Cache Coherence for GPU Architectures

Inderpreet Singh[1] Arrvindh Shriraman[2] Wilson W. L. Fung [1]
Mike O'Connor[3] Tor M. Aamodt[1,4]
[1]University of British Columbia [2]Simon Fraser University
[3]Advanced Micro Devices, Inc. (AMD) [4]Stanford University
isingh@ece.ubc.ca, ashriram@cs.sfu.ca, wwlfung@ece.ubc.ca
mike.oconnor@amd.com, aamodt@ece.ubc.ca

Abstract

While scalable coherence has been extensively studied in the context of general purpose chip multiprocessors (CMPs), GPU architectures present a new set of challenges. Introducing conventional directory protocols adds unnecessary coherence traffic overhead to existing GPU applications. Moreover, these protocols increase the verification complexity of the GPU memory system. Recent research, Library Cache Coherence (LCC) [34, 54], explored the use of time-based approaches in CMP coherence protocols.

This paper describes a time-based coherence framework for GPUs, called Temporal Coherence (TC), that exploits globally synchronized counters in single-chip systems to develop a streamlined GPU coherence protocol. Synchronized counters enable all coherence transitions, such as invalidation of cache blocks, to happen synchronously, eliminating all coherence traffic and protocol races. We present an implementation of TC, called TC-Weak, which eliminates LCC's trade-off between stalling stores and increasing L1 miss rates to improve performance and reduce interconnect traffic.

By providing coherent L1 caches, TC-Weak improves the performance of GPU applications with inter-workgroup communication by 85% over disabling the non-coherent L1 caches in the baseline GPU. We also find that write-through protocols outperform a writeback protocol on a GPU as the latter suffers from increased traffic due to unnecessary refills of write-once data.

1 Introduction

Graphics processor units (GPUs) have become ubiquitous in high-throughput, general purpose computing. C-based programming interfaces like OpenCL [29] and NVIDIA CUDA [46] ease GPU programming by abstracting away the SIMD hardware and providing the illusion of independent scalar threads executing in parallel. Traditionally limited to regular parallelism, recent studies [21, 41] have shown that even highly irregular algorithms can attain significant speedups on a GPU. Furthermore, the inclusion of a multi-level cache hierarchy in recent GPUs [6, 44] frees the programmer from the burden of software managed caches and further increases the GPU's attractiveness as a platform for accelerating applications with irregular memory access patterns [22, 40].

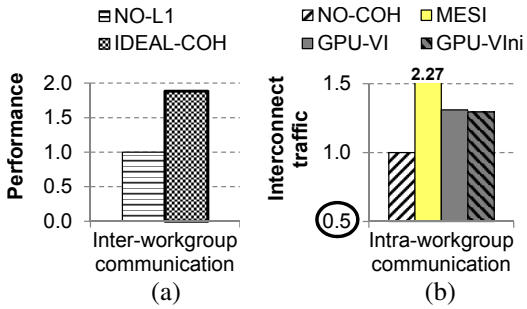


Figure 1. (a) Performance improvement with ideal coherence. (b) Traffic overheads of conventional coherence.

GPUs lack cache coherence and require disabling of private caches if an application requires memory operations to be visible across all cores [6, 44, 45]. General-purpose chip multiprocessors (CMPs) regularly employ hardware cache coherence [17, 30, 32, 50] to enforce strict memory consistency models. These consistency models form the basis of memory models for high-level languages [10, 35] and provide the synchronization primitives employed by multi-threaded CPU applications. Coherence greatly simplifies supporting well-defined consistency and memory models for high-level languages on GPUs. It also helps enable a unified address space in heterogeneous architectures with single-chip CPU-GPU integration [11, 26]. This paper focuses on coherence in the realm of GPU cores; we leave CPU-GPU cache coherence as future work.

Disabling L1 caches trivially provides coherence at the

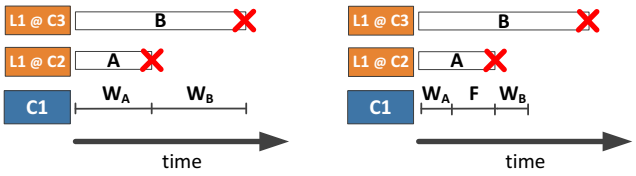
cost of application performance. Figure 1(a) shows the potential improvement in performance for a set of GPU applications (described in Section 7) that contain inter-workgroup communication and require coherent L1 caches for correctness. Compared to disabling L1 caches (NO-L1), an ideally coherent GPU (IDEAL-COH), where coherence traffic does not incur any latency or traffic costs, improves performance of these applications by 88% on average.

GPUs present three main challenges for coherence. Figure 1(b) depicts the first of these challenges by comparing the interconnect traffic of the baseline non-coherent GPU system (NO-COH) to three GPU systems with cache coherence protocols: writeback MESI, inclusive write-through GPU-VI and non-inclusive write-through GPU-VIni (described in Section 4). These protocols introduce **unnecessary coherence traffic overheads** for GPU applications containing data that does not require coherence.

Second, on a GPU, CPU-like worst case sizing [18] would require an **impractical amount of storage** for tracking thousands of in-flight coherence requests. Third, existing coherence protocols introduce **complexity in the form of transient states and additional message classes**. They require additional virtual networks [58] on GPU interconnects to ensure forward progress, and as a result increase power consumption. The challenge of tracking a large number of sharers [28, 64] is not a problem for current GPUs as they contain only tens of cores.

In this paper, we propose using a time-based coherence framework for minimizing the overheads of GPU coherence without introducing significant design complexity. Traditional coherence protocols rely on explicit messages to inform others when an address needs to be invalidated. We describe a time-based coherence framework, called Temporal Coherence (TC), which uses synchronized counters to self-invalidate cache blocks and maintain coherence invariants without explicit messages. Existing hardware implements counters synchronized across components [23, Section 17.12.1] to provide efficient timer services. Leveraging these counters allows TC to eliminate coherence traffic, lower area overheads, and reduce protocol complexity for GPU coherence. TC requires prediction of cache block lifetimes for self-invalidation.

Shim *et al.* [34, 54] recently proposed a time-based hardware coherence protocol, Library Cache Coherence (LCC), that implements sequential consistency on CMPs by stalling writes to cache blocks until they have been self-invalidated by all sharers. We describe one implementation of the TC framework, called TC-Strong, that is similar to LCC. Section 8.3 shows that TC-Strong performs poorly on a GPU. Our second implementation of the TC framework, called TC-Weak, uses a novel timestamp-based memory fence mechanism to eliminate stalling of writes. TC-Weak uses timestamps to drive all consistency operations. It implements Release Consistency [19], enabling full support of C++ and Java memory models [58] on GPUs.



(a) TC-Strong / LCC [34, 54] (b) TC-Weak

Figure 2. TC operation. W_A=Write to address A. F=Fence. $\times$=Self-invalidation[2] of a cache block.

Figure 2 shows the high-level operation of TC-Strong and TC-Weak. Two cores, C2 and C3, have addresses A and B cached in their private L1, respectively. In TC-Strong, C1's write to A stalls completion until C2 self-invalidates its locally cached copy of A. Similarly, C1's write to B stalls completion until C3 self-invalidates its copy of B. In TC-Weak, C1's writes to A and B do not stall waiting for other copies to be self-invalidated. Instead, the fence operation ensures that all previously written addresses have been self-invalidated in other local caches. This ensures that all previous writes from this core will be globally visible after the fence completes.

The contributions of this paper are:

- It discusses the challenges of introducing existing coherence protocols to GPUs. We introduce two optimizations to a VI protocol [30] to make it more suitable for GPUs.
- It provides detailed complexity and performance evaluations of inclusive and non-inclusive directory protocols on a GPU.
- It describes Temporal Coherence, a GPU coherence framework for exploiting synchronous counters in single-chip systems to eliminate coherence traffic and protocol races.
- It proposes the TC-Weak coherence protocol which employs timestamp based memory fences to implement Release Consistency [19] on a GPU.
- It proposes a simple lifetime predictor for TC-Weak that performs well across a range of GPU applications.

