simple, primarily software-based coherence mechanisms, without writer-initiated invalidations or ownership tracking. We first consider the protocols without scoped synchronization.

GPU protocols use reader-initiated invalidations – an acquire synchronization (e.g., atomic reads or kernel launches) invalidates the entire cache so future reads do not return stale values. A write results in a writethough to a cache (or memory) shared by all the cores participating in the coherence protocol (the L2 cache with our assumptions) – for improved performance, these writethroughs are buffered and coalesced until the next release (or until the buffer is full). Thus, a (correctly synchronized) read miss can always obtain the up-to-date copy from the L2 cache.

Since GPU protocols do not have writer-initiated invalidations, ownership tracking, or scoped synchronization, they perform synchronization accesses at the shared L2 (more generally, the closest memory shared by all participating cores). For the program order requirement, preceding writes are now considered complete by a release when their writethroughs reach the shared L2 cache. Synchronization accesses are considered complete when they are performed at the shared L2 cache.

The GPU protocols are simple, do not require protocol state bits (other than valid bits), and do not incur invalidation and other protocol traffic overheads. However, synchronization operations are expensive – the operations are performed at the L2 (or the closest shared memory), an acquire invalidates the entire cache, and a release must wait until all previous writethroughs reach the shared L2. Scoped synchronizations reduce these penalties for local scopes.

In our two level hierarchy, there are two scopes – private L1 (shared by thread blocks on a CU) and shared L2 (shared by all cores and CUs). We refer to these as *local* and *global* scopes, respectively. A locally scoped synchronization does not have to invalidate the L1 (on an acquire), does not have to wait for writethroughs to reach the L2 (on a release), and is performed locally at the L1. Globally scoped synchronization is similar to synchronization accesses without scopes.

Although scopes reduce the performance penalty, they complicate the programming model, effectively exposing the memory hierarchy to the programmer.

DeNovo: A Hybrid Hardware-Software Protocol

DeNovo is a recent hybrid hardware-software protocol that uses reader-initiated invalidations with hardware tracked ownership. Since there are no writer-initiated invalidations, there is no directory needed to track sharers lists. DeNovo uses the shared L2's data banks to track ownership – either the data bank has the up-to-date copy of the data (no L1 cache owns it) or it keeps the ID of the core that owns the data. DeNovo refers to the L2 as the registry and the obtaining of ownership as registration. DeNovo has three states – Registered, Valid, and Invalid – similar to the Modified, Shared, and Invalid states of the MSI protocol. The key difference with MSI is that DeNovo has precisely these three states with no transient states, because DeNovo exploits data-race-freedom and does not have writer-initiated invalidations. A consequence of exploiting data-race-freedom is that the coherence states are stored at word granularity (although tags and data communication are at a larger conventional line granularity, like sector caches).[1]

Like GPU protocols, DeNovo invalidates the cache on an acquire; however, these invalidations can be selective in several ways. Our baseline DeNovo protocol exploits the property that data in registered state is up-to-date and thus does not need to be invalidated (even if the data is accessed globally by multiple CUs). Previous DeNovo work has also explored additional optimizations such as software regions and touched bits. We explore a simple variant where we identify read-only data regions and do not invalidate those on acquires (for simplicity, we do not explore more comprehensive regions or touched bits). The read-only region is a hardware oblivious, program level property and is easier to determine than annotating all synchronization accesses with (hardware- and schedule-specific) scope information.

For synchronization accesses, we use the DeNovoSync0 protocol [18] which registers both read and write synchronizations. That is, unless the location is in registered state in the L1, it is treated as a miss for both (synchronization) reads and writes and requires a registration operation. This potentially provides better performance than conventional GPU protocols, which perform all synchronization at the L2 (i.e., no synchronization hits).

DeNovoSync0 serves racy synchronization registrations immediately at the registry, in the order in which they arrive. For an already registered word, the registry forwards a new registration request to the registered L1. If the request reaches the L1 before the L1's own registration acknowledgment, it is queued at the L1's MSHR. In a high contention scenario, multiple racy synchronizations from different cores will form a distributed queue. Multiple synchronization requests from the same CU (from different thread blocks) are coalesced within the CU's MSHR and all are serviced before any queued remote request, thereby exploiting locality even under contention. As noted in previous work, the distributed queue serializes registration acknowledgments from different CUs – this throttling is beneficial when the contending synchronizations will be unsuccessful (e.g., unsuccessful lock accesses) but can add latency to the critical path if several of these synchronizations (usually readers) are successful. As discussed in [18], the latter case is uncommon.

DeNovoSync optimizes DeNovoSync0 by incorporating a backoff mechanism on registered reads when there is too much read-read contention. We do not explore it for simplicity.

To enforce the program order requirement, DeNovo considers a data write and a synchronization (read or write) complete when it obtains registration. As before, data reads are complete when they return their value.

DeNovo has not been previously evaluated with scoped synchronization, but can be extended in a natural way. Local acquires and releases do not invalidate the cache or flush the store buffer. Additionally, local synchro-

[1]This does not preclude byte granularity accesses as discussed in [16]. None of our benchmarks, however, have byte granularity accesses.

nization operations can delay obtaining ownership.

4. QUALITATIVE ANALYSIS OF THE PROTOCOLS

4.1 Qualitative Performance Analysis

We study the GPU and DeNovo protocols, with and without scopes, as described in Section 3. In order to understand the advantages and disadvantages of each protocol, Table 2 qualitatively compares coherence protocols across several key features that are important for emerging workloads with fine-grained synchronization: exploiting reuse of data across synchronization points (in L1), avoiding bursty traffic (especially for writes), decreasing network traffic by avoiding overheads like invalidations and acknowledgment messages, only transferring useful data by decoupling the coherence and transfer granularity, exploiting reuse of synchronization variables (in L1), and efficient support for dynamic sharing patterns such as work stealing. The coherence protocols have different advantages and disadvantages based on their support for these features:

GPU coherence, DRF consistency (*GPU-D*): Conventional GPU protocols with DRF do not require invalidation or acknowledgment messages because they self-invalidate all valid data at all synchronization points and write through all dirty data to the shared, backing LLC. However, there are also several inefficiencies which stem from poor support for fine-grained synchronization and not obtaining ownership. Because GPU coherence protocols do not obtain ownership (and don't have writer-initiated invalidations), they must perform synchronization accesses at the LLC, they must flush all dirty data from the store buffer on releases, and they must self-invalidate the entire cache on acquires. As a result, *GPU-D* cannot reuse any data across synchronization points (e.g., acquires, releases, and kernel boundaries). Flushing the store buffer at releases and kernel boundaries also causes bursty writethrough traffic. GPU coherence protocols also transfer data at a coarse granularity to exploit spatial locality; for emerging workloads with fine-grained synchronization or strided accesses, this can be sub-optimal. Furthermore, algorithms with dynamic sharing must synchronize at the LLC to prevent stale data from being accessed.

