Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence

Xiaowei Ren  
Mieszko Lis  
The University of British Columbia  
{xiaowei,mieszko}@ece.ubc.ca

Abstract—Recent work has argued that sequential consistency (SC) in GPUs can perform on par with weak memory models, provided ordering stalls are made less frequent by relaxing ordering for private and read-only data. In this paper, we address the complementary problem of reducing stall latencies for both read-only and read-write data.

We find that SC stalls are particularly problematic for workloads with inter-workgroup sharing, and occur primarily due to earlier stores in the same thread; a substantial part of the overhead comes from the need to stall until write permissions are obtained (to ensure write atomicity). To address this, we propose RCC, a GPU coherence protocol which grants write permissions without stalling but can still be used to implement SC. RCC uses logical timestamps to determine a global memory order and L1 read permissions; even though each core may see a different logical “time,” SC ordering can still be maintained.

Unlike previous GPU SC proposals, our design does not require invasive core changes and additional per-core storage to classify read-only/private data. For workloads with inter-workgroup sharing overall performance is 29% better and energy is 25% less than in best previous GPU SC proposals, and within 7% of the best non-SC design.

I. Introduction

Modern processors and GPUs can support multiple inflight memory requests not only from different cores but also from independent instructions in the same thread. This can result in memory operations appearing to execute out of order: two cores — or even two instructions in the same thread — could potentially observe memory writes in different order, leading to difficult-to-debug synchronization bugs. To constrain the range of allowable behaviour, processors and programming languages define memory models, which specify precisely which writes a memory read may observe.

Sequential consistency (SC) — the most intuitive model — requires that (a) all memory accesses appear to execute in program order and (b) all threads observe writes in the same sequence [1]. To ensure in-order load/store execution, a thread must delay issuing some memory operations until preceding writes complete; we refer to these delays as SC stalls. Moreover, since all cores must observe writes in the same order, stores cannot complete until they are guaranteed to be visible to all other threads and cores. Because of these restrictions, few modern commercial CPUs have supported SC [2]; typically SC is relaxed to permit limited [3, 4] or near-arbitrary reordering [5–8]; programmers must then insert memory fences for specific memory operations, in essence manually reintroducing SC stalls. GPUs manufacturers have followed suit: both NVidia and AMD GPUs exhibit weak consistency [9] similar to WO [10] or RC [11] models.

Correctly inserting fences is difficult, however, especially in GPUs where all practical programs are concurrent and performance-sensitive. The authors of [9] found missing fences in a variety of peer-reviewed publications, and even vendor guides [12]. Such bugs are very difficult to detect: some occurred in as few as 4 out of 100,000 executions in real hardware, and most occurred in fewer than 1% of executions [9]. Code fenced properly for a specific GPU may not even work correctly on other GPUs from the same vendor: some of these bugs were observable in Fermi and Kepler but not in older or newer microarchitectures [9].

SC hardware is desirable, then, if it can be implemented without significant performance loss. Recent work [13, 14] has argued that this is possible in GPUs: unlike CPUs, which lack enough instruction-level parallelism (ILP) to cover the additional latency of SC stalls, GPUs can leverage abundant thread-level parallelism (TLP) to cover most SC stalls. The authors of [14] propose reducing the frequency of the remaining SC stalls by relaxing SC for read-only and private data; classifying these at runtime, however, requires complex changes to GPU core microarchitecture and carries an area overhead in devices where silicon is already at a premium. Moreover, both studies focused on SC built using CPU coherence protocols (MOESI and MESI) with write-back L1 caches. In GPUs, however, write-through L1s perform better [15]: GPU L1 caches have very little space per thread, so a write-back policy brings infrequently written data into the L1 only to write it back soon afterwards. Commercial
GPUs have write-through L1s and require bypassing/flushing L1 caches to ensure intra-GPU coherence [16–18]. Compared to the best GPU relaxed consistency design, the performance cost of implementing SC appears to be closer to 30% [15].

To trace the roots of this performance loss, we evaluated an SC implementation similar to prior work [13, 14] but with GPU-style write-through L1 caches (see Sec. IV-A for simulation setup). We examined memory-intensive workloads with and without inter-workgroup sharing previously used to evaluate GPU cache coherence [15]: the inter-workgroup benchmarks rely on inter-core coherence traffic, while the intra-workgroup benchmarks communicate only within each GPU core. We found SC stalls to be relatively infrequent (Fig 1a): in only one case were more than 20% memory operations ever stalled because of SC; this supports prior arguments [13] that the massive parallelism available in GPUs can cover most ordering stalls introduced by SC.

We next examined the cause of each stall — i.e., the type of the preceding memory operation from the same thread. Fig. 1b shows that most SC stall cycles are spent waiting for a previous store (or atomic) instruction to complete; indeed, in most cases, nearly all stall delays are due to waiting for prior writes. This is because average store latencies are very long: for workloads with inter-threadblock communication, store latencies are often much longer than load latencies (2.4× gmean), and up to 3.7× longer (Fig. 1c).

This makes sense: to maintain SC, each store must receive an ACK before completing to ensure that the new value has become visible to all cores. There are two parts to this latency: one — the round-trip to L2 — is unavoidable with the write-through L1 caches found in GPUs. The other part is ensuring exclusive coherence permissions: in our MESI-based experiment the write waits until other sharers have invalidated their copies, while in timestamp-based GPU coherence protocols like TC-STRONG [15] the store waits for all read leases to expire. Long-latency stores can affect performance not only by delaying SC stall resolution, but also by occupying buffer space or stalling same-cache-line stores from other threads in MSHRs until the ACK is received.