Our experiments show that TC-Weak with a simple lifetime predictor improves performance of a set of GPU applications with inter-workgroup communication by 85% over the baseline non-coherent GPU. On average, it performs as well as the VI protocols and 23% faster than MESI across all our benchmarks. Furthermore, for a set of GPU applications with intra-workgroup communication, it reduces the traffic overheads of MESI, GPU-VI and GPU-VIni by 56%, 23% and 22%, while reducing interconnect energy usage by

[2]Time-based self-invalidation does not require explicit events; the block will be invalid for the next access.

40%, 12% and 12%. Compared to TC-Strong, TC-Weak performs 28% faster with 26% lower interconnect traffic across all applications.

The remainder of the paper is organized as follows. Section 2 discusses related work, Section 3 reviews GPU architectures and cache coherence, Section 4 describes the directory protocols, and Section 5 describes the challenges of GPU coherence. Section 6 details the implementations of TC-Strong and TC-Weak, Sections 7 and 8 present our methodology and results, and Section 9 concludes.

2 Related Work

The use of timestamps has been explored in software coherence [42, 63]. Nandy *et al.* [43] first considered timestamps for hardware coherence. Library Cache Coherence (LCC) [34, 54] is a time-based hardware coherence proposal that stores timestamps in a directory structure and delays stores to unexpired blocks to enforce sequential consistency on CMPs. The TC-Strong implementation of the TC framework is similar to LCC as both enforce write atomicity by stalling writes at the shared last level cache. Unlike LCC, TC-Strong supports multiple outstanding writes from a core and implements a relaxed consistency model. TC-Strong includes optimizations to eliminate stalls due to private writes and L2 evictions. Despite these changes, we find that the stalling of writes in TC-Strong causes poor performance on a GPU. We propose TC-Weak and a novel time-based memory fence mechanism to eliminate all write-stalling, improve performance, and reduce interconnect traffic compared to TC-Strong. We also show that unlike for CPU applications [34, 54], the fixed timestamp prediction proposed by LCC is not suited for GPU applications. We propose a simple yet effective lifetime predictor that can accommodate a range of GPU applications. Lastly, we present a full description of our proposed protocol, including state transition tables that describe the implementation in detail.

Self invalidation of blocks in a private cache has also been previously explored in the context of cache coherence. Dynamic Self-Invalidation (DSI) [33] reduces critical path latency due to invalidation by speculatively self-invalidating blocks in private caches before the next exclusive request for the block is received. In the sequentially consistent implementation, DSI requires explicit messages to the directory at self-invalidation and would not alleviate the traffic problem on a GPU. In its relaxed consistency implementation, DSI can reduce traffic through the use of *tear-off* blocks, which are self-invalidated at synchronization points. Recently, Ros *et al.* [48] proposed extending tear-off blocks to all cache blocks to eliminate coherence directories entirely, reducing implementation complexity and traffic for CPU coherence. Their protocol requires self-invalidation of all shared data at synchronization points. Synchronization events, however, are much more frequent on a GPU.

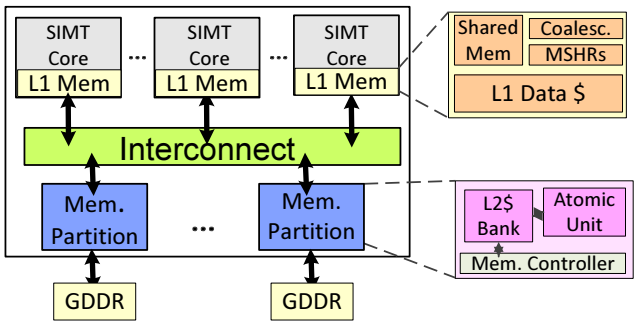


Figure 3. Baseline non-coherent GPU Architecture.

Thousands of scalar threads share a single L1 cache and would cause frequent self-invalidations. Their protocol also requires blocking and buffering atomic operations at the last level cache. GPUs support thousands of concurrent atomic operations; buffering these would be very expensive.

Denovo [16] simplifies the coherence directory but uses a restrictive programming model that requires user annotated code. TC-Weak does not change the GPU programming model. Recent coherence proposals [28, 51, 64] simplify tracking sharer state for 1000s of cores. GPUs have tens of cores; exact sharer representation is not an issue.

3 Background

This section describes the memory system and cache hierarchy of the baseline non-coherent GPU architecture, similar to NVIDIA's Fermi [44], that we evaluate in this paper. Cache coherence is also briefly discussed.

3.1 Baseline GPU Architecture

Figure 3 shows the organization of our baseline non-coherent GPU architecture. An OpenCL [29] or CUDA [46] application begins execution on a CPU and launches *compute kernels* onto a GPU. Each kernel launches a hierarchy of threads (an *NDRange* of *work groups* of *wavefronts* of *work items/scalar threads*) onto a GPU. Each workgroup is assigned to a heavily multi-threaded GPU core. Scalar threads are managed as a SIMD execution group consisting of 32 threads called a warp (NVIDIA terminology) or wavefront (AMD terminology).

GPU Memory System. A GPU kernel commonly accesses the *local*, *thread-private* and *global* memory spaces. Software managed local memory is used for intra-workgroup communication. Thread-private memory is private to each thread while the global memory is shared across all threads on a GPU. Both thread-private and global memory are stored in off-chip GDDR DRAM and cached in the multi-level cache hierarchy, however only global memory requires coherence. The off-chip DRAM memory is divided among a number of memory partitions that connect to the GPU cores through an interconnection network. Memory accesses to the same cache block from different threads

within a wavefront are merged into a single wide access by the Coalescing Unit. A memory instruction generates one memory access for every unique cache line accessed by the wavefront. All requests are handled in FIFO order by the in-order memory stage of a GPU core. Writes to the same word by multiple scalar threads in a single wavefront do not have a defined behaviour [46]; only one write will succeed. In this paper, from the memory consistency model's perspective, a GPU wavefront is similar to a CPU thread.

GPU Cache Hierarchy. The GPU cache hierarchy consists of per-core private L1 data caches and a shared L2 cache. Each memory partition houses a single bank of the L2 cache. The L1 caches are not coherent. They follow a *write-evict* [46] (write-purge [24]), write no-allocate caching policy. The L2 caches are writeback with write-allocate. Memory accesses generated by the coalescing unit in each GPU core are passed, one per cycle, to the per-core MSHR table. The MSHR table combines read accesses to the same cache line from different wavefronts to ensure only a single read access per-cache line per-GPU core is outstanding. Writes are not combined and, since they write-through, any number of write requests to the same cache line from a GPU core may be outstanding. Point-to-point ordering in the interconnection network, L2 cache controllers and off-chip DRAM channels ensures that multiple outstanding writes from the same wavefront to the same address complete in program order. All cache controllers service one memory request per cycle in order. Misses at the L2 are handled by allocating an MSHR entry and removing the request from the request queue to prevent stalling.

Atomic Operation. Read-modify-write atomic operations are performed at each memory partition by an Atomic Operation Unit. In our model, the Atomic Operation Unit can perform a read-modify-write operation on a line resident in the L2 cache in a single cycle.

3.2 Consistency and Coherence

A cache coherence protocol performs the following three duties [3]. It propagates newly written values to all privately cached copies. It informs the writing thread or processor when a write has been completed and is visible to all threads and processors. Lastly, a coherence protocol may ensure write atomicity [3], *i.e.*, a value from a write is logically seen by all threads at once. Write atomicity is commonly enforced in write-invalidate coherence protocols by requiring that all other copies of a cache block are invalidated before a write is completed. Memory consistency models may [4, 19, 57, 59] or may not [2, 19, 53] require write atomicity.

4 Directory Protocols

This section describes the MESI and GPU-VI directory protocols that we compare against in this paper. All of MESI, GPU-VI and GPU-VIni require a coherence directory to track the L1 sharers. MESI and GPU-VI enforce inclusion through invalidation (*recall*) of all L1 copies of a cache line upon L2 evictions. Inclusion allows the sharer list to be stored with the L2 tags. GPU-VIni is non-inclusive and requires separate on-chip storage for a directory.