GPU coherence, HRF consistency (*GPU-H*): Changing the memory model from DRF to HRF removes several inefficiencies from GPU coherence protocols while retaining the benefit of no invalidation or acknowledgment messages. Although globally scoped synchronization accesses have the same behavior as *GPU-D*, locally scoped synchronization accesses occur locally and do not require bursty writebacks, self-invalidations, or flushes, improving support for fine-grained synchronization and allowing data to be reused across synchronization points. However, scopes do not provide efficient support for algorithms with dynamic sharing because programmers must conservatively use a global scope for these algorithms to prevent stale data from being accessed.

DeNovo coherence, DRF consistency (*DeNovo-D*): The DeNovo coherence protocol with DRF has several advantages over *GPU-D*. *DeNovo-D*'s use of ownership enables it to provide several of the advantages of *GPU-H* without exposing the memory hierarchy to the programmer. For example, *DeNovo-D* can reuse written data across synchronization boundaries since it does not self-invalidate registered data on an acquire. With the read-only optimization, this benefit also extends to read-only data. *DeNovo-D* also sees hits on synchronization variables with temporal locality both within a thread block and across thread blocks on the same CU. Obtaining ownership also allows *DeNovo-D* to avoid bursty writebacks at releases and kernel boundaries. Unlike *GPU-H*, obtaining ownership specifically provides efficient support for applications with dynamic sharing and also transfers less data by decoupling the coherence and transfer granularity.

Although obtaining ownership usually results in a higher hit rate, it can sometimes increase miss latency; e.g., an extra hop if the requested word is in a remote L1 cache or additional serialization for some synchronization patterns with high contention (Section 3). The benefits, however, dominate in our results.

DeNovo coherence, HRF consistency (*DeNovo-H*): Using the HRF memory model with the DeNovo coherence protocol combines all the advantages of ownership that *DeNovo-D* enjoys with the advantages of local scopes that *GPU-H* enjoys.

4.2 Protocol Implementation Overheads

Each of these protocols has several sources of implementation overhead:

GPU-D: Since *GPU-D* does not track ownership, the L1 and L2 caches only need 1 bit (a valid bit) per line to track the state of the cache line. GPU coherence also needs support for flash invalidating the entire cache on acquires and buffering writes until a release occurs.

GPU-H: GPU coherence with the HRF memory model additionally requires a bit per word in the L1 caches to keep track of partial cache block writes (3% overhead compared to *GPU-D*'s L1 cache). Like *GPU-D*, *GPU-H* also requires support for flash invalidating the cache for globally scoped acquires and releases and has an L2 overhead of 1 valid bit per cache line.

DeNovo-D and DeNovo-H: DeNovo needs per-word state bits for the DRF and HRF memory models because DeNovo tracks coherence at the word granularity. Since DeNovo has 3 coherence states, at the L1 cache we need 2 bits per-word (3% overhead over *GPU-H*). At the L2, DeNovo needs one valid and one dirty bit per line and one bit per word (3% overhead versus *GPU-H*).

DeNovo-D with read-only optimization (DeNovo-D+RO): Logically, DeNovo needs an additional bit per word at the L1 caches to store the read-only information. However, to avoid incurring additional overhead, we reuse the extra, unused state from DeNovo's coherence bits. There is some overhead to convey the region information from the software to the hardware. We pass this information through an opcode bit for memory in-

Feature	Benefit	GD	GH	DD	DH
Reuse Written Data	Reuse written data across synch points	✗	✓ (if local scope)	✓	✓
Reuse Valid Data	Reuse cached valid data across synch points	✗	✓ (if local scope)	✗[2]	✓ (if local scope)
No Bursty Traffic	Avoid bursts of writes	✗	✓ (if local scope)	✓	✓
No Invalidations/ACKs	Decreased network traffic	✓	✓	✓	✓
Decoupled Granularity	Only transfer useful data	✗	✗	✓	✓
Reuse Synchronization	Efficient support for fine-grained synch	✗	✓ (if local scope)	✓	✓
Dynamic Sharing	Efficient support for work stealing	✗	✗	✓	✓

Table 2: Comparison of studied coherence protocols.

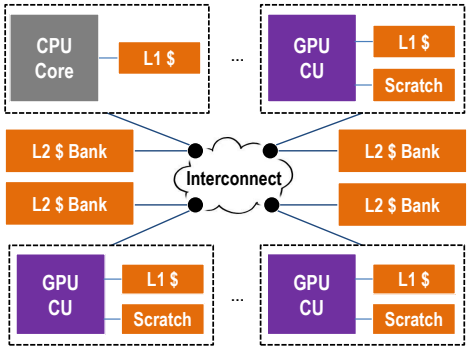


Figure 1: Baseline heterogeneous architecture [21].

CPU Parameters	
Frequency	2 GHz
Cores	1
GPU Parameters	
Frequency	700 MHz
CUs	15
Memory Hierarchy Parameters	
L1 Size (8 banks, 8-way assoc.)	32 KB
L2 Size (16 banks, NUCA)	4 MB
Store Buffer Size	256 entries
L1 hit latency	1 cycle
Remote L1 hit latency	35–83 cycles
L2 hit latency	29–61 cycles
Memory latency	197–261 cycles

Table 3: Simulated heterogeneous system parameters.

structions.