To find out whether coherence delays are significant, we implemented an idealized variant of SC where acquiring read and write permissions is instant (SC-Ideal). Fig. 1d shows the speedup of SC-Ideal over realistic SC: for workloads with inter-workgroup sharing, idealizing coherence yields a substantial performance improvement (1.6× gmean); workloads with only intra-workgroup sharing see no benefit.

To address this, we leverage Lamport’s observation that ordering constraints need to be maintained only in logical time [20], prior observations that SC can be maintained logically [21, 22], and the recent insight that logical timestamps can be used directly to implement a coherence protocol [23]. We propose Relativistic Cache Coherence (RCC), a simple, two-state GPU coherence protocol where each core maintains — and independently advances — its own logical time. The L2 keeps track of the last logical write time for each cache block; whenever a core accesses the L2, it must ensure that its own logical time exceeds the last write time of the relevant block. Data may be cached in L1s for a limited (logical) time, after which the block self-invalidates.

Fig. 2 shows how RCC maintains SC in logical time. First, core 0 loads address A, and receives a fixed-time lease for A from the L2, which records the lease duration; core 0 may then read its L1 copy until its logical time exceeds the lease expiration time. Core 1 writes to A, but to do this it must advance its own logical time to past the lease given out for A; this step (dashed line) is equivalent to establishing write permissions in other protocols, but occurs instantly in RCC. Core 2 loads A from L2 and advances its logical time past the time of core 1’s write. Finally, core 3 also reads A. The load is logically before the store to A (because core 3’s logical clock is earlier than A’s), but physically the write to A has already happened, and only the new value of A is available at the L2. Core 3 thus receives the new value of A, but must also advance its logical time to that of A’s write.

Naturally, the cost of synchronization does not entirely disappear: advancing a core’s logical time may cause other L1 cache blocks to expire. In essence, we are exchanging a reduction in store latency for A for potentially some additional L1 misses on other addresses. While this would be problematic for latency-sensitive CPUs, throughput-focused GPUs were explicitly designed to amortize this kind of cost; we will show that in GPUs this tradeoff is worth making.

Lamport’s logical time has recently been proposed as a coherence mechanism for CPUs [23, 24]. Performance, however, was subpar even compared to the much simpler MSI protocol, even though the proposed protocol was more complex than RCC and relied on complex speculation-and-rollback mechanism. RCC is not only much simpler, but actually outperforms the best existing GPU protocols.

In the rest of this paper, we describe RCC and demonstrate...
how it addresses the store latency and SC stall problems
identified above. In contrast with prior GPU SC work [14],
RCC does not explicitly classify read-only/private data;
instead, a predictor naturally learns to assign short cache
lifetimes to frequently written shared data. Unlike prior GPU
coherence work [15], RCC operates in logical time; as a
result, stores acquire write permissions instantly but still
maintain SC. RCC underpins a sequentially coherent GPU
memory system that outperforms all previous proposals and
closes the gap between SC and weak consistency in GPUs.

The contributions of our work are:
- we trace the cost of SC overheads in realistic GPUs to
  the need to acquire write permissions for shared data;
- we propose RCC, a simple two-state GPU coherence
  protocol that significantly improves store performance;
- we demonstrate that an SC implementation using RCC
  significantly reduces SC stall rates and resolve latencies,
  and outperforms the best prior GPU proposal by 29%;
- we close the performance gap between best SC and
  weak consistency proposals for GPUs to within 7%.

II. Background

A. Consistency and coherence

Consistency. A memory consistency model defines which
sequences of values may be legally returned from the
sequence of load operations in each program thread. For
example, the following code snippet from [25] represents
a common synchronization pattern found in many inter-
workgroup sharing workloads (e.g., work queues in DLB):

```c
core C0 | core C1

data = new

done = true

weaken ordered models

need a memory fence here

while (!done) {
    // wait for new data value
    ... use new data...
}
```

The question is, should core C1 be allowed to see done=true
even if data=old? This is clearly not the intended behaviour,
since C1 could see a stale copy of data; nevertheless, it is
allowed by many commercial CPUs and all extant GPUs [9].

Sequencial Consistency [1] most closely corresponds to
most programmers’ intuition: it requires that (a) memory
operations appear to execute and complete in program order,
and (b) all threads observe stores in the same global sequence.
In SC, an execution where done=true when data=old is
illegal because either (a) the writes to data and done were
executed out of order by core C0, or (b) they were executed
in one order by C0 but observed in a different order by C1.

Weak consistency models, on the other hand, allow near-
unrestricted reordering of loads and stores in the program,
provided that data dependencies are respected; such reordering
typically occurs during compilation and during execution
in the processor. Special memory fence instructions must be
used to restrict reordering and restore sequentially consistent
behaviour: in the example above, a fence is needed to ensure
that the store to data completes before the store to done. As
discussed in Sec. I, missing fences can be very difficult to find
in a massively multithreaded setting like a GPU; conversely,
adding too many fences compromises performance.