4.1 MESI

MESI is a four-state coherence protocol with writeback L1 and L2 caches. It contains optimizations to eliminate the point-to-point ordering requirement of the non-coherent GPU interconnect and cache controllers. Instead, MESI relies on five physical or virtual networks to support five different message classes to prevent protocol deadlocks. MESI implements complex cache controllers capable of selecting serviceable requests from a pool of pending requests. The write-allocate policy at L1 requires that write data be buffered until proper coherence permission has been obtained. This requires the addition of area and complexity to buffer stores in each GPU core.

4.2 GPU-VI

GPU-VI is a two-state coherence protocol inspired by the write-through protocol in Niagara [30]. GPU-VI implements write-through, no write-allocate L1 caches. It requires that any write completing at the L2 invalidate all L1 copies. A write to a shared cache line cannot complete until the L2 controller has sent invalidation requests and received acknowledgments from all sharers.

GPU-VI adds two optimizations to a conventional VI protocol [60]. First, it writes data directly to the L1 cache on a write hit before receiving an acknowledgement, eliminating the area and complexity overheads of buffering stores. Second, it treats loads to L1 blocks with pending writes as misses. This reduces stalling at the cache controller while maintaining write atomicity. GPU-VI requires 4 physical or virtual networks to guarantee deadlock-free execution.

4.3 GPU-VIni

The non-inclusive GPU-VIni decouples the directory storage in GPU-VI from the L2 cache to allow independent scaling of the directory size. It adds additional complexity to manage the states introduced by a separate directory structure. The same cache controller in GPU-VIni manages the directory and the L2 cache. Eviction from the L2 cache does not generate recall requests, however eviction from the directory requires recall. GPU-VIni implements an 8-way associative directory with twice the number of entries as the number of total private cache blocks (R=2 as in the framework proposed by Martin *et al.* [39]). Section 8.5 presents data for GPU-VIni with larger directory sizes.

5 Challenges of GPU Coherence

This section describes the main challenges of introducing conventional coherence protocols to GPUs.

Table 1. Number of protocol states.

	State Type	Non-Coh.	GPU-VI	GPU-VIni	MESI	**TC-Weak**
L1 Cache	Stable	2	2	2	4	**2**
	Transient Cache	2	2	2	2	**2**
	Transient Coherent	0	1	1	4	**1**
	Total L1 States	4	5	5	10	**5**
L2 Cache	Stable	2	3	5	4	**4**
	Transient Cache	2	2	2	3	**2**
	Transient Coherent	0	3	8	9	**1**
	Total L2 States	4	8	15	16	**7**

5.1 Coherence Traffic

Traditional coherence protocols introduce unnecessary traffic overheads to existing GPU applications that are designed for non-coherent GPU architectures. These overheads consist of recall traffic due to directory evictions, false sharing invalidation traffic, and invalidation traffic due to inter-kernel communication. Recall traffic becomes especially problematic for inclusive protocols on GPUs because the shared GPU L2 cache size matches the aggregate private L1 cache size [6, 44]. An inclusive cache hierarchy is an attractive [7] choice for low-complexity coherence implementations. Moreover, large directories required to reduce recall traffic [39] in non-inclusive protocols take valuable space from the GPU L2 cache.

An effective way to reduce coherence traffic is to selectively disable coherence for data regions that do not require it. Kelm *et al.* [27] proposed a hybrid coherence protocol to disable hardware coherence for regions of data. It requires additional hardware support and code modifications to allow data to migrate between coherence domains. Section 6.3 explains how TC-Weak uses timestamps to enforce coherence at cache line granularity without requiring any code modifications to identify coherent and non-coherent data.

5.2 Storage Requirements

With only tens of threads per core, CPU coherence implementations can dedicate enough on-chip storage resources to buffer the worst case number of coherence requests [18]. GPUs, however, execute tens of thousands of scalar threads in parallel. In a CPU-like coherence implementation [18] with enough storage to handle the worst case number of memory accesses (one memory request per thread), a directory protocol would require an impractical on-chip buffer as large as 28% of the total GPU L2 cache for tracking coherence requests. Reducing the worst-case storage overhead requires throttling the network via back-pressure flow-control mechanisms when the end-point queues fill up [37]. TC-Weak eliminates coherence messages and the storage cost of buffering them.

5.3 Protocol Complexity

Table 1 lists the number of states in the protocols we evaluate. We term *stable states* as states conventionally associated with a coherence protocol, for example, Modified, Exclusive, Shared and Invalid for the MESI protocol. Transient states are intermediate states occurring between stable states. Specifically, *transient cache states* are states associated with regular cache operations, such as maintaining the state of a cache block while a read miss is serviced. Transient cache states are present in a coherence protocol as well as the non-coherent architecture. *Transient coherent states* are additional states needed by the coherence protocol. An example is a state indicating that the given block is waiting for invalidation acknowledgments. Coherence protocol verification is a significant challenge that grows with the number of states [16], a problem referred to as state space explosion [47]. As shown in Table 1, MESI, GPU-VIni and GPU-VI add 13, 9 and 4 transient coherent states over the baseline non-coherent caches, increasing verification complexity. TC-Weak requires only a single transient state in the L1 and L2. Message based coherence protocols require additional virtual networks [58] or deadlock detection mechanisms [31] to ensure forward progress. As shown in Table 4, MESI requires 3 additional, and GPU-VI and GPU-VIni require 2 additional virtual networks over the baseline GPU. The additional virtual networks prevent deadlocks when circular resource dependencies, introduced by coherence messages, arise. Since TC-Weak eliminates coherence messages, additional virtual networks are not necessary.

6 Temporal Coherence

This section presents Temporal Coherence (TC), a timestamp based cache coherence framework designed to address the needs of high-throughput GPU architectures. Like LCC, TC uses time-based self-invalidation to eliminate coherence traffic. Unlike LCC, which implements sequential consistency for CMPs, TC provides a relaxed memory model [58] for GPU applications. TC requires fewer modifications to GPU hardware and enables greater memory level parallelism. Section 6.1 describes time-based coherence. Section 6.2 describes TC-Strong and compares it to LCC. Section 6.3 describes TC-Weak, a novel TC protocol that uses time to drive both coherence and consistency operations.

6.1 Time and Coherence

In essence, the task of an invalidation-based coherence protocol is to communicate among a set of nodes the beginnings and ends of a memory location's epochs [58]. Time-based coherence uses the insight that single chip systems can implement synchronized counters [23, Section 17.12.1] to enable low cost transfer of coherence information. Specifically, if the lifetime of a memory address' current epoch can be predicted and shared among all readers when the location is read, then these counters allow the readers to self-invalidate synchronously, eliminating the need for end-of-epoch invalidation messages.

Figure 4 compares the handling of invalidations between the GPU-VI directory protocol and TC. The figure depicts a

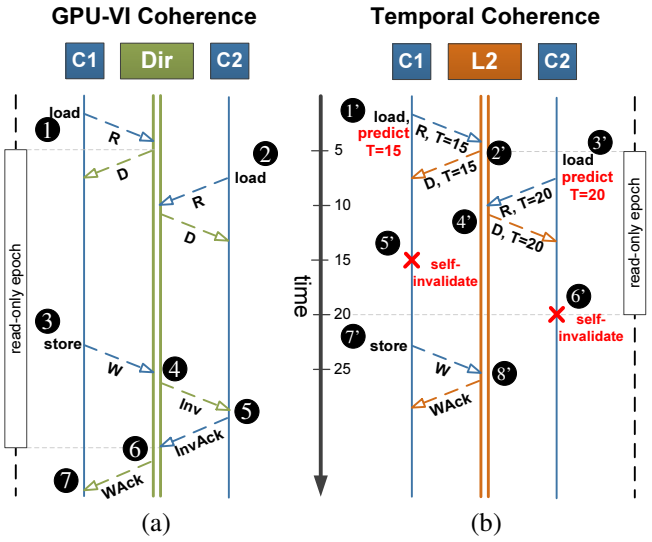


Figure 4. Coherence invalidation mechanisms. Messages: R=read, D=data, W=write, Inv=invalidation

read by processors C1 and C2, followed by a store from C1, all to the same memory location. Figure 4(a) shows the sequence of events that occur for the write-through GPU-VI directory protocol. C1 issues a load request to the directory (❶), and receives data. C2 issues a load request (❷) and receives the data as well. C1 then issues a store request (❸). The directory, which stores an exact list of sharers, sees that C2 needs to be invalidated before the write can complete and sends an invalidation request to C2 (❹). C2 receives the invalidation request, invalidates the block in its private cache, and sends an acknowledgment back (❺). The directory receives the invalidation acknowledgment from C2 (❻), completes C1's store request, and sends C1 an acknowledgment (❼).