5. METHODOLOGY

Our work is influenced by previous work on DeNovo [16, 17, 18, 21]. We leverage the project's existing infrastructure [21] and extend it to support GPU synchronization operations based on the DeNovoSync0 coherence protocol for multicore CPUs [18].

5.1 Baseline Heterogeneous Architecture

We model a tightly coupled CPU-GPU architecture with a unified shared memory address space and coherent caches. The system connects all CPU cores and GPU Compute Units (CUs) via an interconnection network. Like prior work, each CPU core and each GPU CU (analogous to an NVIDIA SM) is on a separate network node. Each network node has an L1 cache (local to the CPU core or GPU CU) and a bank of the shared L2 cache (shared by all CPU cores and GPU CUs).[3] The GPU nodes also have a scratchpad. Figure 1 illustrates this baseline system which is similar to our prior work [21]. The coherence protocol, consistency model, and write policy depend on the system configuration studied (Section 5.3).

5.2 Simulation Environment and Parameters

We simulate the above architecture using an integrated CPU-GPU simulator built from the Simics full-system functional simulator to model the CPUs, the Wisconsin GEMS memory timing simulator [22], and GPGPU-Sim v3.2.1 [23] to model the GPU (the GPU is similar to an NVIDIA GTX 480). The simulator also uses Garnet [24] to model a 4x4 mesh interconnect with a GPU CU or a CPU core at each node. We use CUDA 3.1 [20] for the GPU kernels in the applications since this is the latest version of CUDA that is fully supported in GPGPU-Sim. Table 3 summarizes the common key parameters of our system.

For energy modeling, GPU CUs use GPUWattch [25] and the NoC energy measurements use McPAT v.1.1 [26] (our tightly coupled architecture more closely resembles a multicore system's NoC than the NoC modeled in GPUWattch). We do not model the CPU core or CPU L1 energy since the CPU is only functionally simulated and not the focus of this work.

We provide an API for manually inserted annotations for region information (for *DeNovo-D+RO*) and distinguishing synchronization instructions and their scope (for HRF).

5.3 Configurations

We evaluate the following configurations with GPU and DeNovo coherence protocols combined with DRF and HRF consistency models, using the implementations described in Section 3. The CPU always uses the DeNovo coherence protocol. For all configurations we assume 256 entry coalescing store buffers next to the L1 caches. We also assume support for performing synchronization accesses (using atomics) at the L1 and L2. We do not allow relaxed atomics [12] since precise semantics for them are under debate and their use is discouraged [15, 27].

GPU-D (GD): *GD* combines the baseline DRF memory model (no scopes) with GPU coherence and performs all synchronization accesses at the L2 cache.

[2]Mitigated by the read-only enhancement.

[3]HRF [8] uses a three-level cache hierarchy; we use two levels because the GEMS simulation environment (Section 5.2) only supports two levels. We believe our results are not qualitatively affected by the depth of the memory hierarchy.

Benchmark	Input
No Synchronization	
Backprop (BP)[28]	32 KB
Pathfinder (PF)[28]	10 x 100K matrix
LUD[28]	256x256 matrix
NW[28]	512x512 matrix
SGEMM[29]	medium
Stencil (ST)[29]	128x128x4, 4 iters
Hotspot (HS)[28]	512x512 matrix
NN[28]	171K records
SRAD v2 (SRAD)[28]	256x256 matrix
LavaMD (LAVA)[28]	2x2x2 matrix
Global Synchronization	
FA Mutex (FAM_G), Sleep Mutex (SLM_G), Spin Mutex (SPM_G), Spin Mutex+backoff (SPMBO_G),	3 TBs/CU, 100 iters/TB/kernel, 10 Ld&St/thr/iter
Local or Hybrid Synchronization	
FA Mutex (FAM_L), Sleep Mutex (SLM_L), Spin Mutex (SPM_L), Spin Mutex+backoff (SPMBO_L), Tree Barr+local exch (TBEX_LG), Tree Barr (TB_LG),	3 TBs/CU, 100 iters/TB/kernel, 10 Ld&St/thr/iter
Spin Sem (SS_L), Spin Sem+backoff (SSBO_L)[6]	3 TBs/CU, 100 iters/TB/kernel, readers: 10 Ld/thr/iter writers: 20 St/thr/iter
UTS[8]	16K nodes

Table 4: Benchmarks with input sizes. All thread blocks (TBs) in the synchronization microbenchmarks execute the critical section or barrier many times. Microbenchmarks with local and global scope are denoted with a '_L' and '_G', respectively.

GPU-H (GH): *GH* uses GPU coherence and HRF's HRF-Indirect memory model. GH performs locally scoped synchronization accesses at the L1s and globally scoped synchronization accesses at the L2.
DeNovo-D (DD): *DD* uses the DeNovoSync0 coherence protocol (without regions), a DRF memory model, and performs all synchronization accesses at the L1 (after registration).
DeNovo-D with read-only optimization (DD+RO): *DD+RO* augments *DD* with selective invalidations to avoid invalidating valid read-only data on acquires.
DeNovo-H (DH): *DH* combines DeNovo-D with the HRF-Indirect memory model. Like *GH*, local scope synchronizations always occur at the L1 and do not require invalidations or flushes.

5.4 Benchmarks

Evaluating our configurations is challenging because there are very few GPU application benchmarks that use fine-grained synchronization. Thus, we use a combination of application benchmarks and microbenchmarks to cover the space of use cases with (1) no synchronization within a GPU kernel, (2) synchronization that requires global scope, and (3) synchronization with mostly local scope. All codes execute GPU kernels on 15 GPU CUs and use a single CPU core. Parallelizing the CPU portions is left for future work.