Since compilers can reorder or elide memory references
(e.g., via register allocation), a programming language must
also define a memory model. Due to the range of consistency
models present in extant CPUs, languages like Java [26] or
C++ [27] guarantee sequentially consistent semantics only for
programs that are data-race-free (i.e., properly synchronized
and fenced); this is known as DRF-0 [28]. The HRF model
recently proposed for hybrid CPU/GPU architectures further
constrains DRF-0 by requiring proper scoping [29].

Coherence. In systems with private caches, a cache
coherence protocol ensures that writes to a single location are
ordered and become visible in the same order to all cores [30];
the aim is to make caches logically transparent. Since caches
are ubiquitous, providing coherence is a fundamental part of
implementing any memory consistency model.

Not all coherence protocols can support SC. The best
prior GPU coherence protocol TC-weak [15] allows stores to
proceed without exclusive write permissions (unless properly
fenced); while this yields a 30% performance improvement, it
compromises write atomicity, which is necessary for SC [31].
RCC performs close to TC-weak without giving up SC.

B. GPUs vs. CPUs: a consistency and coherence perspective

Consistency. Modern multicore CPUs have largely settled
on weak memory models to enable reordering in-flight
memory operations [3–7]; because CPUs support at most a
few hardware threads, the memory-level parallelism (MLP)
obtained from reordering memory operations is key to
performance. GPUs, on the other hand, buffer many tens of
warps (e.g., 48–64 [16–18]) of 32–64 threads in each GPU
core (SM), and when one warp is stalled (because of an L1
cache miss, for example), the core simply executes another.

With fine-grained multithreading, GPUs can amortize
hundreds of cycles of latency without reordering memory
operations; recent work [13, 14] has suggested that the same
mechanism can cover the ordering stalls required by SC.
Indeed, hardware techniques that reorder accesses — such
as store buffers — are either too expensive or ineffective in
GPUs, so leaving them out does not hurt performance [14].

Coherence. CPU caches are generally kept coherent by
tracking each block’s sharers and invalidating all copies
before writing the block. Most protocols in commercial
products are quite similar: they have slightly different states
(MESI, MESIF, MOESI, etc.) or sharer tracking methods, but
the basic operation relies on request-reply communication
between cores and an ordering point such as a directory.

All commercial GPUs we are aware of lack automatic
coherence among private L1 caches: in GPU vendor literature,
“coherence” refers only to the boundary between the host CPU
and the GPU. NVidia Pascal allows the GPU to initiate page faults and synchronize GPU and CPU memory spaces [32], but intra-GPU coherence requires bypassing the L1 caches [9]. AMD Kaveri APUs bypass and flush the L1 cache for intra-GPU coherence, and bypass the L2 for CPU-GPU sharing [33]. Details for ARM MALI GPUs are scant, but it appears that the coherence boundary terminates at the GPU shared L2 cache and does not include the L1s [34].

Efficient intra-GPU coherence implementations are subject to different constraints than CPUs. GPUs have 15, 32, or even 56 SM cores [16–18, 32], simultaneously executing around 100,000 threads. While some prior studies [13, 14] (and our motivation study in Sec. I) have assumed CPU-like MESI coherence, a realistic implementation could face simultaneous coherence requests from tens of thousands of threads; just the buffering requirements would be prohibitive [15].

The only other coherence protocol proposed for GPUs leveraged two observations: (a) that write-through caches provide a natural ordering point at the L2, and (b) that inter-core synchronization can be implicit via a shared on-chip clock [15]. A cache that requests read permissions receives a read-only copy with a limited-time lease; this copy may be read until the shared clock has ticked past the lease time. Two protocols are proposed: TC-STRONG (TCS) can support SC if the core does not reorder accesses, but stalls stores at the L2 to ensure that all leases for the block have expired; TC-WEAK (TCW) allows stores to proceed without stalling, but compromises write atomicity and cannot support SC.

In the next section, we describe Relativistic Cache Coherence, a new GPU coherence protocol that supports SC (like TCS) but allows stores to execute without waiting for write permissions (like TCW). Table I compares RCC with prior protocols proposed for GPUs in the context of SC.

### III. RELATIVISTIC CACHE COHERENCE

Relativistic Cache Coherence leverages the observation by Lamport [20] that consistency need only be maintained in *logical time*. Two threads may see the memory as it was at two different logical times, as long as each *only* observes all writes logically before — and never sees any writes logically after — its own logical “now.” In RCC, cores maintain separate logical times, which become synchronized only when read-write data is shared.

Like all library coherence protocols [15, 23, 24, 35, 36], RCC allows L1 caches to keep private copies of data only for limited-time “leases” granted for each requested block; when a lease expires, the block self-invalidate in L1 without the need for any coherence traffic. Writes to a block must ensure that no valid copies are present in any L1s by ensuring that the write time exceeds the expiration time of all outstanding leases. In RCC, leases are granted and maintained in logical time, so writes can complete instantly by advancing the writing core’s logical clock.

#### A. Logical clocks, versions, and leases

In relativistic coherence, each core maintains, and independently advances, its own logical clock *(now)*. Similarly, each shared cache (L2) block maintains its own logical version *(ver)*, equal to the logical time of the last write to this block.

Since the L2 grants per-block read leases to private L1 caches, it keeps track of when the last lease for a given block will expire *(exp)*. Each L1 cache also keeps track of the *exp* it was given by the L2. Different L1s may have different *exp* for the same block, but none will exceed the latest *exp* in L2. Because L1s are write-through, they do not need to record *ver* for each block.

