Turn-based Spatiotemporal Coherence for GPUs

SOORAJ PUTHOOR, University of Wisconsin-Madison, AMD Research
MIKKO H. LIPASTI, University of Wisconsin-Madison

33

This article introduces turn-based spatiotemporal coherence. Spatiotemporal coherence is a novel coherence implementation that assigns write permission to epochs (or turns) as opposed to a processor core. This paradigm shift in the assignment of write permissions satisfies all conditions of a coherence protocol with virtually no coherence overhead. We discuss the implementation of this coherence mechanism on a baseline GPU. The evaluation shows that spatiotemporal coherence achieves a speedup of 7.13% for workloads with read data reuse across kernels compared to the baseline software-managed GPU coherence implementation while also providing write atomicity and avoiding the need for software inserted acquire-release operations.[1]

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; Multicore architectures;

Additional Key Words and Phrases: Cache coherency, GPU

ACM Reference format:
Sooraj Puthoor and Mikko H. Lipasti. 2023. Turn-based Spatiotemporal Coherence for GPUs. *ACM Trans. Arch. Code Optim.* 20, 3, Article 33 (July 2023), 27 pages.
https://doi.org/10.1145/3593054

1 INTRODUCTION

A cache coherence protocol keeps the private caches coherent in a multi-processor system. Traditional coherence protocols rely on invalidation messages to prevent the processors from reading stale data [52]. In this article, we make the observation that these invalidation messages are not essential for write propagation. If all the processors can agree upon an ordering for address modifications, then the writes can be propagated without invalidation messages. Leveraging this observation, we propose a new cache coherence protocol called spatiotemporal coherence that satisfies all conditions of a coherence protocol but without any invalidation traffic.

Spatiotemporal coherence grants write permissions to groups of addresses called address bands. Each band is granted write permission in a specific interval of time called an epoch. A band gets its write permission in an epoch that is mutually agreed upon by all the processors. Since all

[1]New article, not an extension of a conference paper.

This work was supported in part by NSF Award No. CCF-2010830 and AFRL Award No. FA9550-18-1-0166.
Authors' addresses: S. Puthoor, B100.2.407, 7171 Southwest Pkwy, Austin, TX 78735; email: Sooraj.Puthoor@amd.com; M. H. Lipasti, Room 3621, Engineering Hall, 1415 Engineering Drive, Madison, WI 53706; email: mikko@engr.wisc.edu.

1544-3566/2023/07-ART33 $15.00
https://doi.org/10.1145/3593054

processors have mutually agreed on the address modification epochs, the processors know the modified addresses at any given time without explicit coherence messages.

Spatiotemporal coherence is characteristically different from traditional coherence protocols with regard to granting write permissions. Traditional coherence protocols grant write permissions on an address to a modifying processor but spatiotemporal coherence grants write permissions to an epoch. Moreover, the write permissions are granted within an address band. Thus, spatiotemporal coherence fundamentally changes the processor-centric write permission to an epoch- and band-centric write permission. An epoch has temporal properties and an address band has spatial properties, hence the name **spatiotemporal coherence (STC)**.

Timestamp-based coherence protocols also avoid invalidation messages by getting a time lease on an address and self-invalidating that address after the time lease has expired [42, 51, 58]. However, even these coherence protocols provide exclusive modification permission to a processor by giving it an exclusive time lease for an address. Thus, although the mechanism for recalling write permissions is different (expired timestamps versus invalidation messages), the underlying idea of having an exclusive modifying processor per address is still preserved. STC does not grant modification rights to a processor and hence differs from timestamp-based coherence.

Removing coherence traffic has profound implications in a massively many-core system like a **graphics processor (GPU)**. Current GPUs avoid strict hardware coherence implementations because of their coherence traffic overhead [37, 42, 51]. Instead, GPUs today implement caches that maintain coherence at **release consistency (RC)** synchronization markers inserted by the software [15, 22, 27, 31]. Naturally, they adhere to the release consistency memory model. Here, we refer to such cache coherence implementations as software-managed cache coherence. Software-managed cache coherence does not satisfy all conditions of a coherence protocol [1]. For example, write atomicity is not guaranteed by this mechanism. While the cache design becomes simple and coherence traffic is eliminated with software-managed cache coherence, programmers will have to deal with the non-intuitive nature of weak models, complicating the programming of such systems and increasing the likelihood of software bugs.

STC retains all the hardware advantages of a software-managed cache coherence implementation while also satisfying the properties of a strict hardware cache coherence protocol [1]. In this article, our objective is to introduce the basic concepts of STC, provide an implementation of STC, and demonstrate that even a simple implementation of STC achieves performance comparable to the software-managed baseline cache coherence. Since write atomicity [1]—which is a precondition for implementing strong models—is also provided, STC could support strong memory models as well, though we do not explore this opportunity here. As compared to the baseline, STC can perform well for workloads that have read data reuse across kernel launches. Software-managed coherence invalidates the private caches before a kernel launch (acquire operation) and each kernel in these types of workloads will incur cold misses accessing the read data set. Figure 1 compares the private L1 cache hit rate of a few workloads with acquire operation against a version without acquire operation. Suppressing the acquire operation nearly doubled the hit rate for most of these benchmarks. STC private caches do not cache stale data and hence acquire operation is not needed to maintain coherence (details in later sections). Consequently, these types of workloads benefit from STC. The major contributions of this article are:

- We introduce STC for GPUs.
- We provide an implementation of STC that adheres to the RC memory model. This implementation grants write permission to the epochs in turns and retains all advantages of the software-managed cache coherence.
- We show that STC improves performance by 7.13% for workloads with read data reuse across kernels, and provides performance comparable to the software-managed baseline coherence

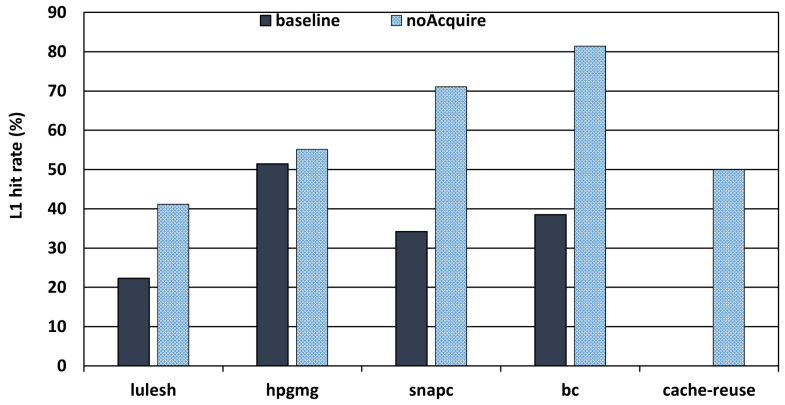

Fig. 1. Impact of acquire operation.

implementation for the rest of the evaluated workloads. STC also performs better than prior work. STC achieves this without software assistance while providing write atomicity.