Figure 4(b) shows how TC handles the invalidation for this example. When C1 issues a load request to the L2, it predicts that the read-only epoch for this address will end at time T=15 (❶'). The L2 receives C1's load request and epoch lifetime prediction, records it, and replies with the data and timestamp of T=15 (❷'). The timestamp indicates to C1 that it must self-invalidate this address in its private cache by T=15. When C2 issues a load request, it predicts the epoch to end at time T=20 (❸'). The L2 receives C2's request, checks the timestamp stored for this address and extends it to T=20 to accommodate C2's request, and replies with the data and a timestamp of T=20 (❹'). At time T=15 (❺'), C1's private cache self-invalidates the local copy of the address. At time T=20 (❻'), C2 self-invalidates its local copy. When C1 issues a store request to the L2 (❼'), the L2 finds the global timestamp (T=20) to be less than the current time (T=25) indicating that no L1's contain a valid copy of this address. The L2 completes the write instantly and sends an acknowledgment to C1 (❽').

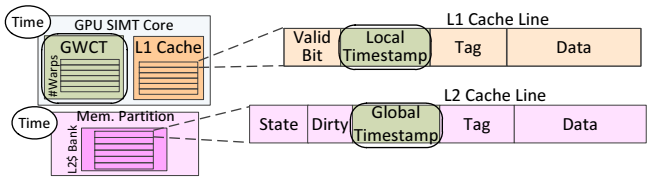


Figure 5. Hardware extensions for TC-Weak. (a) GPU cores and memory partitions with synchronized counters. A GWCT table added to each GPU core. (b) L1 and L2 cache lines with timestamp field.

Compared to GPU-VI, TC does not use invalidation messages. Globally synchronized counters allow the L2 to make coherence decisions locally and without indirection. This example shows how a TC framework can achieve our desired goals for GPU coherence; all coherence traffic has been eliminated and, since there are no invalidation messages, the transient states recording the state of outstanding invalidation requests are no longer necessary. Lifetime prediction is important in time-based coherence as it affects cache utilization and application performance. Section 6.4 describes our simple predictor for TC-Weak that adjusts the requested lifetime based on application behaviour.

6.2 TC-Strong Coherence

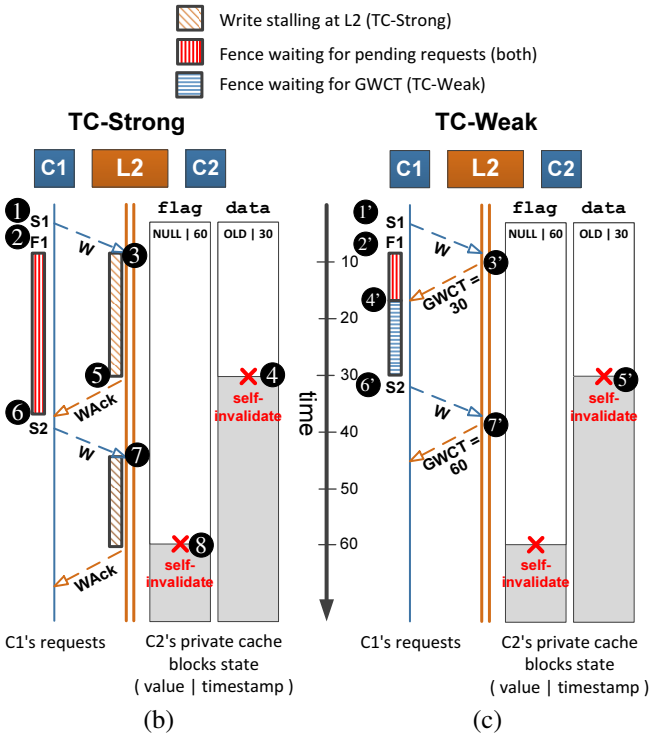
TC-Strong implements release consistency with write atomicity [19]. It uses write-through L1's and a writeback L2. TC-Strong requires synchronized timestamp counters at the GPU cores and L2 controllers shown in Figure 5(a) to provide the components with the current system time. A small timestamp field is added to each cache line in the L1 and L2 caches, as shown in Figure 5(b). The local timestamp value in the L1 cache line indicates the time until the particular cache line is valid. An L1 cache line with a local timestamp less than the current system time is invalid. The global timestamp value in the L2 indicates a time by when all L1 caches will have self-invalidated this cache line.

6.2.1 TC-Strong Operation

Every load request checks both the tag and the local timestamp of the L1 line. It treats a valid tag match but an expired local timestamp as a miss; self-invalidating an L1 block does not require explicit events. A load miss at the L1 generates a request to the L2 with a lifetime prediction. The L2 controller updates the global timestamp to the maximum of the current global timestamp and the requested local timestamp to accommodate the amount of time requested. The L2 responds to the L1 with the data and the global timestamp. The L1 updates its data and local timestamp with values in the response message before completing the load. A store request writes through to the L2 where its completion is delayed until the global timestamp has expired.

Core C1	Core C2
S1: `data = NEW`	**L1**: `r1 = flag`
F1: `FENCE`	**B1**: `if (r1 ≠ SET) goto L1`
S2: `flag = SET`	**L2**: `r2 = data`

(a)



(b) (c)

Figure 6. TC coherence. (a) Code snippet from [58]. (b) Sequence of events for C1 (left) that occur due to code in (a) and state of C2's blocks (right) for TC-Strong. (c) Sequence of events with TC-Weak.

Figure 6(b) illustrates how TC-Strong maintains coherence. The code snippet shown in Figure 6(a) is an example from Sorin *et al.* [58] and represents a common programming idiom used to implement non-blocking queues in pipeline parallel applications [20]. Figure 6(b) shows the memory requests generated by core C1 on the left, and the state of the two memory locations, `flag` and `data`, in C2's L1 on the right. Initially, C2 has `flag` and `data` cached with local timestamps of 60 and 30, respectively. For simplicity, we assume that C2's operations are delayed.

C1 executes instruction S1 and generates a write request to L2 for `data` (❶), and subsequently issues the memory fence instruction F1 (❷). F1 defers scheduling the wavefront because the wavefront has an outstanding store request. When S1's store request reaches the L2 (❸), the L2 stalls it because `data`'s global timestamp will not expire until time T=30. At T=30, C2 self-invalidates `data` (❹), and the L2 processes S1's store (❺). The fence instruction completes when C1 receives the acknowledgment for S1's request (❻). The same sequence of events occurs for the store to `flag` by S2. The L2 stalls S2's write request (❼) until `flag` self-invalidates in C2 (❽).

L2 Eviction Optimization. Evictions at the write-through L1s do not generate messages to the L2. Only expired global timestamps can be evicted from the L2 to maintain inclusion. TC-Strong uses L2 MSHR entries to store unexpired timestamps.

Private Write Optimization. TC-Strong implements an optimization to eliminate write-stalling for private data. It differentiates the single valid L2 state into two stable states, *P* and *S*. The *P* state indicates private data while the *S* state indicates shared data. An L2 line read only once exists in *P*. Writes to L2 lines in *P* are private writes if they are from the core that originally performed the read. In TC-Strong, store requests carry the local timestamp at the L1, if it exists, to the L2. This timestamp is matched to the global timestamp at the L2 to check that the core that originally performed the read is performing a private write.

6.2.2 TC-Strong and LCC comparison

Both LCC and TC-Strong use time-based self-invalidation and require synchronized counters and timestamps in L1 and L2. Both protocols stall writes at the last level cache to unexpired timestamps.

TC-Strong requires minimal hardware modifications to the baseline non-coherent GPU architecture. It supports multiple outstanding write requests per GPU wavefront. In contrast, LCC assumes only one outstanding write request per core. By relaxing the memory model and utilizing the point-to-point ordering guarantee of the baseline GPU memory system, TC-Strong provides much greater memory level parallelism for the thousands of concurrent scalar threads per GPU core.