A unique, global SC ordering of memory accesses is maintained in logical time by applying three rules:

1. **Core C reading cache block B** must advance its logical time *now* to match B’s current version *ver* if $B.ver > C.now$. This ensures that C cannot use B to compute new data values with logical times $< B.ver$, i.e., that C does not observe a value of B “from the future.”

2. **Core C writing cache block B** must advance B’s *ver* to C’s *now* if $B.ver < C.now$, and advance its own *now* to B’s *ver* if $B.ver > C.now$. This ensures the new value of B cannot be used for computation in cores whose *now* is earlier, i.e., that C is not “sent back in time.”

3. **Core C writing cache block B** must advance its *now* as well as the new B.ver beyond the expiration time *exp* of the last outstanding lease for B. This ensures that the new value of B does not “leak” i.e., that any values computed from the new value of B by other cores cannot coexist in their L1s with the old value of B.

The logical *now* times of memory operations provide a sequentially consistent ordering. Provided the core scheduler is modified to ensure that only one global memory access per warp is issued at any given time, RCC supports SC.

#### B. Example walkthrough

Fig. 3 shows how RCC operates on a sequence of instructions from two different cores. Initially, C0’s cache has neither A and B (since *now > exp*) and core C1 has both. In the shared L2 cache, B has since been written by a third core and has *ver = 30*; because C1’s *now* has not advanced past 10, however, it may still read its cached copy of B.

*The proof that RCC supports SC is essentially the same as for Tardis [37], we refer the interested reader there. The main difference is that RCC permits a sequence of unobserved stores to share the same logical version; the SC ordering in that case is provided by the physical arrival times at the L2.*
C. Coherence protocol: states and transitions

The full state transition diagram for RCC, including both stable and transient states, is shown in Fig. 4.

Stable states. RCC has two stable states: V (VALID) and I (INVALID). Blocks loaded into the L1 transition to the V state, and may be read until they are evicted, written, or until their leases expire, at which point they self-invalidate and transition to the I state. Stores (and atomic read-modify-write operations) may occur in both V and I states; the request is forwarded to the L2 (GPU L1s are write-through, write-no-allocate), and the block eventually transitions to I after the store ack is received. Expired blocks in V state (exp < now) are treated exactly the same way as blocks in I state for memory operations and cache replacement purposes.

The L2 also only has V and I states. L2 misses retrieve the value from memory and transition to V. Because the L2 is write-back (like in commercial GPUs), the V state allows reads, writes, and atomic operations; a block transitions to I only when evicted by the L2 cache replacement algorithm.

Transistent states. L1 blocks also have three transient states:

Figure 3. RCC executing accesses to two addresses (A and B) from two cores (C0 and C1). The table (left) tracks each core’s logical time (now), and each cache block’s version (ver) and read lease expiration (exp) after each instruction has executed; the rows represent the order of instructions as executed in physical time. The diagram (right) illustrates the lease durations in each cache (top) and how the logical time now advances in each core as the corresponding operations from the table execute (bottom); logical time flows left to right while physical time flows top to bottom. Bold values denote changes since the last step; crossed-out leases have expired.

First, core C0 writes A, which updates the A.ver in the L2 (rule 2); C1 still has now = 0 and can read its old copy of A. C0 then reads B, which receives a new lease (until logical time 40) but must advance its now past B.ver (rule 1).

Next, C1 writes B, which updates B.ver and C1.now to 41, past the last outstanding lease for B (rule 3). This step enforces SC ordering between the two cores: C1 next reads A, and is forced to pick up the value written by C0.

Finally, C0 writes B, advancing its now past the previous write to B (rule 2), and then A, advancing past the last lease for A (rule 3). Because C1.now is earlier, however, C1’s next load will happen logically before C0’s write to A, and will not observe the new value. Note that SC has been maintained, as the overall behaviour is explained by the following sequential interleaving: C0: ST A, LD B; C1: ST B, LD A, LD A; C0: ST B, ST A.

Figure 4. Full L1 and L2 coherence FSMs (stable and transient states).
hundreds of cycles [38]. To permit non-blocking misses, the L2 coherence controller has two transient states:

IV buffers new GETS and WRITE requests in the relevant MSHR, keeping track of the maximum now times from the reading and writing processors. Once the data arrives from DRAM, the block’s version is updated to reflect any writes in the MSHR and a new lease is generated to satisfy any readers.

IAV indicates an ATOMIC operation received in an invalid state; this stalls any further L1 requests until the block has been retrieved from DRAM, its version has been established, and the atomic operation has completed.

RCC instead allows the eviction but ensures that, if the block is reloaded from DRAM, reading or writing it will cause any outstanding leases for it to expire. To enforce this, we could keep track of ver and exp for each block in DRAM, but this would require additional storage provisions in main memory. Instead, we store the maximum ver or exp of any evicted block as the “memory time” mnow, one in each memory partition. To maintain logical ordering, a block loaded from DRAM will have its ver and exp set to mnow: any cores that read or write this block will have to advance their logical time to prevent the issue described above.