2 SPATIOTEMPORAL COHERENCE

This section introduces spatiotemporal coherence. STC is implemented with address bands and epochs. We start this section by defining both.

2.1 Address Bands and Epochs

In STC, an address band is simply a group of addresses. All addresses in an address band, hereafter referred simply as a band, share the same write coherence permission. An epoch is defined as a time window in which a band is given write permission. Thus, each band will have at least one epoch associated with it. For example, band_A will have at least an epoch_A when stores to addresses belonging to band_A are allowed to be issued to the memory hierarchy.

An epoch differs from the **modified (M)** state of a coherence protocol although both grant write permission [52]. An epoch grants write permission *to the band associated with it* but M state grants write permission *of an address to a processor*. STC shifts the processor-centric write permissions in traditional protocols to epoch-centric write permissions.

2.2 STC Rules

STC works with four simple **epoch rules (ER)**:

(1) ER1: Addresses in band_A cannot be cached in epoch_A.
(2) ER2: Addresses in band_A can be written only in epoch_A.
(3) ER3: At any (logical) time, all processors should be in the same epoch (or epochs).
(4) ER4: Addresses in a band can be read in any epoch.

ER1 ensures that a potentially modifiable cache block is not cached in a private cache and hence no processor will see stale data as long as all processors are observing the same epoch. Private caches can still cache data from bands when the processors are not in that band's epoch. ER2 ensures that writes are visible to the memory side cache or memory. ER3 makes sure all processors observe the same epoch (or epochs) and there is no conflict among the processors about the current epoch (or epochs). ER4 ensures that reads are never blocked.

We will illustrate these rules with the help of an example. Figure 2 shows the epoch transition from epoch_A to epoch_B for two private caches PC1 and PC2. That figure also shows the loads and

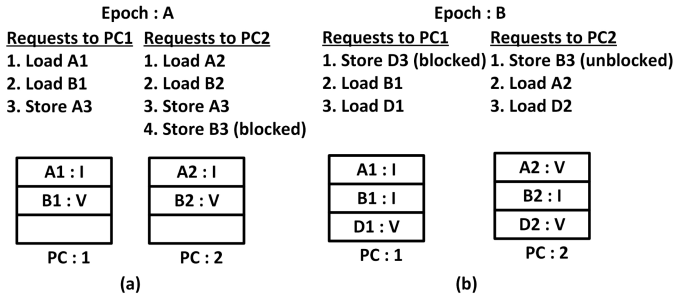

Fig. 2. STC epoch transition from epoch$_A$ to epoch$_B$.

stores arriving at these caches in that epoch. Figure 2(a) shows both PC1 and PC2 after encountering the memory request sequence in epoch$_A$. The loads to band$_A$ (A1, A2) are issued to memory but they are not cached adhering to ER1. However, loads to band$_B$ (B1, B2) are cached in the private caches and the addresses B1 and B2 are in valid (V) state. The store to address A3 belonging to band$_A$ is issued to the memory but the store to band$_B$ is blocked by ER2. Figure 2(b) shows the epoch transition to epoch$_B$. The band$_B$ addresses (B1, B2) are invalidated and the store to B3 from epoch$_A$ is now unblocked and issued to the memory. The loads to epochs other than epoch$_B$ can now cache data in private caches.

2.3 STC Satisfies Cache Coherence

The two conditions associated with cache coherence protocols are: (a) a write is eventually made visible to all processors, and (b) the writes to the same address should be visible to all processors in the same order (write atomicity) [1]. STC's epoch rules are sufficient to satisfy these two conditions.

Writes are eventually made visible to all processors: ER1 allows a band to cache its data only when the processors are not in that band's epoch. However, ER2 allows writes to a band only in that band's epoch. ER3 ensures that all processors agree on the current epoch. Thus, these three STC rules ER1, ER2, and ER3 ensure that no potentially modifiable data is cached in the private caches, and all processors agree which addresses are going to be modified at any given time. ER1 does not allow a store to be cached in a private cache. Consequently, every store updates the memory side cache (or memory) and every load reads the most up-to-date data from that shared cache. Thus, STC satisfies the first condition of making a write eventually visible to all processors.

Write atomicity: Write atomicity comes naturally with STC. The very essence of the STC rules is that no modifiable data is cached in the private caches. Thus, every single write to a band and consequently every address in a band is visible to all processors in the system.

In addition to the above-mentioned conditions, a third condition is often associated with cache coherence protocols: a coherence protocol should inform the writing processor about the completion of a store [1, 51]. Satisfying this condition is required to maintain program order between the write instruction and younger instructions. Toward this, STC sends an acknowledgment back to the writing processor when a store is completed. However, unlike traditional protocols that have to wait for the invalidation of private copies before sending this acknowledgment back, STC can immediately respond with the acknowledgment when the store reaches the memory side cache (or memory). Maintaining program order between the write instructions is a requirement for strong models such as **sequential consistency (SC)** and **total store order (TSO)**, and not a requirement for a weak model such as release consistency [1]. Thus, even though STC sends the write completion ack, a strong model implementation will use it for enforcing ordering but a relaxed model implementation does not need to do it.

3 IMPLICATIONS OF STC

STC allows processors to take localized decisions based on the current epoch and based on the information about that epoch's band. However, for STC to function correctly, all processors should be in the same epoch. Irrespective of the mechanism employed to synchronize the current epoch across all processors, this synchronizing mechanism can be stateless. Since STC does not rely on invalidation messages for write propagation, there is no need for a centralized structure to track the sharers of an address. Additionally, since processors do not own write permissions in STC, tracking global cacheline permissions is not necessary either. Contrast this to a traditional coherence directory that stores the sharer list and global cacheline states for maintaining coherence or a snooping protocol that relies on broadcast snoops and responses for maintaining coherence [52]. Thus, STC eliminates the need for broadcasts or structures such as a global directory. In directory-based coherence protocols, even a null directory (stateless directory) has to broadcast invalidations on receiving a write request, whereas STC does not need invalidation messages for maintaining coherence.