5.4.1 *Applications without Intra-Kernel Synchronization*

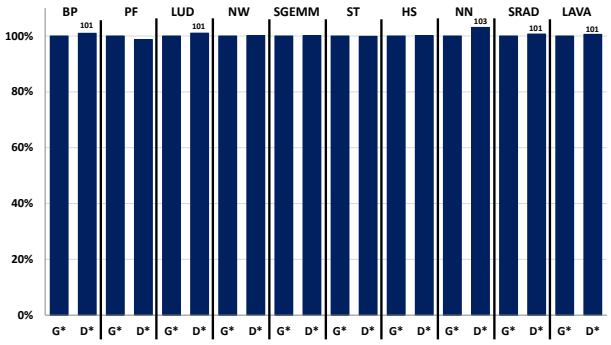
We examine 10 applications from modern heterogeneous computing suites such as Rodinia [28, 30] and Parboil [29]. None of these applications use synchronization within the GPU kernel and are also not written to exploit reuse across kernels. These applications therefore primarily serve to establish DeNovo as a viable protocol for today's use cases. The top part of Table 4 summarizes these applications and their input sizes.

5.4.2 *(Micro)Benchmarks with Intra-Kernel Synchronization*

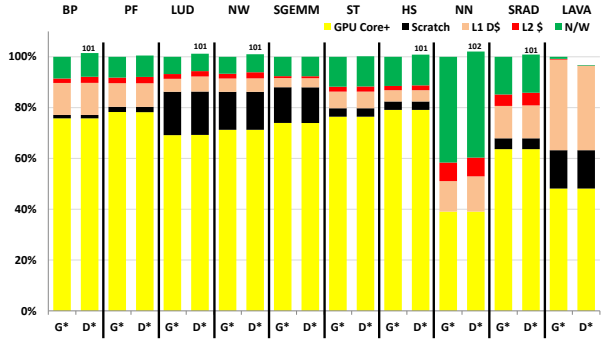
Most GPU applications do not use fine-grained synchronization because it is not well supported on current GPUs. Thus, to examine the performance for benchmarks with various kinds of synchronization we use a set of synchronization primitive microbenchmarks, developed by Stuart and Owens [6] – these include mutex locks, semaphores, and barriers. We also use the Unbalanced Tree Search (UTS) benchmark [8], the only benchmark that uses fine-grained synchronization in the HRF paper.[4] The microbenchmarks include centralized and decentralized algorithms with a wide range of stall cycles and scalability characteristics. The amount of work per thread also varies: the mutex and tree barrier algorithms access the same amount of data per thread while UTS and the semaphores access different amounts of data per thread. The bottom part of Table 4 summarizes the benchmarks and their input sizes.

We modified the original synchronization primitive microbenchmarks to perform data accesses in the critical section such that the mutex microbenchmarks have two versions: one performs local synchronization and accesses unique data per CU while the other uses global synchronization because the same data is accessed by all thread blocks. We also changed the globally synchronized barrier microbenchmark to use local and global synchronization with a tree barrier: all thread blocks on a CU access unique data and join a local barrier before one thread block from each CU joins the global barrier. After the global barrier, thread blocks exchange data for the subsequent iteration of the compute phase. We also added a version of the tree barrier where each CU exchanges data locally before joining the global barrier. Additionally, we changed the semaphores to use a reader-writer format with local synchronization: each CU has one writer thread block and two reader thread blocks. Each reader reads half of the CU's data. The writer shifts the reader thread block's data to the right such that all elements are written except for the first element of each thread block. To ensure that no stale data is accessed, the writers obtain the entire semaphore. The working set fits in the L1 cache for all microbenchmarks except TBEX_LG and TB_LG, which have larger working sets because they repeatedly exchange data across CUs.

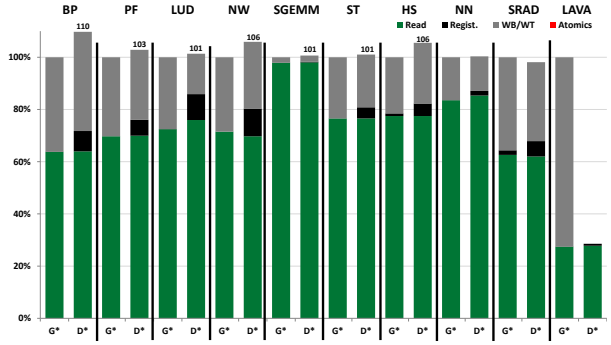
[4]RemoteScopes [11] uses several GPU benchmarks with fine-grained synchronization from Pannotia [10] but these benchmarks are not publicly available.



(a) Execution time

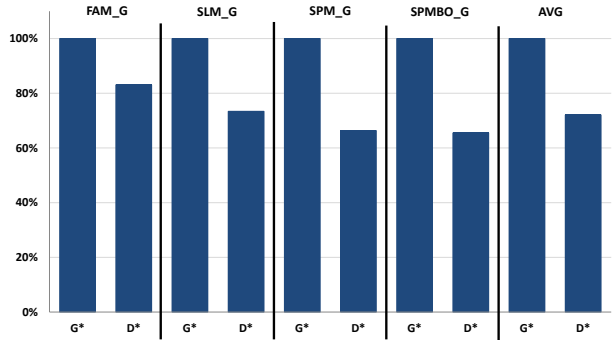


(b) Dynamic energy

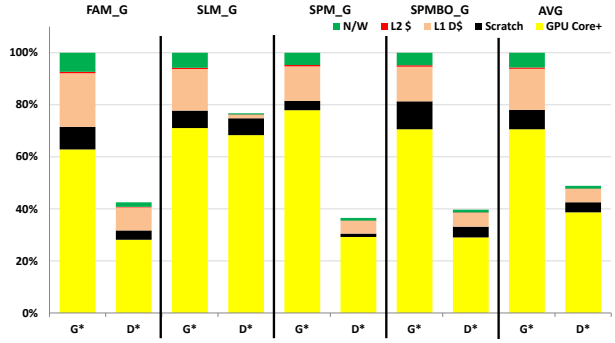


(c) Network traffic

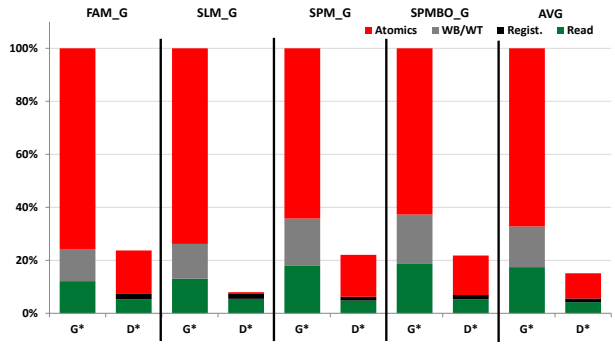
Figure 2: $G*$ and $D*$, normalized to $D*$, for benchmarks without synchronization.