LCC stalls evictions of unexpired L2 blocks. TC-Strong removes this stalling by allocating an L2 MSHR entry to store the unexpired timestamp. This reduces expensive stalling of the in-order GPU L2 cache controllers. LCC also penalizes private read-write data by stalling writes to private data until the global timestamp expires. The private write optimization in TC-Strong detects and eliminates these stalls.

6.3 TC-Weak Coherence

This section describes TC-Weak. TC-Weak relaxes the write atomicity of TC-Strong. As we show in Section 8.3, doing so improves performance by 28% and lowers interconnect traffic by 26% compared to TC-Strong.

TC-Strong and LCC enforce coherence across all data by stalling writes. TC-Weak uses the insight that GPU applications may contain large amounts of data which does not require coherence and is unnecessarily penalized by write-stalling. By relaxing write-atomicity, TC-Weak eliminates write-stalling and shifts any potential stalling to explicit

Table 2. Complete TC-Weak Protocol (Left: L1 FSM, Right: L2 FSM). Shaded regions indicate additions to non-coherent protocol.

State	Processor Request: Load	Store	Atomic	L1 Action: Eviction	Expire	From L2: Data	Write Ack
I	GETS → **I_V**	GETX → **I_I**	ATOMIC → **I_I**	×	×	×	×
V	hit	UPGR → **V_M**	ATOMIC → **I_I**	evict → **I**	→ **I**	×	×
V_M (upgrade)	hit	UPGR	ATOMIC → **I_I**	stall	→ **I_I**	write done update GWCT *(pending 0?)* → **V**	write done *(global?)* update GWCT *(pending 0?)* → **V**
I_V (Rd-miss)	×	GETX → **I_I**	ATOMIC → **I_I**	stall	×	read done → **V**	×
I_I (Wr-miss)	GETS	GETX	ATOMIC	stall	×	*(read?)* read done *(write/atomic?)* write/atomic done update GWCT *(pending 0?)* → **I**	write done *(global?)* update GWCT *(pending 0?)* → **I**

State	L1 Request: GETS	GETX	UPGR	ATOMIC	L2 Action: Evict	Expire	From Mem: Mem Data
I	FETCH → **I_S**	FETCH → **I_M**	FETCH → **I_M**	FETCH → **I_M**	×	×	×
P (Private)	extend TS DATA → **S**	TS++ ACK-G	TS++ *(TS==?)* ACK *– else –* DATA-G	TS++ DATA-G	*(dirty?)* WB → **M_I**	→ **E**	×
S (Shared)	extend TS DATA	TS++ ACK-G → **P**	TS++ *(TS==?)* ACK-G *– else –* DATA-G → **P**	TS++ DATA-G	*(dirty?)* WB → **M_I**	→ **E**	×
E (Expired)	extend TS DATA → **P**	TS++ ACK	TS++ ACK	TS++ DATA-G	*(dirty?)* WB → **I**	×	×
I_S (Rd miss)	merge	stall	stall	stall	stall	×	DATA *(multiple?)* → **S** *– else –* → **P**
I_M (Wr miss)	stall	stall	stall	stall	stall	×	*(write?)* ACK *(atomic?)* DATA-G → **E**
M_I (Evicted)	FETCH → **I_S**	FETCH → **I_M**	FETCH → **I_M**	FETCH → **I_M**	×	→ **I**	×

L1⇒L2 msgs: GETS (read). GETX (write). ATOMIC. UPGR (upgrade).
L2⇒L1 msgs: ACK (write done). ACK-G (ACK with GWCT). DATA (data response). DATA-G (DATA with GWCT).
L2⇒MEM msgs: FETCH (fetch data from memory). WB (writeback data to memory).
L2 Events @ L1: Data (valid data). Write Ack (write complete from L2).
L1 Conditionals: *read/write/atomic?* (response to GETS/GETX/ATOMIC?). *global?* (response includes GWCT?). *pending 0?* (all pending requests satisfied?).
L2 Conditionals: *TS==?* (requester's timestamp matches pre-incremented L2 timestamp?). *dirty?* (L2 data modified?). *multiple?* (multiple read requests merged?).
L2 Timestamp Actions: extend TS (extend L2 timestamp according to request). TS++ (increment L2 timestamp).

memory fence operations. This provides two main benefits. First, it eliminates expensive stalling at the shared L2 cache controllers, which affects all cores and wavefronts, and shifts it to scheduling of individual wavefronts at memory fences. A wavefront descheduled due to a memory fence does not affect the performance of other wavefronts. Second, it enforces coherence only when required and specified by the program through memory fences. It implements the RCpc [19] consistency model; a detailed discussion on this is available elsewhere [56].

In TC-Weak, writes to unexpired global timestamps at the L2 do not stall. The write response returns with the global timestamp of the L2 cache line at the time of the write. The returned global timestamp is the guaranteed time by which the write will become visible to all cores in the system. This is because by this time all cores will have invalidated their privately cached stale copies. TC-Weak tracks the global timestamps returned by writes, called *Global Write Completion Times* (GWCT), for each wavefront. A memory fence operation uses this information to deschedule the wavefront sufficiently long enough to guarantee that all previous writes from the wavefront have become globally visible.

As illustrated in Figure 5(a), TC-Weak adds a small GWCT table to each GPU core. The GWCT table contains 48 entries, one for each wavefront in a GPU core. Each entry holds a timestamp value which corresponds to the maximum of all GWCT's observed for that wavefront.

6.3.1 TC-Weak Operation

A memory fence in TC-Weak deschedules a wavefront until all pending write requests from the wavefront have returned acknowledgments, and until the wavefront's timestamp in the GWCT table has expired. The latter ensures that all previous writes have become visible to the system by fence completion.

Figure 6(c) illustrates how coherence is maintained in TC-Weak by showing the execution of C1's memory instructions from Figure 6(a). C1 executes S1 and sends a store request to the L2 for `data` (❶'). Subsequently, C1 issues a memory fence operation (❷') that defers scheduling of the wavefront because S1 has an outstanding memory request. The L2 receives the store request (❸') and returns the current global timestamp stored in the L2 for `data`. In this case, the value returned is 30 and corresponds to C2's initially cached copy. The L2 does not stall the write and sends back an acknowledgment with the GWCT, which updates the C1's GWCT entry for this wavefront. After C1 receives the acknowledgment (❹'), no memory requests are outstanding. The scheduling of the wavefront is now deferred because the GWCT entry of this wavefront containing a timestamp of 30 has not yet expired. As `data` self-invalidates in C2's cache (❺'), the wavefront's GWCT expires and the fence is allowed to complete (❻'). The next store instruction, S2, sends a store request (❻) to the L2 for `flag`. The L2 returns a GWCT time of 60 (❼'), corresponding to the copy cached by C2.

Comparing Figure 6(c) to 6(b) shows that TC-Weak performs better than TC-Strong because it only stalls at explicit memory fence operations. This ensures that writes to data that does not require coherence has minimal impact.

Table 2 presents TC-Weak's complete L1 and L2 state machines in the format used by Martin [36]. Each table entry lists the actions carried out and the final cache line state for a given transition (top) and an initial cache line state (left). The 4 stable L2 states, *I*, *P*, *S* and *E*, correspond to invalid lines, lines with one reader, lines with multiple readers, and lines with expired global timestamps, respectively. The *I_S* and *I_M* L2 transient cache states track misses at the L2 for read and write requests. The *M_I* transient coherent state tracks evicted L2 blocks with unexpired global timestamps. Note the lack of transient states and stalling at the L2 for writes to valid (*P*, *S* and *E*) lines. At the L1, the stable *I* state indicates invalid lines or lines with expired local timestamps, and the stable *V* state indicates valid local timestamps. The *I_V* and *I_I* transient cache states are used to track read and write misses, while the *V_M* transient coherent state tracks write requests to valid lines.

Private Write Optimization. To ensure that memory fences are not stalled by writes to private data, TC-Weak uses a private write optimization similar to the one employed by TC-Strong and described in Section 6.2.1. Write requests to L2 lines in the *P* state where the L1 local timestamp matches the L2 global timestamp indicate private writes and do not return a GWCT. Since TC-Weak does not stall writes at the L2, an L2 line in *P* may correspond to multiple unexpired but stale L1 lines. Writes in TC-Weak always modify the global timestamp by incrementing it by one. This ensures that a write request from another L1 cache with stale data carries a local timestamp that mismatches with the global timestamp at the L2, and that the write response replies with the updated data.