Since the L2 is write-back (like in extant GPUs [16–18]), a WRITE request that misses in L2 will be stored in MSHR while the block is set to IV state and retrieved from DRAM, and any additional write requests are merged into the MSHR. To maintain correct logical write ordering, each MSHR keeps track of lastwr, the highest write time (originating core now value) of any write requests received in IV state. WRITE requests with now ≥ lastwr update the MSHR data and lastwr: write requests with now < lastwr do not change lastwr but must be tracked until the final write time is known. The larger of lastwr and mnow will become the block’s ver; since this is the logical write time, the store can be acknowledged without waiting for the DRAM response. The store data will remain in the MSHR until the DRAM response arrives.

A similar case arises for read requests that miss in L2. MSHRs keep track of lastrd, the latest now of any reading cores; this is used to calculate the lease expiration (exp) once the block is available, and can be elided to save space (lastwr would be used instead).

### Timestamp rollover

Because timestamps have finite exact representations and keep increasing, they are subject to arithmetic rollover. In our experiments, 32-bit logical timestamps advanced on average once for every 1073 core clock cycles; this corresponds to approximately one rollover per hour at clock speeds found in high-performance GPUs.

In principle, this can be handled simply by setting core now clocks to 0, flushing all L1s, setting all L2 ver and exp entries to 0, and setting all mnow values to 0; SRAMs that support flash-clearing [39] make this easy. However, rollover must be processed atomically in the presence of in-flight messages, transient cache states, and independent L2 banks. To implement this correctly, we observe that the L2 is the only coherence actor that actually increases timestamps (L1s only copy timestamps received from L2); therefore, the L2 will be the first component to know that rollover is required.

When an L2 partition needs to roll over a timestamp, it first ensures that all other L2 partitions have stalled and set their timestamps to 0. This can be done in many ways, perhaps using a narrow unidirectional ring with the rollover L2 partition sending a STALL flit and all other cores stalling before allowing the flit to continue; when STALL returns to the originating core, all cores will have stalled (in case of concurrent stall requests, lowest L2 partition ID wins). All

<table>
<thead>
<tr>
<th>name</th>
<th>granularity</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>now</td>
<td>GPU core</td>
<td>logical time seen by this core</td>
</tr>
<tr>
<td>exp</td>
<td>cache block</td>
<td>lease expiration time</td>
</tr>
<tr>
<td>ver</td>
<td>cache block</td>
<td>data version (last write time)</td>
</tr>
<tr>
<td>mnow</td>
<td>mem. partition</td>
<td>max(exp, ver) evicted to DRAM</td>
</tr>
<tr>
<td>lastrd</td>
<td>L2 MSHR</td>
<td>latest now of any reading core</td>
</tr>
<tr>
<td>lastwr</td>
<td>L2 MSHR</td>
<td>latest now of any writing core</td>
</tr>
</tbody>
</table>

Table II

TIMESTAMPS USED IN RCC
stalling partitions must set all of their timestamps (including lastwr and lastrd) to 0; queued requests and MSHR entries are retained, with all timestamps reset to 0. The rollover partition then sends a flush request to all L1s, and waits for responses from all; once these have been received, a resume flit is sent on the inter-partition ring, and all L2 partitions resume processing requests. An L1 that receives a flush request sets its now to 0 and invalidates all entries before replying to L2; addresses with MSHR entries enter the II state, while the remaining addresses transition to I.

E. Lease extension, and prediction

When the L2 receives a gets request, it generates a read lease for the block and sends the logical expiration time exp back to the requesting L1. So far, we have assumed all leases have the same duration (of 10 in Sec. III-B); intuitively, however, read-only data should receive very long leases to avoid expiration, whereas data shared frequently should receive short leases to avoid advancing the logical time too much when they are written (and thus causing other cache blocks to expire).

When a lease is too short, a load request finds the L1 block in V state but with an expired lease (now > exp). Fig. 6 (left) shows how many L1 cache blocks are in V state but expired when accessed. Sometimes, this is the coherence protocol working as intended and indicates a transitive logically-before relation; at other times, the expiration reflects imperfect lease assignment. Fig. 6 (right) shows that most such expirations are premature (i.e., the block’s L2 entry has not changed).

Lease extension. Every such block generates a gets request and a data response from the L2. While the gets is small, a data response includes the full cache block, which poses an unnecessary traffic overhead.

Since the L2 knows when the block was last written (ver), it could potentially renew the lease by sending the new lease expiration time but no data (which the L1 already has). Before deciding whether to send renew or the full data, the L2 needs to know whether the L1’s previous lease is older than ver; if it is, the L1 may have incorrect data. To provide this information, we modify Gets requests to carry the exp time of the expired lease (tracked by the L1): if this is newer than the data version ver in the L2, a renew grant can be sent. The required protocol changes are shaded in Figure 5; note that the complexity cost is minimal, with no additional states and only two new transitions. Prior work [23] also features a lease extension mechanism, but the renew mechanism there relies on keeping track of data versions ver in the L1 caches.

Fig. 7 (left) shows that the renewal mechanism is effective in reducing interconnect traffic for inter-workgroup sharing workloads by 15% (traffic is also reduced for the intra-workgroup benchmarks, but their expiration rates are negligible to begin with).