3.1 STC and Write-through Caches

In a traditional protocol with write-through caches, a write request can be issued directly to the memory without that processor acquiring the write permission. However, the write can complete only after the sharers are invalidated by invalidation messages. STC does not require these invalidation messages and the memory-side cache (or memory) can send the write completion ack immediately after seeing the write request. Thus, STC is not a mere optimization of the traditional protocol for write-through caches.

STC and GPU Caches: STC with write-through private cache has profound implications. Typically, GPU caches are write-through with a simple VI (valid/invalid) protocol for coherence [5, 6, 25, 37, 51]. Previous studies have suggested that a write-through L1 cache performs much better than a write-back L1 for GPUs because of the streaming data access pattern of a GPU [51]. Even commercial GPUs employ write-through private caches [6, 15, 31, 35].

However, implementing strict hardware coherence on a GPU is challenging because of the overwhelming number of the invalidation, eviction and recall messages from traditional directory-based coherence protocols [37, 51]. Because of this, GPUs today adhere to RC model, which makes the L1 caches coherent only at synchronization points with the help of software inserted synchronization primitives [5, 6, 25, 37].

The STC eliminates all the above-mentioned challenges of hardware cache coherence in a GPU. STC does not rely on invalidation messages for making the caches coherent. Since invalidations are not needed, sharers need not be tracked, and since the need for sharer tracking is eliminated, the private L1 caches can now perform silent evictions. Thus, STC implements hardware coherence without invalidation traffic, eviction notifications, and recall traffic. Since STC is a natural fit for GPUs, we will implement and evaluate STC on them.

3.2 Write Atomicity

In traditional protocols, write atomicity is implemented with invalidation or update messages that propagate a new write to all processors in the system. This approach is not easy to scale and can incur a severe performance penalty in a many-core system like a GPU [37, 51]. However, STC does not have this limitation. Since the epoch rules ensure that the processors have agreed on the band to be modified in each epoch, and since the epochs themselves are synchronized, the processors do not cache a potentially modifiable address in the cache. Thus, the processors need not be notified about a write but they fetch the updated value from a memory-side cache (or memory). Thus, the STC implementation provides write atomicity without any additional coherence traffic.

3.3 Deadlocks

Epoch rule ER2 states that a band can write only in its epoch. Thus, a write reaching the cache controller on a different epoch is blocked till its epoch arrives. Consequently, a processor may deadlock if an epoch for a write never arrives. Thus, to prevent deadlocks and to ensure forward progress for all writes, an STC implementation should adhere to the following **deadlock avoidance rules (DARs)**: *DAR1: There must be at least one epoch for every band. DAR2: All epochs must eventually terminate. DAR3: All epochs must be granted repeatedly.*

DAR1 ensures that all bands have at least one epoch, DAR2 ensures that an epoch will eventually end and DAR3 ensures that every band's epoch will be eventually granted. Thus, they ensure a write will eventually complete.

3.4 Performance Pitfalls and Optimizations

To eliminate the possibility of reading stale data, a band is not allowed to cache its data during its epoch. Thus, if a processor enters an epoch in which no writes are issued to the corresponding band, then the band unnecessarily lost its opportunity to cache data, adversely impacting the performance of the processor. Another potential performance pitfall with STC is a band with read-only addresses. Since epochs are only needed to grant write permissions to a band, a read-only band does not need an epoch. Hence, granting an epoch for a read-only band is unnecessary, and since that band is non-cacheable in that epoch, it is undesirable as well from a performance standpoint. To solve these performance pitfalls, we introduce two optimizations (a) epoch skipping and (b) adaptive bands.

Epoch skipping: The epoch skipping optimization grants an epoch only if there is a demand for it. A demand to an epoch indicates that at least one write is pending to that epoch from a processor. With epoch skipping, the epochs that did not have any requests to them are skipped. Thus, epoch skipping avoids the performance penalty of granting epochs to a band with no demand. Since a read-only band will not issue any write-request, epoch skipping will avoid granting permissions to read-only epochs as well.

Epoch skipping does not violate rules DAR2 and DAR3. It can be treated as an optimization that terminates the current epoch adhering to DAR2 and grants zero cycles for all epochs with no write demands, adhering to DAR3.

Adaptive bands: The effectiveness of epoch skipping is amplified if the bands are clearly separated into read-only bands and non-read-only bands (write and read-write bands). The adaptive band optimization dynamically changes the addresses in a band to regroup read-only addresses together. Coupled with the epoch-skipping implementation, this dynamic identification of read-only datasets enables sustained caching of blocks across otherwise unnecessary write epochs, since there are no pending writes to such read-only bands.

4 IMPLEMENTATION

We implement STC on a GPU with per **compute unit (CU)** write-through private L1 caches and a shared L2 cache as shown in Figure 3. The GPU adheres to the RC model and implements a **valid/invalid (VI)** protocol for making L1s coherent at RC synchronization points [5, 6, 25, 30, 35, 37, 54].

4.1 RC Implementation on Baseline GPU

Our baseline GPU executes GCN3 ISA [7] and implements RC with two instructions: (a) *s_waitcnt* and (b) *buffer_wbinvl1_vol* [7, 10]. The *s_waitcnt vmcnt(0)* stalls the wavefront till all pending memory operations are completed and hence used for ensuring the ordering requirements of

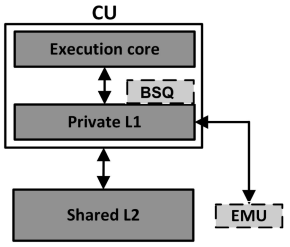

Fig. 3. STC baseline with EMU.