(a) Execution time



(b) Dynamic energy



(c) Network traffic

Figure 3: $G*$ and $D*$, normalized to $G*$, for globally scoped synchronization benchmarks.

Similar to the tree barrier, the UTS benchmark utilizes both local and global synchronization. By performing local synchronization accesses, UTS quickly completes its work. However, since the tree is unbalanced, it is likely that some thread blocks will complete before others. To mitigate load imbalance, CU's push to and pull from a global task queue when their local queues become full or empty, respectively.

6. RESULTS

Figures 2, 3, and 4 show our results for the applications without fine-grained synchronization, for microbenchmarks with globally scoped fine-grained synchronization, and for codes with locally scoped or hybrid synchronization, respectively. Parts (a)-(c) in each figure show execution time, energy consumed, and network traffic, respectively. Energy is divided into multiple components based on the source of energy: GPU core+,[5] scratchpad, L1, L2, and network. Network traffic is measured in flit crossings and is also divided into multiple components: data reads, data registrations (writes), writebacks/writethroughs, and atomics.

In Figures 2 and 3, we only show the *GPU-D* and *DeNovo-D* configurations because HRF does not affect these cases (there is no local synchronization) – we denote the systems as $G*$ to indicate that *GPU-D* and *GPU-H* obtain the same results and $D*$ to indicate that *DeNovo-D* and *DeNovo-H* obtain the same results.[6] For Figure 4, we show all five configurations, denoted as GD, GH, DD, DD+RO, and DH.

Overall, compared to the best GPU coherence protocol (*GPU-H*), we find that *DeNovo-D* is comparable for

[5]GPU core+ includes the instruction cache, constant cache, register file, SFU, FPU, scheduler, and the core pipeline.
[6]We do not show the read-only enhancement here because $D*$ is significantly better than $G*$.

applications with no fine-grained synchronization and better for microbenchmarks that employ synchronization with only global scopes (average of 28% lower execution time, 51% lower energy). For microbenchmarks with mostly locally scoped synchronization, *GPU-H* is better (on average 6% lower execution time and 4% lower energy) than *DeNovo-D*. This modest benefit of *GPU-H* comes at the cost of a more complex memory model – adding a read-only region enhancement with *DeNovo-D* removes most of this benefit and using HRF with *DeNovo* makes it the best performing protocol.

6.1 GPU-D vs. GPU-H

Figure 4 shows that when locally scoped synchronization can be used, *GPU-H* can significantly improve performance over *GPU-D*, as noted in prior work [8]. On average *GPU-H* decreases execution time by 46% and energy by 42% for benchmarks that use local synchronization. There are two main sources of improvement. First, the latency of locally scoped acquires is much smaller because they are performed at L1 (which reduces atomic traffic by an average of 94%). Second, local acquires do not invalidate the cache and local releases do not flush the store buffer. As a result, data can be reused across local synchronization boundaries. Since accesses hit more frequently in the L1 cache, execution time, energy, and network traffic improve. On average, the L1, L2, and network energy components decrease by 71% for *GPU-H* while data (non-atomic) network traffic decreases by an average of 78%.

6.2 DeNovo-D vs. GPU Coherence

6.2.1 Traditional GPU Applications

For the ten applications studied that do not use fine-grained synchronization, Figure 2 shows there is generally little difference between *DeNovo** and *GPU**. *DeNovo** increases execution time and energy by 0.5% on average and reduces network traffic by 5% on average.

For LavaMD, *DeNovo** significantly decreases network traffic because LavaMD overflows the store buffer, which prevents multiple writes to the same location from being coalesced in *GPU**. As a result, each of these writes has to be written through separately to the L2. Unlike *GPU**, after *DeNovo** obtains ownership to a word, all subsequent writes to that word hit and do not need to use the store buffer.

For some other applications, obtaining ownership causes *DeNovo** to slightly increase network traffic and energy. First, multiple writes to the same word may require multiple ownership requests if the word is evicted from the cache before the last write. *GPU** may be able to coalesce these writes in the store buffer and incur a single writethrough to the L2. Second, *DeNovo** may incur a read or registration miss for a word registered at another core, requiring an extra hop on the network compared to *GPU** (which always hits in the L2). In our applications, however, these sources of overheads are minimal and do not affect performance. In general, the first source (obtaining ownership) is not on the critical path for performance and the second source (remote L1 miss) can be partly mitigated (if needed) using direct cache to cache transfers as enabled by *DeNovo* [16].

6.2.2 Global Synchronization Benchmarks

Figure 3 shows the execution time, energy, and network traffic for the four benchmarks that use only globally scoped fine-grained synchronization. For these benchmarks, HRF has no effect because there are no synchronizations with local scope.

The main difference between *GPU** and *DeNovo** is that *DeNovo** obtains ownership for written data and global synchronization variables, which gives the following key benefits for our benchmarks with global synchronization. First, once *DeNovo** obtains ownership for a synchronization variable, subsequent accesses from all thread blocks on the same CU incur hits (until another CU is granted ownership or the variable is evicted from the cache). These hits reduce average synchronization latency and network traffic for *DeNovo**. Second, *DeNovo** also benefits because owned data is not invalidated on an acquire, resulting in data reuse across synchronization boundaries for all thread blocks on a CU. Finally, release operations require getting ownership for dirty data instead of writing through the data to L2, resulting in less traffic.