Lease prediction. Although lease extension reduces interconnect traffic, many expirations would not occur to begin
appears in the proceedings of the 23rd international symposium on high performance computer architecture (HPCA), 2017

<table>
<thead>
<tr>
<th>GPU cores</th>
<th>16 streaming multiprocessors (SMs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>core config</td>
<td>1.4 GHz, 48 warps × 32 threads, 32 lanes</td>
</tr>
<tr>
<td>warp sched.</td>
<td>loose round-robin</td>
</tr>
<tr>
<td>register file</td>
<td>32,768 registers (32-bit)</td>
</tr>
<tr>
<td>scratchpad</td>
<td>48 KB</td>
</tr>
</tbody>
</table>

| per-core L1 | 32 KB, 4-way set-associative, 128-byte lines, 128 MSHRs |
| total L2 | 1024 MB = 8 partitions × 128 KB |
| L2 partition | 128 KB, 8-way set-associative, 128-byte lines, 128 MSHRs; 340-cycle minimum latency [38] |
| interconnect | one xbar/direction, one 32-bit flit/cycle/dir. |
| DRAM | 1400 MHz, GDDR, 8 bytes/cycle (175 GB/s peak); 460-cycle minimum latency, FR-FCFS queues, \( t_{CL}=12 \), \( t_{RP}=12 \), \( t_{RC}=40 \), \( t_{RAS}=28 \), \( t_{CCD}=2 \), \( t_{WL}=4 \), \( t_{RCD}=12 \), \( t_{RRD}=6 \), \( t_{CDL}=5 \), \( t_{WR}=12 \), \( t_{CCDL}=3 \), \( t_{WR}=2 \) |

| lease times | 32 bits, predicted from 8–16–⋯⋯–1024–2048 |

**Table III**

**Simulated GPU and memory hierarchy**

<table>
<thead>
<tr>
<th>inter-threadblock communication</th>
<th>BFS: breadth-first-search</th>
<th>graph traversal [40]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BH</td>
<td>Barnes-Hut n-body simulation kernel [41]</td>
<td></td>
</tr>
<tr>
<td>CL</td>
<td>RopaDemo cloth physics kernel [42]</td>
<td></td>
</tr>
<tr>
<td>DLB</td>
<td>dynamic load balancing algorithm for octree partitioning [43]</td>
<td></td>
</tr>
<tr>
<td>STN</td>
<td>stencil finite difference solver synchronized using fast barriers [44]</td>
<td></td>
</tr>
<tr>
<td>VPR</td>
<td>place &amp; route FPGA synthesis tool [45]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>intra-threadblock communication</th>
<th>HSP: hotspot 2D thermal simulation kernel [46]</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMN</td>
<td>k-means iterative clustering algorithm [46]</td>
</tr>
<tr>
<td>LPS</td>
<td>Laplace solver 3D Laplace Solver [46]</td>
</tr>
<tr>
<td>NDL</td>
<td>Needleman-Wunsch DNA sequence alignment [46]</td>
</tr>
<tr>
<td>SR</td>
<td>anisotropic diffusion speckle reduction for ultrasound images [46]</td>
</tr>
<tr>
<td>LUD</td>
<td>matrix LU matrix LU decomposition [46]</td>
</tr>
</tbody>
</table>

**Table IV**

**Benchmarks used for evaluation.**

with if each block received an optimal lease. We attempted to sweep a range of fixed leases, but found that the performance spread among them was negligible. This is because RCC operates in logical time and most operations advance time in lease-sized amounts; therefore choosing a single fixed lease merely changes the rate at which logical clocks run for everyone. Optimally choosing leases, however, is a non-trivial problem for read-write shared data partly because the “correct” lease depends on the precise scheduling and interleaving of threads; while the correct lease is obvious for read-only data (=∞), detecting read-only data at runtime requires microarchitectural changes [14].

Instead, we observe that GPU applications tend to work in synchronized phases, with most data being read at the beginning of a phase and written at the end. These (and read-only) data should receive fairly long leases, while data that is shared often (e.g., locks) should receive short leases.

To find the best lease, the L2 initially predicts the maximum lease (2048) for every block. When the block is written, the prediction drops to the minimum (8), and grows (2×) every time a read lease is successfully renewed. This way the L2 quickly learns to predict short leases for frequently shared read-write blocks (such as those containing locks), but long leases for data that is mostly read and blocks that miss in the L2 (e.g., streaming reads). A similar per-block lease prediction mechanism has been proposed [24] for logical-time CPU coherence protocols; unlike our predictor, however, short leases are preferred, and the consistency model is relaxed (to TSO) to maintain performance. Fig. 7 (right) shows that the predictor reduces expired reads by 31% for inter-workgroup workloads (again, intra-workgroup benchmarks benefit but start with negligible expiration rates).

**Potential livelock.** Because RCC allows cores to read cached data without advancing their logical clocks, a spinlock that only reads a synchronization variable may livelock unless other warps advance the logical time. This optimization is common in multicore CPUs with invalidate-based coherence, but relies on implicit store-to-load synchronization that is not guaranteed by coherence or consistency requirements. To the best of our knowledge, these kinds of spinlocks are not used in GPUs, as most workloads have enough available parallelism to cover synchronization delays; spinning merely prevents other (potentially more productive) warps from executing (in general, synchronization in GPUs requires different optimizations than in CPUs [44]). Nevertheless, this potential livelock can be avoided by periodically incrementing the logical time now (say, by 1 every 10,000 cycles).