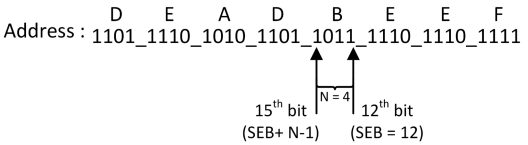

Fig. 4. Epoch and band implementation.

acquire/release operations in the GCN3-based GPUs. To track the pending stores, the store completion acknowledgments are forwarded to the execution pipeline. The pending loads are simply tracked by load data returns. The *buffer_wbinvl1_vol* instruction performs the L1 cache invalidation part of the acquire operation. The baseline flash invalidates all valid L1 entries in one cycle by flipping the valid bits. An acquire-release operation is implemented by a combination of these two instructions. A more detailed discussion of the acquire-release operations with GCN3 ISA can be found elsewhere [7, 10].

Our STC implementation adheres to the RC model. The release operation is implemented similar to the baseline GPU with the write completion acknowledgments forwarded to the GPU pipeline and the *s_waitcnt vmcnt(0)* instruction stalling the wavefront till the write completion. However, the *buffer_wbinvl1_vol* instruction is treated as a no-op, since STC private caches do not cache stale data.

4.2 STC Baseline

STC relies on two concepts: epochs that subdivide time and bands that subdivide the address space. The potential design space for these is large, as STC will operate correctly for very flexible definitions of each, as long as the epoch rules are satisfied. In this section, as a starting point, we define both epochs and bands in the simplest possible way: as fixed-duration windows of time and as contiguous regions of the address space. More complex and flexible definitions of epochs and bands may provide additional advantages, but, as we will show, even the simplest definitions presented here enable the correct operation with high performance.

In our implementation, we use N contiguous bits of an address to identify a band, with matching address bits in that set of N bits forming a band. We then associate an epoch in time for every such band. Thus, our implementation will have 2^N total epochs. The ***start epoch bits* (SEB)** identify the position from which these N contiguous bits should be extracted to form a band. This simple implementation guarantees that every band has one epoch conforming to the deadlock avoidance rule DAR1. Figure 4 shows the epoch and band with N = 4 and SEB = 12 for an address 0xDEAD_BEEF. Since the four bits from 12th bit is 0xB (11 in decimal), this address corresponds to epoch_{11} and the set of addresses with bits[15:12] = 0xB forms band_{11}. Thus, this simple band implementation

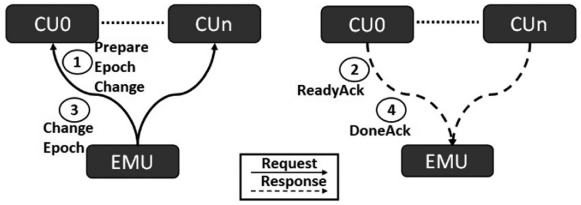

Fig. 5. Epoch transition and synchronization.

allows individual CUs to identify the band of a request from only its address for a given N and SEB.

A CU tracks the current epoch with an N-bit **current epoch register (CER)**. For implementing the epoch rules, a CU compares the band of a memory request to the current epoch stored in CER. Only stores with bands matching the current epoch are issued to the cache and the remaining stores are blocked. All blocked stores are moved to a **blocked store queue (BSQ)** and they are reanalyzed on an epoch change. The BSQ is banked so that multiple stores can be analyzed in a cycle. The store data is kept in the coalescing buffers [33, 39] and only addresses of the blocked stores are moved to BSQ. If this coalescing buffer (or BSQ) gets full, then the backpressure mechanism will stall the GPU pipeline, thus preventing the overflow of these buffers. The coalescing unit can only coalesce requests from the same wavefront instruction [39]. It also blocks requests to an address when there are older outstanding requests to that address. Thus, since the coalescing unit blocks loads under a pending write to the same address from being issued, the younger loads need not access the BSQ for reading the last store. The loads belonging to the current epoch's band are not allowed to cache and hence are issued as non-cacheable loads.

To synchronize epochs across all the CUs, we implemented a **epoch management unit (EMU)**, which plays a conceptually simple role in STC: ensuring that all CUs transition across epochs correctly at well-defined epoch boundaries. The epochs are defined as a fixed duration of time in this implementation. More complex implementations can have a dynamic epoch duration, but, as we will show, even this fixed duration epoch provided good performance for the benchmarks we evaluated. The EMU periodically wakes up at this fixed time intervals and changes the epoch in a round-robin fashion for all CUs, thus ensuring writes to all bands corresponding to these epochs are completed eventually. Figure 3 shows the STC baseline after adding the EMU and BSQ. For synchronizing the new epoch with all the CUs, the EMU uses a four-way handshake. This four-way handshake involves two epoch requests and the acks to these requests. Figure 5 illustrates this operation in detail. The EMU wakes up and broadcasts a *PrepareEpochChange* request to all the CUs①. Each CU then stops issuing stores, waits for the outstanding writes to complete, and responds back with the *ReadyAck* notifying the EMU that they are ready for an epoch change ②. The EMU on receiving ReadyAcks from all CUs broadcasts the *ChangeEpoch* request ③. The ChangeEpoch request carries the new epoch value and the CUs on receiving this ChangeEpoch request transition to the new epoch after responding with a *DoneAck* ④. The epoch transitioning ends after the EMU receives DoneAck from all CUs.

On receiving a PrepareEpochChange message, the CUs block all stores and wait for all outstanding stores to complete. Moving to a new epoch with pending stores will result in those stores modifying a band that does not belong to the current epoch violating ER2. When all outstanding stores are complete, the CU responds back to the EMU with a ReadyAck. Thus, ReadyAck is a guarantee given by that CU to the EMU that no further modifications to any band will be carried out by that CU. Once EMU receives ReadyAcks from all CUs, EMU broadcasts the new epoch value to all the CUs with a ChangeEpoch message.

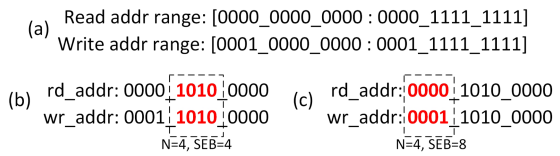

Fig. 6. Need for adaptive bands.