6.4 Lifetime Prediction

Predicted lifetimes should not be too short that L1 blocks are self-invalidated too early, and not too long that storing evicted timestamps wastes L2 cache resources and potentially introduces resource stalls. In Section 8.4 we show that a single lifetime value for all accesses performs well. Moreover, this value is application dependent. Based on this insight, we propose a simple lifetime predictor that maintains a single lifetime prediction value at each L2 cache bank, and adjusts it based on application behaviour. A load obtains its lifetime prediction at the L2 bank.

The predictor updates the predicted lifetime based on events local to the L2 bank. First, the local prediction is decreased by t_{evict} cycles if an L2 block with an unexpired timestamp is evicted. This reduces the number of timestamps that need to be stored past an L2 eviction. Second, the local prediction is increased by t_{hit} cycles if a load request misses at the L1 due to an expired L1 block. This helps reduce L1 misses due to early self-invalidation. The lifetime is also increased by t_{hit} cycles if the L2 receives a load request to a valid block with an expired global timestamp. This ensures that the prediction is increased even if L1 blocks are quickly evicted. Third, the lifetime is decreased by t_{write} cycles if a store operation writes to an unexpired block at the L2. This helps reduce the amount of time that fence operations wait for the GWCT to expire, *i.e.*, for writes to become globally visible. This third mechanism is disabled for applications not using fences as it would unnecessarily increase the L1 miss rate. Table 4 lists the constant values used in our evaluation; we found these to yield the best performance across all applications.

6.5 Timestamp Rollover

L1 blocks in the valid state but with expired timestamps may become unexpired when the global time counters rollover. This could be handled by simply flash invalidating the valid bits in the L1 cache [52]. More sophisticated approaches are possible, but beyond the scope of this work. None of the benchmarks we evaluate execute long enough to trigger an L1 flush with 32-bit timestamps.

7 Methodology

We model a cache coherent GPU architecture by extending GPGPU-Sim version 3.1.2 [8] with the Ruby memory system model from GEMS [38]. The baseline non-coherent memory system and all coherence protocols are implemented in SLICC. The MESI cache coherence protocol is acquired from gem5 [9]. Our GPGPU-Sim extended with Ruby is configured to model a generic NVIDIA Fermi GPU [44]. We use Orion 2.0 [25] to estimate the interconnect power consumption.

The interconnection network is modelled using the detailed fixed-pipeline network model in Garnet [5]. Two crossbars, one per direction, connect the GPU cores to the memory partitions. Each crossbar can transfer one 32-byte flit per interconnect cycle to/from each memory partition for a peak bandwidth of $\sim$175GB/s per direction. GPU cores connect to the interconnection network through private ports. The baseline non-coherent and all coherence protocols use the detailed GDDR5 DRAM model from GPGPU-Sim. Minimum L2 latency of 340 cycles and minimum DRAM latency of 460 cycles (in core cycles) is modelled to match the latencies observed on Fermi GPU hardware via microbenchmarks released by Wong *et al.* [62]. Table 4 lists other major configuration parameters.

We used two sets of benchmarks for evaluation: one set contains inter-workgroup communication and requires coherent caches for correctness, and the other only contains intra-workgroup communication. While coherence can be disabled for the latter set, we kept coherence enabled and used this set as a proxy for future workloads which contain

Table 3. Benchmarks

Inter-workgroup communication		Intra-workgroup communication	
Name	**Abbr.**	**Name**	**Abbr.**
Barnes Hut [13]	BH	HotSpot [15]	HSP
CudaCuts [61]	CC	K-means [15]	KMN
Cloth Physics [12]	CL	3D Laplace Solver [8]	LPS
Dynamic Load Balancing [14]	DLB	Needleman [15]	NDL
Stencil (Wave Propagation)	STN	Gaussian Filter [1]	RG
Versatile Place and Route	VPR	Anisotropic Diffusion [15]	SR

Table 4. Simulation Configuration

GPGPU-Sim Core Model	
# GPU Cores	16
Core Config	48 Wavefronts/core, 32 threads/wavefront, 1.4Ghz Pipeline width:32, #Reg: 32768 Scheduling: Loose Round Robin. Shared Mem.: 48KB
Ruby Memory Model	
L1 Private Data$	32KB, 4way. 128B line, 4-way assoc. 128 MSHRs
L2 Shared Bank	128KB, 8-way, 128B line, 128 MSHRs. Minimum Latency: 340 cycles, 700 MHz
# Mem. Partitions	8
Interconnect	1 Crossbar/Direction. Flit: 32bytes Clock: 700 MHz. BW: 32 (Bytes/Cycle). (175GB/s/Direction)
Virtual Channels	8-flit buffer per VC.
# Virtual Networks	Non-coherent: 2. TC-Strong and TC-Weak: 2. MESI: 5. GPU-VI and GPU-VIni: 4.
GDDR Clock	1400 MHz
Memory Channel BW	8 (Bytes/Cycle) (175GB/s peak). Minimum Latency: 460 cycles
DRAM Queue Capacity	32. Out-of-Order (FR-FCFS)
GDDR5 Memory Timing	t_{CL}=12 t_{RP}=12 t_{RC}=40 t_{RAS}=28 t_{CCD}=2 t_{WL}=4 t_{RCD}=12 t_{RRD}=6 t_{CDLR}=5 t_{WR}=12 t_{CCDL}=3 t_{RTPL}=2
TC-Weak Parameters	
Timestamp Size	32 bits
Predictor Constants	t_{evict}=8 cycles. t_{hit}=4 cycle. t_{write}=8 cycles.

both data needing coherence and data not needing it. The following benchmarks fall into the former set:

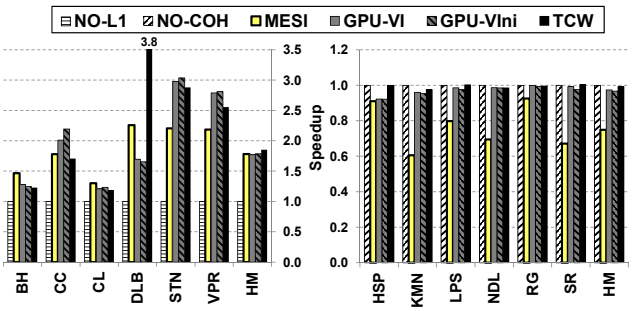
Barnes Hut (BH) implements the Barnes Hut n-body algorithm in CUDA [13]. We report data for the tree-building kernel which iteratively builds an octree of 30000 bodies.

CudaCuts (CC) implements the maxflow/mincut algorithm for image segmentation in CUDA [61]. We optimized CC by utilizing a coherent memory space to combine the push, pull and relabel operations into a single kernel, improving performance by 30% as a result.

Cloth Physics (CL) is a cloth physics simulation based on "RopaDemo" [12]. We focus on the Distance Solver kernel which adjusts cloth particle locations using a set of constraints to model a spring-mass system.

Dynamic Load Balancing (DLB) implements task-stealing in CUDA [14]. It uses non-blocking task queues to load balance the partitioning of an octree. We report data for an input graph size of 100000 nodes.

Stencil (STN) uses stencil computation to implement a finite difference solver for 3D wave propagation useful in seismic imaging. Each workgroup processes a subset of the stencil nodes. Each node in the stencil communicates with 24 adjacent neighbours. A coherent memory space ensures that updates to neighbours in a different subset are visible. STN uses fast barriers [55] to synchronize workgroups between computational time steps.



(a) Inter-workgroup comm. (b) Intra-workgroup comm.

Figure 7. Performance of coherent and non-coherent GPU memory systems. HM = harmonic mean.

Versatile Place and Route (VPR) is a placement tool for FPGAs. We ported the simulated annealing based placement algorithm from VTR 1.0 [49] to CUDA. We simulate one iteration in the annealing schedule for the *bgm* circuit. VPR on GPU hardware with disabled L1 caches performs 4x faster over the serial CPU version.