**F. RCC-WO: a weakly ordered variant**

Relative load and store ordering is effected through the per-core logical time now. Keeping track of two separate logical now times — the read view, consulted and updated by load operations, and the write view, consulted and updated by store operations — allows loads and stores to be reordered with respect to each other. In this scheme, full fence operations require only that the read view and write view now values
be set to whichever is larger; performance can potentially improve because stores no longer expire cache data that do not have the same block address. The consistency model is WO [10]; work concurrent with ours [24] proposes a similar adaptation that supports RCsc [11].

IV. Results and discussion

A. Simulation setup

We follow the methodology used in previous GPU coherence work [14, 15]. GPGPUsim 3.x [40] is used to simulate the core, and combined with the Ruby memory hierarchy simulator from gem5 [47] to execute coherence transactions. For the sequentially consistent implementations (MESI, TCS, RCC), we altered the shader core model to execute global memory instructions sequentially, and stall local memory operations if there are outstanding global accesses; this matches the “naïve SC” baseline of [14]. We use Garnet [48] to simulate the NoC and ORION 2.0 [49] to estimate interconnect energy.

The simulated configuration is similar to NVIDIA’s GTX480 (Fermi [16]), with latencies derived from microbenchmark studies [38]; this matches the configurations used in prior work [14, 15]. Table III describes the details.

B. Benchmarks

We use benchmarks identified and classified into inter- and intra-threadblock communication categories in prior work on GPU coherence [15]. The intra-threadblock benchmarks execute correctly without coherence, but are used to quantify the impact of always-on cache coherence on traditional GPU workloads. For non-SC simulations, the inter-threadblock communication benchmarks rely on fences; for SC simulations fences act as no-ops in hardware, but were left in the sources to prevent the compiler from reordering operations.

Benchmark details and sources are listed in Table IV. Most were used in prior work on GPU coherence [15]; we dropped two because our sensitivity studies found them to be highly nondeterministic and unpredictably sensitive to small changes in architectural parameters (e.g., a few cycles’ change in L2 latency). We added missing fences to dlb following [9], and altered tile dimensions in hsp to match GPU cache block sizes and avoid severe false sharing problems.

C. Results

RCC significantly reduces SC overheads compared to prior SC implementations for GPUs. Fig. 8 (top) shows issue stall rates caused by enforcing SC: either direct SC memory ordering stalls or LSU pipeline stalls caused by waiting on store acknowledgements. RCC reduces these by 52% relative to MESI (largely because there are no invalidate delays) and by 25% relative to TCS (largely because stores in RCC acquire write permissions without stalling). Fig. 8 (bottom) shows that SC ordering stalls in RCC are resolved 35% faster than in MESI and 11% faster relative to TCS. Both of these metrics directly correlate to performance (see below).

TCW performs better than RCC for rs because it benefits both from its weak memory model and from relaxing write atomicity. All threads share a “mask” vector, which identifies nodes to be visited in the next iteration (next level of the rs tree); TCW allows different cores to modify parts of this vector without other cores observing the result, while RCC strictly enforces SC on cache block granularity and sees more L1 misses (73% vs. 52%).

Conversely, RCC outperforms TCW on DLB. In DLB, a per-threadblock work scheduler that completes its task steals tasks from a random other threadblock’s scheduler. Since work could be stolen at any time, all per-threadblock queue accesses must be protected with fences; fences stall in TCW until a physical time when all stores have become globally visible. In actuality, however, work stealing events are rare, so most of these stalls are unnecessary. RCC allows cores to progress independently in their own epochs until actual sharing occurs. In addition, stores do not stall even when sharing does occur because SC is enforced in logical time.

SC on top of RCC performs substantially better than prior SC proposals for GPUs. Fig. 9a shows that RCC is 76% faster than MESI and 29% faster than TCS on workloads with inter-workgroup sharing; in fact, performance is within 7% of TCW, the best prior non-SC proposal. On benchmarks with intra-workgroup communication patterns, RCC is 10% better than MESI and within 3% of both TCS and TCW.

Interconnect energy is 45% lower than MESI, 25% lower than TCS, and only 7% below TCW on inter-workgroup workloads (Fig. 9b); on intra-workgroup programs, it is 25% better than MESI and on par with TCS/TCW. This is partly due to reductions in traffic (Fig. 9c) and partly due to RCC needing only two virtual networks to maintain deadlock-free operations vs. five for MESI. Interconnect energy expenditure is becoming more important as GPU core counts grow.

RCC closes the strong–weak ordering gap to 7%. We also developed RCC-WO, a weakly ordered variant of RCC (Sec. III-F) and compared it with both TCW (our implementation supports WO) and the default SC implementation of RCC. RCC-WO performs neck-to-neck with TCW, and both perform 7% better than RCC-SC.

One RCC implementation can support strong and weak consistency. The microarchitectural differences between weak and strong variants of RCC in GPUs consist of one additional scheduler signal per warp to order memops from one thread, and a small change in how stores update L2 metadata. This opens the possibility that the hardware memory model in GPUs could be chosen at boot time (as in, e.g., SPARCv9 [50]) or even at runtime.

RCC has fewer states than TCW, TCS, and especially MESI (Table V). This is important because coherence is notoriously difficult to verify: usually, validation involves very simplified formal models and extensive simulations [51, 52],

Figure 9. Performance normalized to a MESI baseline with write-through L1s: (a) speedup, (b) interconnect energy broken down by component, and (c) interconnect traffic broken down by message type. Left: workloads with inter-workgroup sharing; right: intra-workgroup sharing.