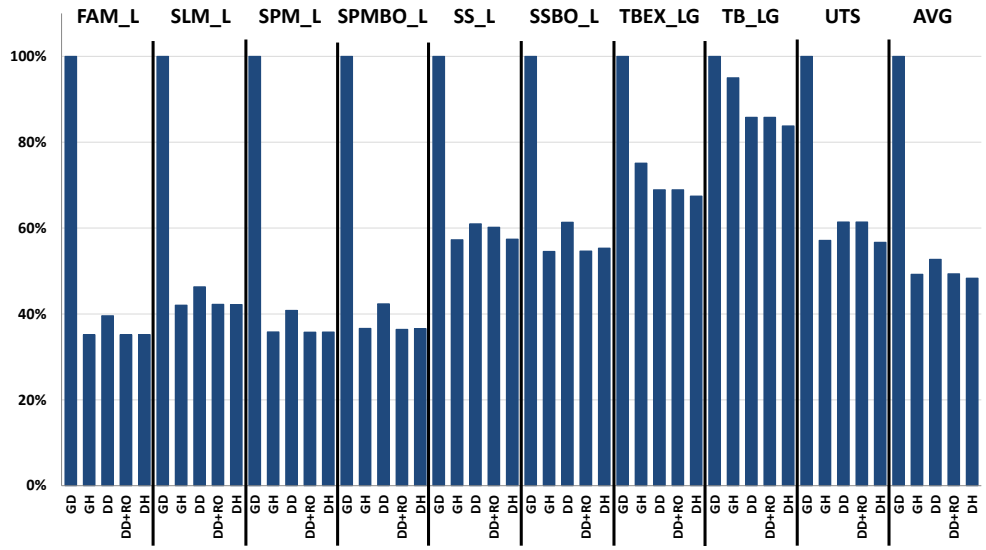
As discussed in Section 4.1, obtaining ownership can incur overheads relative to GPU coherence in some cases. However, for our benchmarks with global synchronization, these overheads are compensated by the reuse effects mentioned above. As a result, on average, *DeNovo** reduces execution time, energy, and network traffic by 28%, 51%, and 81%, respectively, relative to *GPU**.

6.2.3 Local Synchronization Benchmarks

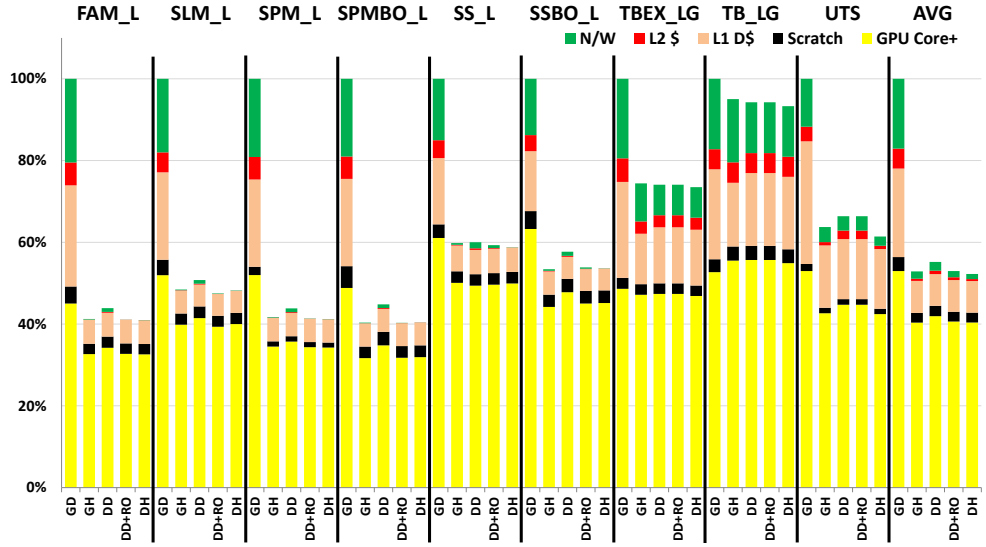
For the microbenchmarks with mostly locally scoped synchronization, we focus on comparing *DeNovo-D* with *GPU-H* since Figure 4 shows that the latter is the best GPU protocol.

DeNovo-D and *GPU-H* both increase reuse and synchronization efficiency relative to *GPU-D* for applications that use fine-grained synchronization, but they do so in different ways. *GPU-H* enables data reuse across local synchronization boundaries, and can perform locally scoped synchronization operations at L1. Therefore, these benefits can only be achieved if the application can explicitly define locally scoped synchronization points. In contrast, DeNovo enables reuse implicitly because owned data can be reused across any type of synchronization point. In addition, *DeNovo-D* obtains ownership for all synchronization operations, so even global synchronization operations can be performed locally. Like the globally scoped benchmarks, obtaining ownership for atomics also improves reuse and locality for benchmarks like TB_LG and TBEX_LG that have both global and local synchronization.

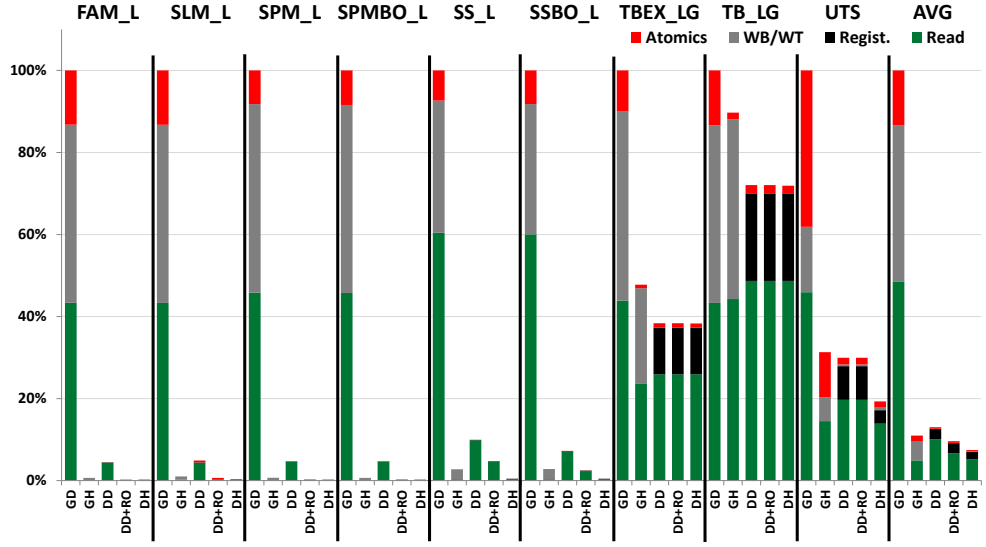
Since *GPU-H* does not obtain ownership, on a globally scoped release, it must flush and downgrade all dirty data to the L2. As a result, if the store buffer is too small, then *GPU-H* may see limited coalescing of writes to the same location, as described in Section 6.2.1. TB_LG and TBEX_LG exhibit this effect.