On receiving the ChangeEpoch message, the CUs update their CER with the new epoch from that message. The blocked stores in the BSQ are now reanalyzed and the stores to the new epoch's band are removed from BSQ and issued to the caches. An epoch transition also requires valid entries that belong to the current epoch's band to be invalidated. A lazy invalidation mechanism is implemented that delays these invalidations till the first access to a cache set. Whenever a demand request reaches a cache set, the tags are read, and they are compared against the tag of the demand request. We leverage this normal cache operation to additionally compare the bands of the addresses stored in that cache set and if the bands match the current epoch, that entry is invalidated (only needs to be done once per cache set per epoch transition). However, in rare situations, it may happen that not all the cache sets are accessed in an entire epoch. So, during the end of every epoch, the cache sets that are still untouched during the entire epoch are read and the cached addresses with bands corresponding to the current epoch are invalidated. After these actions, the CU moves to the new epoch by responding with DoneAck.

4.3 Epoch Skipping Implementation

The baseline STC implementation imposes epochs for all bands in the address space, regardless of whether there are any pending writes to those bands. Epoch skipping considers this case and elides epochs without any pending writes. As described, STC blocks the writes not belonging to the current epoch. With the epoch skipping optimization, the blocked writes send an *EpochDemand* request to the EMU. The EMU keeps track of the epoch demands in an **epoch request vector (ERV)**. The ERV is a bit vector with one bit per epoch. On receiving the epoch demand request, the entry corresponding to the requested epoch is set in the ERV. When the EMU wakes up, it increments the epoch counter till it finds an epoch with an outstanding demand in the ERV. This epoch then becomes the new epoch and EMU sends the PrepareEpochChange to initiate the transition to this new epoch after resetting the ERV entry. If no entry is set in the ERV, then EMU does not change the current epoch and goes back to sleep again.

On receiving an EpochDemand request, the EMU responds back with an EpochDemandAck notifying the requesting CU that its request is acknowledged. The CU issues only one EpochDemand per band, thus avoiding duplicate EpochDemand requests to the same epoch for that band. This EpochDemand request filtering is implemented with an epoch demand bit vector. Each CU has its own epoch demand vector and this vector has one entry for every epoch. A blocked store issues an EpochDemand request only when the corresponding entry in the epoch demand vector is empty. An entry is set after the first EpochDemand request thus preventing duplicate EpochDemand requests to the same epoch from a CU. The EpochDemand request vector is reset during the beginning of an epoch. This epoch filtering mechanism thus keep an upper bound on the number of outstanding EpochDemand requests from a CU and filters out duplicate requests, significantly reducing the number of such requests.

4.4 Adaptive Band Implementation

The adaptive band mechanism attempts to regroup addresses in a band in a way that avoids comingling of read-only addresses with read-write addresses using a simple dynamic search heuristic. Figure 6(a) shows an example where the first 256 addresses are allocated for read datasets and

the next 256 addresses are allocated for write datasets. These types of contiguous allocations are common for GPU data structures with programmers optimizing the data layout for better reference coalescing. Figure 6(b) shows an example where both these read and write data sets share the same band with N = 4 and SEB = 4. With SEB = 4, both the read_addr and write_addr are in $band_{10}$ and this read_addr in $epoch_{10}$ will be issued as a non-cacheable read (adhering to epoch rules) because of this sub-optimal band formation. We call this $epoch_{10}$ a conflicted epoch and its occurrence an epoch conflict. However, if the SEB is increased to 8, the read and write data addresses fall into different bands, that is to $band_0$ and $band_1$, respectively, as shown in Figure 6(c). With epoch skipping, an EpochDemand request to $band_0$ will not be issued and consequently, $band_0$ will never become the current epoch enabling read datasets to be cached throughout the execution.

Our implementation of adaptive bands dynamically increases or decreases the SEB to reduce epoch conflicts. The epoch conflict detection requires both the conflicting read and write addresses for calculating the adaptive SEB. Toward this, we modified the EpochDemand request to send the entire write address and added an EpochConflict message to send the conflicting read address to EMU. EpochConflict message is generated when a CU has a blocked write to a band and then a subsequent read to the same band happens. To store the write address, each entry of the ERV is enhanced with a 32-bit address field. When an EpochConflict message arrives, the read address from that message is compared against the stored write address to identify the optimal SEB. The decision to shift up/down is based on whether the bits lower to the band position bits are different or not. If the lower bits are different, then band position bits must be moved to lower bits so that the reads and writes will be in different bands, and vice versa. In our implementation, each adaptive band formation increases or decreases the SEB by only one bit. So, to move from a band formation shown in Figures 6(b) to 6(c), our implementation will require four adaptive band transitions. But an aggressive implementation may shift multiple bits at a time and can reach the optimal band positioning much faster. Additionally, to reduce the number of EpochConflict messages from a CU, only the first conflicting read access generates an EpochConflict message. The EMU informs the CUs about the updated SEB by sending it along with the ChangeEpoch request. With the adaptive band formation, there may be overlap between the new bands and old bands. So, if there are pending memory operations during adaptive band transition, the memory operations may return in a different band than when they were issued. However, the four-way handshaking mechanism ensures that the CUs have no pending stores during epoch transition, thus avoiding such a situation for a store. The load may return in a different band, but the load data is not installed in the cache if it belongs to the current epoch (the current epoch check is done based on the new band definitions). Thus, overlap of bands during adaptive band transition does not violate STC rules.

Our adaptive band implementation only changes the start bit used for the address to band mapping. Thus, this new mapping ensures that all the addresses have a corresponding band and an epoch. The EMU, like before, round-robins through all the epochs thus ensuring writes to any address will eventually complete.

MultiBands: Once adaptive band optimization filters out the read-only bands, epochs are granted only to write or read-write bands. Multi-band optimization grants epochs corresponding to multiple bands at the same time so that stores belonging to these bands can be issued without moving to the BSQ. Thus, with this optimization, stores are completed faster. To implement this multi-band optimization, CER is enhanced to store multiple concurrent epochs. Also, the comparators that determine if a load/store is in the current epoch are enhanced to check for multiple epochs. The lazy invalidation mechanism discussed earlier is also augmented to check for multiple concurrent epochs. Other than the enhancements to these comparators and the CER, the multi-band implementation works similar to the adaptive band implementation. The additional epochs to be granted are determined similarly to the adaptive band optimization. However, instead of just

Table 1. State Transition Table

	Store	Load	Load_U	Data	Data_E	WA
I	I / PW++	I_V	I_I			PW−−
V	I / PW++	Hit	I_I			PW−−
I_V				V / LD	I / LD	PW−−
I_I				I / LD	I / LD	PW−−

selecting one epoch with pending epoch demand in a round-robin fashion, its adjacent epochs with pending epoch demands are also selected, and these epochs are granted together with the multi-band optimization.