Figure 10. Speedup of weak ordering implementations vs. RCC-SC on inter-workgroup (left) and intra-workgroup (right) workloads.

but bugs survive despite extensive validation efforts [53–56].

RCC has reasonable silicon area overheads. For every L1 block, RCC only stores exp and ver. GPU cache blocks are 128 bytes, with perhaps 3-byte tags; with 32-bit timestamps this is 3% overhead for L1 and 6% area overhead for L2.

V. Related work

GPU memory consistency. Hechtman and Sorin first made the case that the performance impact of SC is likely small in GPUs [13]. Singh et al [14] observed that, while this was true for most workloads, some suffered severe penalties with SC because of read-only and private data; they proposed to classify these accesses at runtime and permit reordering while maintaining SC for read-write shared data. Our approach is orthogonal: we focus on SC stall latency, and improve performance for both read-write and read-only data. Both [13] and [14] used a CPU-like setup with MESI and write-back L1 caches. In GPUs, however, write-through L1s perform better [15]: GPU L1 caches have very little space per thread, so a write-back policy brings infrequently written data into the L1 only to write it back soon afterwards. Commercial GPUs have write-through L1s [16–18]. Our work studies GPU-style write-through L1 caches, and compares against the best prior GPU implementation of weak consistency [15]. Sinclair et al [57] adapted DeNovo [58] to GPUs with DRF-0 and HRF variants, and argued that the benefits of HRF over DRF-0 do not warrant the additional complexity; DeNovo, however, requires software to expose additional details to the coherence hardware, while our proposal requires no software changes. Others have proposed RC for system coherence in CPU-GPU APU systems [13,59].

Strong vs. weak consistency in CPUs. Many quills have been sacrificed to argue that sequential consistency is desirable in CPUs and propose how it could be efficiently
implemented [21, 22, 60–69]. Generally, speculation support or other hardware modifications are required to overcome the overheads of SC. Lin et al [21] and Gope et al [22] also used logical order to enforce SC in a CPU setting. We share the conviction that sequential consistency is preferred, but focus on GPUs, which have different architectural constraints (e.g., no speculation support).

**GPU coherence.** Singh et al [15] proposed a GPU coherence protocol based on physical timestamps, and showed that MESI and write-back caches suffered NoC traffic and performance penalties in GPUs. While the consistency model is weak throughout, the base version (TCS) can support SC if the core does not permit multiple outstanding memory operations from one warp; we use this SC variant as a baseline. The improved version (TCW) cannot support SC, but adds offers 30% better performance; we use this for comparison. RCC uses logical rather than physical timestamps, has lower complexity, and closes the SC-to-weak gap between TCS and TCW.

**Library cache coherence.** Nandy and Narayan [70] first observed that timestamps can reduce interconnect traffic due to invalidate messages in MSI-like protocols, but their protocol did not support SC. Shim et al [35] proposed LCC, a sequentially consistent library protocol, for multicore; LCC is equivalent to our TCS baseline. Singh et al [15] adapted LCC to GPUs and proposed a higher-performance weakly ordered variant with a novel fence completion mechanism; Kumar et al [36] used TCW for FPGA accelerators. Recently, Yao et al [71] adapted TCW to multicore by tracking writes with a Bloom filter. All of these protocols use physical timestamps, and SC variants must stall stores (and weak variants must stall fences) until completion; RCC uses logical time and stalls neither stores nor fences.

Lamport [20] first observed that consistency need only be maintained in logical time. This fact has been used to implement coherency on a logically ordered bus (e.g., [72, 73]) and to extend snooping coherency protocols to non-bus interconnects [74, 75]. Meixner and Sorin used logical timestamps to dynamically verify consistency models [31]. Yu et al [23] proposed using logical timestamps to directly implement coherence in CPU-style multicore, but maintains exclusive write states and recall/downgrade messages that we wish to avoid to reduce store latencies. At the same time, architectural features not present on GPUs (e.g., speculative execution) are required to support a timestamp speculation scheme. Work concurrent with ours [24] proposes non-SC variants. RCC shares the notion of keeping coherence with logical timestamps, but eschews exclusive states to focus on reducing store latencies. RCC is a simpler protocol that offers best-in-class performance in GPUs.

**VI. Conclusion**

In this paper we track the source of SC inefficiency in GPUs to long store latencies caused by coherence traffic; these severely exacerbate SC ordering and structural bottlenecks that GPUs could otherwise easily amortize. We address these by proposing RCC, a coherence protocol that uses logical timestamps to reduce store latency. When used as part of an SC implementation, RCC reduces SC-related stalls by 25%, and stall resolve latency by 11%, compared to the best coherence proposal for GPUs capable of supporting SC; as a result, performance is 29% better.

When used in RC mode, RCC matches the best prior RC proposal; because the hardware needed for RCC is similar for SC and RC, a single implementation can potentially allow runtime selection of the desired memory consistency model.

**VII. Acknowledgements**

We are grateful to Tor Aamodt and Ahmed ElTantawy, as well as the to anonymous reviewers, for insightful feedback. This research was funded in part by the Natural Sciences and Engineering Research Council of Canada.

**References**