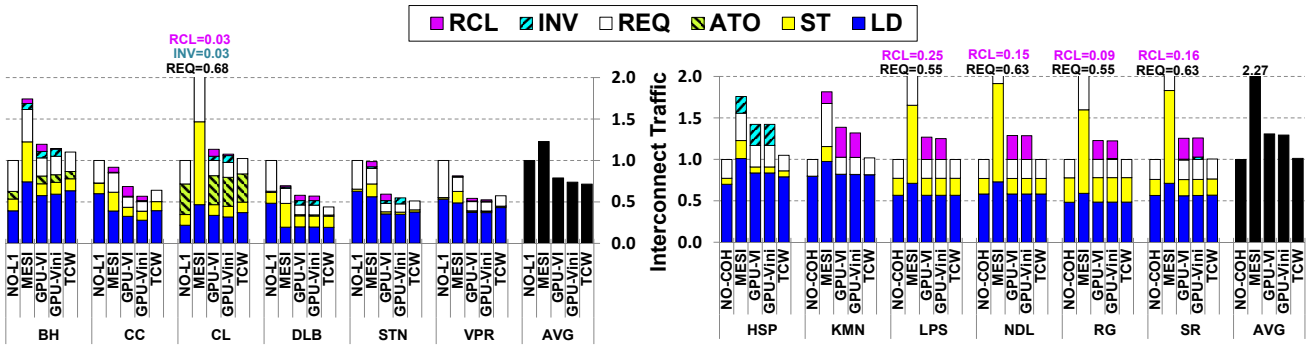
The set of benchmarks with intra-workgroup communication is chosen from the Rodinia benchmark suite [15], benchmarks used by Bakhoda *et al.* [8] and the CUDA SDK [1]. These benchmarks were selected to highlight a variety of behaviours; we did not exclude any benchmarks where TC-Weak performed worse than other protocols. All benchmarks we evaluate are listed in Table 3.

8 Results

This section compares the performance of the coherence protocols on a GPU. Section 8.3 compares TC-Weak to TC-Strong. TCW implements TC-Weak with the lifetime predictor described in Section 6.4.

8.1 Performance and Interconnect Traffic

Figure 7(a) compares the performance of coherence protocols against a baseline GPU with L1 caches disabled (NO-L1) for applications with inter-workgroup communication. Figure 7(b) compares them against the non-coherent baseline protocol with L1 caches enabled (NO-COH) for applications with intra-workgroup communication. TCW achieves a harmonic mean 85% performance improvement over the baseline GPU for applications with inter-workgroup communication. While all protocols achieve similar average performance for applications with inter-workgroup communication, MESI performs significantly worse compared to the write-through protocols on applications without such communication. This is a result of MESI's L1 writeback write-allocate policy which favours write locality but introduces unnecessary traffic for write-once access patterns common in GPU applications. The potentially larger effective cache capacity in non-inclusive GPU-VIni adds no performance benefit over the inclusive GPU-VI. In DLB, each workgroup fetches and inserts tasks



(a) Inter-workgroup communication (b) Intra-workgroup communication

Figure 8. Breakdown of interconnect traffic for coherent and non-coherent GPU memory systems.

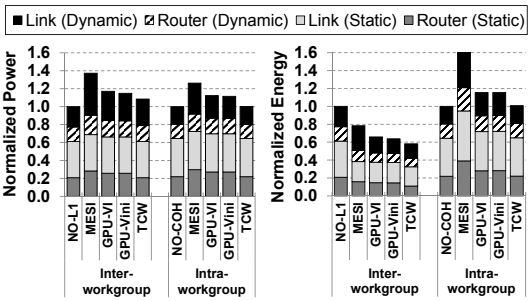


Figure 9. Breakdown of interconnect power and energy.

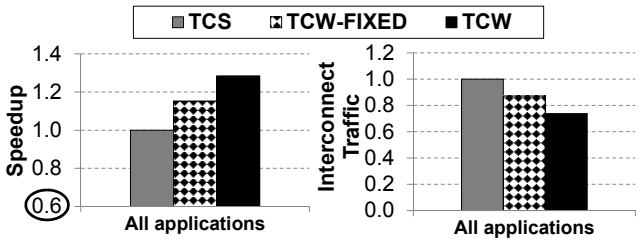


Figure 10. (a) Harmonic mean speedup. (b) Normalized average interconnect traffic.

into a shared queue. As a result, the task-fetching and task-inserting invalidation latencies lie on the critical path for a large number of threads. TCW eliminates this critical path invalidation latency in DLB and performs up to 2x faster than the invalidation-based protocols.

Figures 8(a) and 8(b) show the breakdown of interconnect traffic between different coherence protocols. LD, ST, and ATO are the data traffic from load, store, and atomic requests. MESI performs atomic operations at the L1 cache and this traffic is included in ST. REQ refers to control traffic for all protocols. INV and RCL are invalidation and recall traffic, respectively.

MESI's write-allocate policy at the L1 significantly increases store traffic due to unnecessary refills of write-once data. On average, MESI increases interconnect traffic over the baseline non-coherent GPU by 75% across all applications. The write-through GPU-VI and GPU-VIni introduce unnecessary invalidation and recall traffic, averaging to a traffic overhead of 31% and 30% for applications without inter-workgroup communication. TCW removes all invalidations and recalls and as a result reduces interconnect traffic by 56% over MESI, 23% over GPU-VI and 23% over GPU-VIni for this set of applications.

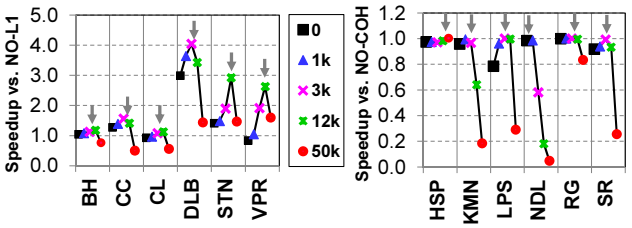
8.2 Power

Figure 9 shows the breakdown of interconnect power and energy usage. TCW lowers the interconnect power usage by 21%, 10% and 8%, and interconnect energy usage by 36%, 13% and 8% over MESI, GPU-VI and GPU-VIni, respectively. The reductions are both in dynamic power, due to lower interconnect traffic, and static power, due to fewer virtual channel buffers in TCW.

8.3 TC-Weak vs. TC-Strong

Figures 10(a) and 10(b) compare the harmonic mean performance and average interconnect traffic, respectively, across all applications for TC-Strong and TC-Weak. TCS implements TC-Strong with the FIXED-DELTA prediction scheme proposed in LCC [34, 54], which selects a single fixed lifetime that works best across all applications. TCS uses a fixed lifetime prediction of 800 core cycles, which was found to yield the best harmonic mean performance over other lifetime values. TCW-FIXED uses TC-Weak and a fixed lifetime of 3200 core cycles, which was found to be best performing over other values. TCW implements TC-Weak with the proposed predictor, as before.

TCW-FIXED has the same predictor as TCS but outperforms it by 15% while reducing traffic by 13%. TCW achieves a 28% improvement in performance over TCS and reduces interconnect traffic by 26%. TC-Strong has a trade-off between additional write stalls with higher lifetimes and additional L1 misses with lower lifetimes. TC-Weak avoids this trade-off by not stalling writes. This permits longer lifetimes and fewer L1 misses, improving performance and reducing traffic over TC-Strong.



(a) Inter-workgroup comm. (b) Intra-workgroup comm.

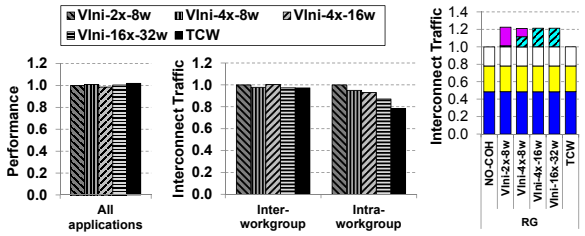
Figure 11. Speedup with different fixed lifetimes for TCW-FIXED. ↓ indicates average lifetime observed on TCW.