Increasing the band range by combining bands together is an alternate approach that may achieve the same benefits as multi-band optimization. However, increasing the band range may also result in combining multiple dissimilar bands together, adversely impacting STC's performance. For example, suppose there are three adjacent independent bands A (0000_**1010**), B (0000_**1011**), and C (0000_**1100**). Suppose B is a read-only band whereas A and C are either write-only or read-write bands. Now, based on our simple N contiguous bits-based band mapping, combining A and C will require us to move the start bit for address to band mapping to bit position 3 and this newly formed band (0**000_1**XXX) will also include band B. That means increasing the band range (or combining bands) may group multiple read-only and read-write bands together, and the newly formed read-write band cannot provide caching benefits to the read-only addresses in that band. In the above scenario, a multiband optimization can simultaneously grant epochs to bands A and C, and still preserve the read-only band B retaining the caching benefits.

4.5 STC and Cache States

The STC cache controller state transition table is given in Table 1. The epoch messages do not interfere with the cache states and hence are omitted from the table. In that table, *pending writes*, *load done*, and *write completion ack* are abbreviated as PW, LD, and WA, respectively. I is the invalid state, V is the valid state, and I_V is the transient state when the cache is waiting on a load return to move to V from I. This transition table is similar to a GPU VI protocol except for the two events Load_U and Data_E, and one transient state I_I. A Load_U is a non-cacheable load issued when the current epoch matches the load's band and hence does not allow installation of the data in the cache. I_I is the transient state that tracks this pending Load_U and moves to I when the load returns, making it non-cacheable. The Data_E is the Data response to a load whose band matches the current epoch and hence the data from that epoch is also not installed in the cache even if that load was issued as a cacheable load (Load) in one of the previous epochs (I_I to I).

Atomic operations are resolved in the shared L2 in GPUs [10]. In STC, atomics are treated as regular writes. Atomics that return a value generate a Data_E response along with a write completion ack (WA) wheres atomics with no return value only generate a WA. Hence, atomics do not have additional states in the cache transition table.

Protocol Complexity: STC adds one transient state I_I to the RC managed baseline. However, an invalidation-based hardware GPU VI protocol will add 4 to 9 transient states to the baseline [51] depending on the optimizations involved. Transient states increase the protocol verification complexity [21, 55]. Additionally, race transitions [55] that further add to the verification complexity are also absent in STC. Because of these reasons, STC is easier to verify than any invalidation-based coherence protocol. STC is a rule-based system and adhering to the rules provides correctness. Hence, the correctness invariant is to check against the STC rules, which is straightforward.

4.6 Hardware Additions

The EMU has an epoch counter and a wakeup counter. The epoch skipping and adaptive band optimization add ERV and ETV to the EMU. The ETV is a 1-bit vector with one entry per epoch. ERV has 33-bits (32 bits for address, 1-bit to store epoch demand to that address) per entry and has one entry per epoch. The CU is augmented with a 32-bits per entry BSQ. All these structures can be implemented with logic cells and does not require SRAMs.

4.7 Working Example

We explain our implementation enforcing RC with the code snippet in Figures 7(a1) and 7(a2). Figure 7(a1) is the CPU implementation of the code and Figure 7(a2) is the GPU implementation of the same code with the GPU wavefront performing all three memory operations with one wavefront instruction. We will explain the implementation using CPU terminology but the working is exactly similar for the GPU as well. Addresses U, V, W and X belong to bands with the epochs U, W, W, and X, respectively. Figure 7(b) shows the BSQ of thread 0 (BSQ 0) and cache of thread 1 (Cache 1) after these threads executed the first three stores in some epoch Z. The transition from this epoch Z to epochs W->V->U->Z->X->Z are shown in Figures 7(c)–7(h). Since thread 0 is only doing stores, nothing is cached in thread 0's cache and hence not shown in these figure. Similarly, thread 1 is only doing loads and hence its empty BSQ is not shown either. Since U, V and W do not have write permissions in epoch Z (Figure 7(b)), these stores are moved to thread0's BSQ. Also, the loads to U, V, and W by thread 1 install these lines in the cache with the initial value 0, whereas the load acquire (LdAcq, line:4) installs X = 0 in to the cache. Since X != 1, the load acquire condition is not satisfied and the load acquire will retry. The acquire/release ordering is implemented with the s_waitcnt instruction inserted by the finalizer [10] (Section 4.1). This s_waitcnt stalls the pipeline till all the older memory operations are completed. Hence the StRel of thread 0 (line:4) is blocked from execution till all the previous stores are completed. Later, when Epoch W arrives, the store to W completes and that entry is removed from BSQ (Figure 7(c)). Additionally, the line W is invalidated from thread 1's private cache adhering to STC rules. Similarly, stores to addresses V and U are completed in epochs V and U, respectively, and the cachelines V and U are invalidated from the private caches in these epochs (Figures 7(c) and 7(d)). Thus, by the end of epoch U, all stores are completed and the cachelines U, V, and W are invalidated. Throughout these epoch changes, thread 1's LdAcq is reading X = 0 and thread 1 continues to spin at that instruction. The store completion acknowledgements are sent to the core after the stores are completed in their own respective epochs and the store release can now be executed by thread 1 as shown in Figure 7(f). However, the released store (X = 1) sits in the BSQ till epoch X arrives and is drained from the BSQ later in epoch X (Figure 7(g)) marking its completion. At the beginning of epoch X, the cacheline X is invalidated in thread 1's private cache and a subsequent read to address X will now see the updated value (Figure 7(h)). Only after the load acquire condition is satisfied, the loads to U, V and W (lines 5, 6, and 7 of Figure 7(a1)) are executed by thread 1. These loads will also see the most up to date value from the shared L2 cache and the outcome will be (U,V,W) = (1,1,1) thus enforcing RC. It should be noted here that a load acquire did not invalidate private caches (as discussed in Section 4.1) and still achieved the write propagation by following the STC's epoch rules.