(a) Execution time



(b) Dynamic energy



(c) Network traffic

Figure 4: All configurations with synchronization benchmarks that use mostly local synchronization, normalized to *GD*.

DeNovo-D also occasionally suffers from full store buffers for these benchmarks, but its cost for flushing is lower – each dirty cache line only needs to send an ownership request to L2. Furthermore, once *DeNovo-D* obtains ownership, any additional writes will hit and do not need to use the store buffer, effectively reducing the number of flushes of a full store buffer. By obtaining ownership for the data, *DeNovo-D* is able to exploit more reuse. In doing so, *DeNovo* reduces network traffic and energy relative to *GPU-H* for these applications.

Conversely, *DeNovo* only enables reuse for owned data; i.e., there is no reuse across synchronization boundaries for read-only data. performance and increases network traffic with locally scoped synchronization. SS_L is also hurt by the order that the readers and writers enter the critical section: many readers enter first, so read-write data is invalidated until the writer enters and obtains ownership for it. *DeNovo-D* also performs slightly worse than *GPU-H* for UTS because DeNovo-D uses global synchronization and must frequently invalidate the cache and flush the store buffer. Although ownership mitigates many disadvantages of global synchronization, frequent invalidations and store buffer flushes limit the effectiveness of *DeNovo-D*.

On average, *GPU-H* shows 6% lower execution time and 4% lower energy than *DeNovo-D*, with maximum benefit of 13% and 10% respectively. However, *GPU-H*'s advantage comes at the cost of increased memory model complexity.

6.3 DeNovo-D with Selective (RO) Invalidations

DeNovo-D's inability to avoid invalidating read-only data is a key reason *GPU-H* outperforms it for the locally scoped microbenchmarks. Using the read-only region enhancement for *DeNovo-D*, however, removes any performance and energy benefit from *GPU-H* on average. In some cases, *GPU-H* is better, but only up to 7% for execution time and 4% for energy. Although *DeNovo-D+RO* needs more program information, unlike HRF, this information is hardware agnostic.

6.4 Applying HRF to DeNovo

DeNovo-H enjoys the benefits of ownership for data accesses and globally scoped synchronization accesses as well as the benefits of locally scoped synchronization. Reuse in L1 is possible for owned data across global synchronization points and for all data across local synchronization points. Local synchronization operations are always performed locally, and global synchronization operations are performed locally once ownership is acquired for the synchronization variable.

Compared to *DeNovo-D*, *DeNovo-H* provides some additional benefits. With *DeNovo-D* many synchronization accesses that would be locally scoped already occur at L1 and much data locality is already exploited through ownership. However, by explicitly defining local synchronization accesses, *DeNovo-H* is able to reuse read-only data and data that is read multiple times before it is written across local synchronization points. It is also able to delay obtaining ownership for both local writes and local synchronization operations. As a result, compared to *DeNovo-D*, *DeNovo-H* reduces execution time, energy, and network traffic for all applications with local scope.

Although *DeNovo-D+RO* allows reuse of read-only data, *DeNovo-H*'s additional advantages described above also provide it a slight benefit over *DeNovo-D+RO* in a few cases.

Compared to *GPU-H*, *DeNovo-H* is able to exploit more locality because owned data can be reused across any synchronization scope and because registration for synchronization variables allows global synchronization requests to also be executed locally.

These results show that *DeNovo-H* is the best configuration of those studied because it combines the advantages of ownership (from *DeNovo-D*) and scoped synchronization (from *GPU-H*) to minimize synchronization overhead and maximize data reuse across all synchronization points. However, *DeNovo-H* significantly increases memory model complexity and does not provide significantly better results than *DeNovo-D+RO*, which uses a simpler memory model but has some overhead to identify the read-only data.

7. RELATED WORK

7.1 Consistency

Previous work on memory consistency models for GPUs found that the TSO and relaxed memory models did not significantly outperform SC in a system with MOESI coherence and writeback caches [31, 32]. However, the work does not measure the coherence overhead of the studied configurations or evaluate alternative coherence protocols. The DeNovo coherence protocol also has several advantages over an ownership-based MOESI protocol, as discussed in Section 2.

7.2 Coherence Protocols

There has also been significant prior work on optimizing coherence protocols for standalone GPUs or CPU-GPU systems. Table 5 compares *DeNovo-D* to the most closely related prior work across the key features from Table 2:

HSC[33]: Heterogeneous System Coherence (HSC) is a hierarchical, ownership-based CPU-GPU cache coherence protocol. HSC provides the same advantages as the ownership-based protocols we discussed in Section 2. By adding coarse-grained hardware regions to MOESI, HSC aggregates coherence traffic and reduces MOESI's network traffic overheads when used with GPUs. However, HSC's coarse regions restrict data layout and the types of communication that can effectively occur. Furthermore, HSC's coherence protocol is significantly more complex than DeNovo.