8.4 TC-Weak Performance Profile

Figure 11 presents the performance of TC-Weak with various fixed lifetime prediction values for the entire duration of the application. The downward arrows in Figure 11 indicate the average lifetime predictions in TCW. An increase in performance with increasing lifetimes results from an improved L1 hit rate. A decrease in performance with larger lifetimes is a result of stalling fences and L2 resource stalls induced by storage of evicted but unexpired timestamps. Note that in DLB, TCW-FIXED with a lifetime of 0 is 3x faster than NO-L1 because use of L1 MSHRs in TCW-FIXED reduces load requests by 50% by merging redundant requests across wavefronts. The performance profile yields two main observations. First, each application prefers a different fixed lifetime. For example, NDL's streaming access pattern benefits from a short lifetime, or an effectively disabled L1. Conversely, HSP prefers a large lifetime to fully utilize the L1 cache. Second, the arrows indicating TCW's average lifetime lie close to the peak performance lifetimes for each application. Hence, our simple predictor can effectively locate the best fixed lifetime for each benchmark for these applications.

8.5 Directory Size Scaling

Figures 12(a) and 12(b) compare the performance and traffic of TCW to GPU-VIni with directories ranging from 8-way associative and 2x the number of entries as total L1 blocks (VIni-2x-8w) to 32-ways and 16x the number of L1 blocks (VIni-16x-32w). In Figure 12(a), directory size and associativity have no impact on performance of GPU applications. In Figure 12(b), while high associativity and large directory sizes reduce the coherence traffic overheads in intra-workgroup communication, they cannot eliminate them. Figure 12(c) shows the breakdown in RG's traffic for these directory configurations. As the directory size is increased from 2x to 16x, the reduction in recall traffic is offset by the increase in invalidation traffic due to inter-kernel communication. Hence, while larger directories may reduce recall traffic, the coherence traffic cost of true communication cannot be eliminated. TCW is able to eliminate both sources of coherence traffic overheads by using synchronized time to facilitate communication.



(a) (b) (c)

Figure 12. Performance (a) and traffic (b) with different GPU-VIni directory sizes. (c) Traffic breakdown for RG benchmark (same labels as Figure 8).

9 Conclusion

This paper presents and addresses the set of challenges introduced by GPU cache coherence. We find that conventional coherence implementations are not well suited for GPUs. The management of transient state for thousands of in-flight memory accesses adds hardware and complexity overhead. Coherence adds unnecessary traffic overheads to existing GPU applications. Accelerating applications with both coherent and non-coherent data requires that the latter introduce minimal coherence overheads.

We present Temporal Coherence, a timestamp based coherence framework that reduces overheads of GPU coherence. We propose an implementation of this framework, TC-Weak, which uses novel timestamp based memory fences to reduce these overheads.

Our evaluation shows that TC-Weak with a simple lifetime predictor reduces the traffic of MESI, GPU-VI and GPU-VIni directory coherence protocols by 56%, 23% and 22% across a set of applications without coherent data. Against TC-Strong, a TC protocol based on LCC, TC-Weak performs 28% faster with 26% lower interconnect traffic. It also provides a 85% speedup over disabling the non-coherent L1's for a set of applications that require coherent caches. A TC-Weak-enhanced GPU provides programmers with a well understood memory consistency model and simplifies the development of irregular GPU applications.

Acknowledgments

We thank Mark Hill, Hadi Jooybar, Timothy Rogers and the anonymous reviewers for their invaluable comments. This work was partly supported by funding from Natural Sciences and Engineering Research Council of Canada and Advanced Micro Devices, Inc.

References

[1] NVIDIA CUDA SDK code samples. http://developer.nvidia.com/cuda-downloads.

[2] *The PowerPC architecture: a specification for a new family of RISC processors*. Morgan Kaufmann Publishers, 1994.

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12), 1996.

[4] S. V. Adve and M. D. Hill. Weak ordering a new definition. In *ISCA*, 1990.

[5] N. Agarwal *et al.* GARNET: A detailed on-chip network model inside a full-system simulator. In *ISPASS*, 2009.

[6] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide*, May 2012.

[7] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA*, 1988.

[8] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[9] N. Binkert *et al.* The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[10] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[11] N. Brookwood. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience, 2010.

[12] A. Brownsword. Cloth in OpenCL. GDC, 2009.

[13] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. *Chapter 6 in GPU Computing Gems Emerald Edition*, 2011.

[14] D. Cederman and P. Tsigas. On dynamic load balancing on graphics processors. In *EUROGRAPHICS*, 2008.

[15] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[16] B. Choi *et al.* DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, 2011.

[17] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2), 2007.

[18] J. Feehrer *et al.* Coherency Hub Design for Multisocket Sun Servers with CoolThreads Technology. *IEEE Micro*, 2009.

[19] K. Gharachorloo *et al.* Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA*, 1990.

[20] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP*, 2008.

[21] T. H. Hetherington *et al.* Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. In *ISPASS*, 2012.

[22] S. Hong *et al.* Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, 2011.

[23] Intel. *Intel 64 and IA-32 Architectures Software Developers Manual*, May 2012.

[24] N. P. Jouppi. Cache write policies and performance. In *ISCA*, 1993.

[25] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. In *DATE*, 2009.

[26] S. Keckler *et al.* GPUs and the Future of Parallel Computing. *IEEE Micro*, 31, 2011.

[27] J. Kelm *et al.* Cohesion: a hybrid memory model for accelerators. In *ISCA*, 2010.

[28] J. Kelm *et al.* WAYPOINT: scaling coherence to thousand-core architectures. In *PACT*, 2010.

[29] Khronos Group. OpenCL. http://www.khronos.org/opencl/.

[30] P. Kongetira *et al.* Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2), 2005.

[31] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, 1997.

[32] H. Q. Le *et al.* IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51(6), 2007.

[33] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *ISCA*, 1995.

[34] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas. Memory coherence in the age of multicores. In *ICCD*, 2011.

[35] J. Manson *et al.* The Java Memory Model. In *POPL*, 2005.

[36] M. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, 2003.

[37] M. Martin *et al.* Timestamp snooping: an approach for extending SMPs. In *ASPLOS*. 2000.

[38] M. Martin *et al.* Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.

[39] M. Martin *et al.* Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7), 2012.

[40] M. Mendez-Lojo et al. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, 2012.

[41] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *PPoPP*, 2012.

[42] S. L. Min and J. L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Trans. Parallel Distrib. Syst.*, 3(1), 1992.

[43] S. K. Nandy and R. Narayan. An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems. In *IWPP*, 1994.

[44] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[45] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[46] NVIDIA Corp. *CUDA C Programming Guide v4.2*, 2012.

[47] F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1), 1997.

[48] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *PACT*, 2012.

[49] J. Rose *et al.* The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *FPGA*, 2012.

[50] S. Rusu *et al.* A 45 nm 8-Core Enterprise Xeon Processor. *J. Solid-State Circuits*, 45(1), 2010.

[51] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *HPCA*, 2012.

[52] J.-P. Schoellkopf. SRAM memory device with flash clear and corresponding flash clear method. Patent. US 7333380, 2008.

[53] D. Seal. *ARM Architecture Reference Manual*. 2000.

[54] K. S. Shim *et al.* Library Cache Coherence. Csail technical report mit-csail-tr-2011-027, May 2011.

[55] Shucai Xiao and Wu-chun Feng. Inter-block gpu communication via fast barrier synchronization. In *IPDPS*, 2010.

[56] I. Singh et al. Temporal Coherence: Hardware Cache Coherence for GPU Architectures. Technical report, University of British Columbia, 2013.

[57] R. L. Sites. *Alpha architecture reference manual*. 1992.

[58] D. J. Sorin *et al.* *A Primer on Memory Consistency and Cache Coherence*. Morgan and Claypool Publishers, 2011.

[59] C. SPARC International, Inc. *The SPARC architecture manual (version 9)*. 1994.

[60] Sun Microsystems. *OpenSPARC T2 Core Microarchitecture Specification*, 2007.

[61] V. Vineet and P. Narayanan. CudaCuts: Fast Graph Cuts on the GPU. In *CVPRW*, 2008.

[62] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, 2010.

[63] X. Yuan, R. G. Melhem, and R. Gupta. A Timestamp-based Selective Invalidation Scheme for Multiprocessor Cache Coherence. In *ICPP, Vol. 3*, 1996.

[64] H. Zhao *et al.* SPATL: Honey, I Shrunk the Coherence Directory. In *PACT*, 2011.