Ordering of writes: In the previous example, the stores are executed in the order U->V->W (Figure 7(a1)) but are completed in the order W->V->U (Figures 7(c)–7(e)). Since the implementation was adhering to RC model, maintaining store->store ordering was not a requirement. The only requirement was to order stores U, V and W before the RC synchronization marker, that is the store release to X, and that requirement was enforced by the core by blocking the execution of the store release till the completion of older stores. A strong model implementation on top of

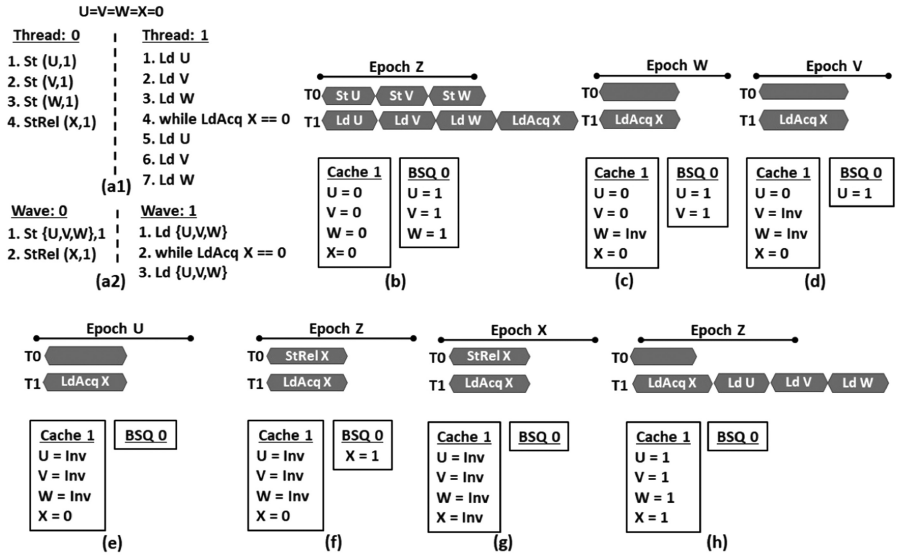

Fig. 7. Working example.

the STC can enforce a stricter store->store ordering by blocking the issue of a younger store till the older store completes. However, our RC implementation did not have that requirement and hence allowed out of order store completion within RC model permitted synchronization marker boundaries.

Correctness with multiband optimization: In the above example, bands W, V and U took three epochs to complete the stores. With multi-band optimization, epochs of these three bands can be granted simultaneously. That means epochs U, V, and W can be granted simultaneously and all three writes from the BSQ can be drained in this multiband epoch. The BSQ may be drained and stores may be completed in a different order compared to the above example. But granting epochs W, V, and U simultaneously still functions correctly and produces the same outcome (U,V,W) = (1,1,1).

5 DISCUSSION

5.1 Timestamp Coherence Protocols and STC

STC differs substantially from related prior work in timestamp-based coherence protocols [42, 46, 51, 53, 58]. The timestamp-based coherence protocols avoid invalidation messages by getting a time lease on an address and self-invalidating them after the time lease has expired. These protocols still follow the idea of an exclusive modifier or single writer for an address. The logical timestamp versions of these protocols also enforce the single-writer property, although in logical time. Thus, these timestamp-based protocols also grant processor-centric write permissions. STC does not grant write permission to a processor but to an epoch.

Moreover, storing time lease information as metadata in every cache block consumes space in both the private and shared caches [42, 51, 53, 58]. STC does not require cache blocks to store their lease information or any other metadata in their caches or controllers. Thus, STC is stateless whereas timestamp-based protocols are not.

Additionally, since these protocols need to store the timestamp metadata of all cache blocks in a shared cache, the shared cache is forced to be designed as an inclusive cache [51]. Since the

Table 2. Comparison of STC with Other Protocols

	Software assisted	Need exclusive write permission	Metadata overhead	Scalability	Write atomicity
GPUIn [25, 51] Coherence by invalidation messages	No	Yes	No	No (invalidation messages)	Yes
TC [51] Temoporal coherence	No	Yes	Yes (timestamp)	No/Limited (inclusive cache)	Yes (in the strong version)
RCC [42] Relativistic coherence	No	Yes (in logical time)	Yes (timestamp)	Limited (untracked blocks)	Yes
GTSC [53] Global timestamp coherence	No	Yes (in logical time)	Yes (timestamp)	Limited (untracked blocks)	Yes (in the strong version)
SRC [5, 23, 25, 39] Release consistency managed caches	Yes	No	No	Yes	No
STC **Spatiotemporal coherence**	**No**	**No**	**No**	**Yes**	**Yes**

size of an inclusive cache increases with the number of private caches, an inclusive cache is not scalable. Also, inclusive caches face the challenge of cache evictions. Since the timestamp-based protocols do not employ recall messages to solve the shared inclusive cache eviction challenge, they resort to the use of MSHRs to store the lease information of the evicted blocks [51]. This again leads to scalability challenges, because the MSHR demand will increase, perhaps dramatically. An alternate solution is to use a non-inclusive cache and then keep a single lease information for all the untracked cache blocks [42, 53]. However, this solution is not scalable either, because as the number of private caches increases, the fraction of tracked cache blocks decreases in the non-inclusive cache. Since all the untracked cache blocks now share the longest lease among them, a modifying processor will be forced to advance its current logical time beyond this long lease time resulting in lease expiry induced invalidations of cache blocks in its private cache.

Table 2 compares STC with some of the GPU protocols proposed in the past [25, 42, 51, 53]. The GPUIn is an invalidation-based GPU protocol that propagates a new write by invalidation messages [25, 51]. This is not a protocol optimized for GPUs [25, 51] but added to this comparison to highlight the characteristics of an invalidation-based protocol. RCC is the relativistic coherence proposed by Ren and Lis [42], GTSC is the coherence protocol for GPUs proposed by Tabbakh et al. [53] and **temporal coherence (TC)** is the coherence protocol proposed by Singh et al. [51]. All these protocols build on top of the timestamp-based coherence with the former two using logical timestamps proposed by Yu and Devadas [58]. The SRC is our baseline protocol with RC-managed caches and used in many previous studies [5, 23, 25, 39].