Stash[21], **TemporalCoherence**[19], **FusionCoherence**[34]: Stash uses an extension of DeNovo for CPU-GPU systems, but focuses on integrating specialized, private memories like scratchpads into the unified address space. It does not provide support for fine-grained synchronization and does not draw any comparisons with conventional GPU style coherence or comment on

Feature	Benefit	HSC [33]	Stash[21], TC[19], FC[34]	Quick Release[3]	Remote Scopes[11]	DD
Reuse Written Data	Reuse written data across synchs	✓	✗	✓	✓	✓
Reuse Valid Data	Reuse cached valid data across synchs	✓	✗	✓	✗	✗
No Bursty Traffic	Avoid bursts of writes	✓	✓	✗	✗	✓
No Invalidations/ACKs	Decreased network traffic	✗	✓	✗	✗	✓
Decoupled Granularity	Only transfer useful data	✗	✓	✓ (for STs)	✓ (for STs)	✓
Reuse Synchronization	Efficient support for fine-grained synch	✓	✗	✓	✓	✓
Dynamic Sharing	Efficient support for work stealing	✓	✓	✗	✓	✓

Table 5: Comparison of DeNovo to other GPU coherence schemes. The read-only region enhancement to DeNovo also allows valid data reuse for read-only data.

consistency models. By extending the cache's coherence protocol used in this work to support local and global GPU synchronization operations, *DeNovo-D* reuses written data across synchronization points. We also explore DRF and HRF consistency models, while the stash assumes a DRF consistency model. FusionCoherence and TemporalCoherence use timestamp-based protocols that utilize self-invalidations and self-downgrades and thus provide many of the same benefits as *DeNovo-D*. However, this work does not consider fine-grained synchronization or impact on consistency models.

QuickRelease[3]: QuickRelease reduces the overhead of synchronization operations in conventional GPU coherence protocols and allows data to be reused across synchronization points. However, QuickRelease requires broadcast invalidations to ensure that no stale data can be accessed. Additionally, QuickRelease does not have efficient support for algorithms with dynamic sharing.

RemoteScopes[11]: RemoteScopes improves on QuickRelease by providing better support for algorithms with dynamic sharing. In the common case, dynamically shared data synchronizes with a local scope and when data is shared, RemoteScopes "promotes" the scope of the synchronization access to a larger common scope to synchronize properly. Although RemoteScopes improves performance for applications with dynamic sharing, because it does not obtain ownership, it must use heavyweight hardware mechanisms to ensure that no stale data is accessed. For example, RemoteScopes flushes the entire cache on acquires and uses broadcast invalidations and acknowledgments to ensure data is flushed.

Overall, while each of the coherence protocols in previous work provides some of the same benefits as *DeNovo-D*, none of them provide all of the benefits of *DeNovo-D*. Furthermore, none of the above work explores the impact of ownership on consistency models.

8. CONCLUSION

GPGPU applications with more general sharing patterns and fine-grained synchronization have recently emerged. Unfortunately, conventional GPU coherence protocols do not provide efficient support for them. Past work has proposed HRF, which uses scoped synchronization, to address the inefficiencies of conventional GPU coherence protocols. In this work, we choose instead to resolve these inefficiencies by extending DeNovo's software-driven, hardware coherence protocol to GPUs. DeNovo is a hybrid coherence protocol that provides the best features of ownership-based and GPU-style coherence protocols. As a result, DeNovo provides efficient support for applications with fine-grained synchronization. Furthermore, the DeNovo coherence protocol enables a simple SC-for-DRF memory consistency model. Unlike HRF, SC-for-DRF does not expose the memory hierarchy or require programmers to carefully annotate all synchronization accesses with scope information.

Across 10 CPU-GPU applications, which do not use fine-grained synchronization or dynamic sharing, DeNovo provides comparable performance to a conventional GPU coherence protocol. For applications that utilize fine-grained, globally scoped synchronization, DeNovo significantly outperforms a conventional GPU coherence protocol. For applications that utilize fine-grained, locally scoped synchronization, GPU coherence with HRF modestly outperforms the baseline DeNovo protocol, but at the cost of a more complex consistency model. Augmenting DeNovo with selective invalidations for read-only regions allows it to obtain the same average performance and energy as GPU coherence with HRF. Furthermore, if HRF's complexity is deemed acceptable, then a (modest) variant of DeNovo under HRF provides better performance and energy than conventional GPU coherence with HRF. These findings show that DeNovo with DRF provides a sweet spot for performance, energy, hardware overhead, and memory model complexity – HRF's complexity is not needed for efficient fine-grained synchronization on GPUs. Moving forward, we will analyze full-sized applications with fine-grained synchronization, once they become available, to ensure that DeNovo with DRF provides similar benefits for them. We will also examine what additional benefits we can obtain by applying additional optimizations, such as direct cache to cache transfers [16], to DeNovo with DRF.

Acknowledgments

This work was supported in part by a Qualcomm Innovation Fellowship for Sinclair, the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and the National Science Foundation under grants CCF-1018796 and CCF-1302641. We would also like to thank Brad Beckmann and Mark Hill for their insightful comments that improved the quality of this paper.

9. REFERENCES

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

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

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

[4] T. Sorensen, J. Alglave, G. Gopalakrishnan, and V. Grover, "ICS: U: Towards Shared Memory Consistency Models for GPUs," in *International Conference on Supercomputing*, 2013.

[5] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "GPU Concurrency: Weak Behaviours and Programming Assumptions," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[6] J. A. Stuart and J. D. Owens, "Efficient Synchronization Primitives for GPUs," *CoRR*, vol. abs/1110.4623, 2011.

[7] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IEEE International Symposium on Workload Characterization*, 2012.

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

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

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

[11] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, "Synchronization Using Remote-Scope Promotion," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[12] B. R. Gaster, D. Hower, and L. Howes, "HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models," *ACM Transactions on Architecture and Code Optimizations*, vol. 12, April 2015.

[13] L. Howes and A. Munshi, "The OpenCL Specification, Version 2.0." Khronos Group, 2015.

[14] S. Adve and M. Hill, "Weak Ordering – A New Definition," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

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

[16] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[17] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 13–26, 2013.

[18] H. Sung and S. V. Adve, "DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

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

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

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

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

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

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

[25] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[26] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[27] H.-J. Boehm and B. Demsky, "Outlawing Ghosts: Avoiding Out-of-thin-air Results," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, 2014.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization*, 2009.

[29] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," tech. rep., Department of ECE and CS, University of Illinois at Urbana-Champaign, 2012.

[30] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP workloads," in *IEEE International Symposium on Workload Characterization*, 2010.

[31] B. Hechtman and D. Sorin, "Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

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

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

[34] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design Tradeoffs in Coherence Cache Hierarchies for Accelerators," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.