The GPUIn approach is not scalable because of the need for invalidation messages for every write and hence cannot be employed in a GPU. The timestamp-based coherence protocols provide write atomicity and avoid invalidation messages. Hence, scalability of these protocols is better than the GPUIn protocol. However, the need for a large inclusive cache to store lease information or the increasing number of untracked blocks with a non-inclusive cache limits their scalability. They also have the additional storage overhead of tracking the lease information of every cache line in the metadata of a cache. SRC is scalable but requires software assistance and cannot provide write atomicity. STC is scalable and provides write atomicity without software assistance and without incurring metadata overhead.

5.2 Global Ordering with STC

Global ordering is dictated by a directory in traditional protocols. The directory serves as the ordering point for coherence requests. However, since coherence messages are completely eliminated in STC, an explicit ordering point like a directory or snooping bus is not necessary. An ordering point is needed when the processors do not have any agreement on the modifying addresses and

the time of modification. With epochs and bands, the processors have mutually agreed upon the time and addresses they are modifying. As such, the only requirement is to synchronize epochs across all processors. This synchronization is the only purpose of EMU. Hence, EMU is not an ordering point.

5.3 Programmability of RC

Weak models are difficult to program and verify [2, 3, 11, 16–18, 26, 47, 48, 60, 61]. They rely on programmers to explicitly insert synchronization primitives to initiate communication. However, there are testimony from programmers about their struggles with relaxed model's synchronization primitives, including spending days trying to get code with just 2 addresses and 4 accesses to work [48].

The **heterogeneous-race-free (HRF)** model [27] of today's GPUs not only asks the programmer to initiate the communication but also to specify the communication scope. Scopes are essential for faster synchronization operations in a GPU. Today, these scopes are defined in terms of GPU execution hierarchy. However, recent works on GPU initiated networking [32] and fine-grained task scheduling [36] employ the **command processor (CP)** [7] (traditionally used for launching work on GPU) as a compute element in a GPU, making it difficult to define precise scopes, because CP threads are not part of the GPU execution hierarchy. A detailed discussion on this can be found elsewhere [40].

A strong model avoids all the above mentioned issues in a GPU. However, strong models require implicit coherence support and traditional hardware coherence mechanisms are not suitable for GPUs [51]. STC provides such implicit coherence support with low implementation and verification overhead.

5.4 Applicability of STC

STC allows writes only in their corresponding epochs and hence will delay write completion. While this is acceptable for latency tolerant GPUs, this may not be acceptable for latency sensitive CPUs. Hence, STC is better suited for GPUs. Among GPU applications, STC is generally suited for applications with sparse writes (write overhead from STC will not be pronounced) and high read locality (taking advantage of STC's invalidation-free acquire operation). Based on these observations, we list few GPU applications that are better suited for STC and few others that are less suited for STC.

Applications Better Suited for STC: Applications with back-to-back kernel launches and with read-data reuse across these kernels can benefit from STC. The baseline GPU invalidates the private caches at the beginning of every kernel launch (acquire operation), but the STC caches provide invalidation-free acquire operations thus allowing caching of data across kernel launches and consequently increasing the private cache hit rate. This scenario is observed in snapc application and is discussed in the result section. Also, task-parallel applications are a class of emerging GPU applications from the **high-performance computing (HPC)** domain that launches GPU kernels as dependent tasks and these applications, because of their high data reuse across kernels, can get benefited from STC [13, 14, 38, 41, 56].

Applications Less Suited for STC: One of the limitation of STC as compared to the baseline RC coherence is its inability to avoid coherence overhead for private data. In a GPU, the compiler produces spills to private segment when the data set cannot fit into the available registers. Spills may also be generated by a compiler for increasing the GPU occupancy by reducing per-workgroup register pressure. Irrespective of the reason for spills, STC may not be well-suited for applications with large amount of spills (private data). However, STC can leverage some of the classification

from prior work [12, 24, 49, 50] and allow the private data writes to complete without waiting for their epoch. This optimization is left for future work.

STC is generally not expected to perform well on applications with fine-grained synchronization and heavy data sharing. The write updates to the synchronization variables and shared data sets will be slow with STC. However, even the RC baseline performs poorly with data sharing because of the frequent synchronizations with acquire-release operations. Additionally, the STC's multi-band optimization can grant epochs for the synchronization variable and shared data structure simultaneously enabling faster write completion. Thus, STC can perform better than the baseline for fine-grained data sharing workloads as observed with the fg-share benchmark (discussed in the result section). Additionally, STC may not perform well for write-intensive workloads. For example, vector copy workload that copies one buffer to another observed a performance drop with STC (vec-cpy workload is discussed in the result section).

5.5 Suppressing the Acquire Operation

STC benefits from invalidation-free acquire operation for benchmarks with read-data reuse across kernel invocations. STC achieves this by avoiding caching of stale data in private caches thus making cache invalidation unnecessary during an acquire operation. The RC managed private caches cache both modified (by a peer core) and unmodified data. An acquire operation is needed only to invalidate the modified data and the baseline can get away by suppressing the acquire operation if there is a way to ensure the private cache is only caching unmodified data. But distinguishing between the modified and unmodified data is difficult. Hence the baseline resorts to the entire cache invalidation. There are mechanisms like Denovo protocol [21] that identify read-only regions similar to STC but with the help of software assistance and/or programmer inserted hints. Such models either put the onus on the programmer to annotate the access type of a data region and/or add constraints to the model such as data-race-freedom for their correct operation. STC's invalidation-free acquire operation ensures correct operation without software assistance, without relying on programmer inserted hints and without imposing any constraints on the model.

5.6 Address Bands

In our implementation, all addresses with the same contiguous N-bits starting from the SEB are grouped to the same band. However, it should be noted that the address to band mapping can be implemented in multiple ways with several other hashing schemes to map an address to a band including software/programmer defined address to band mapping. We are employing a simple hardware-only contiguous N-bit address to band mapping scheme. This will result in non-contiguous addresses grouped together to the same band. Although adaptive band optimization tries to minimize addresses with dissimilar read-write characteristics from being grouped in to the same band, there may be situations where adaptive band optimization could not separate addresses with dissimilar characteristics because of this address to band mapping scheme. Such situations may impact the performance but functional correctness is still ensured by the STC's epoch rules.

Also, since the epochs are granted in a round robin fashion, the bands are given write permission in ascending order (i.e., band-0, band-1, . . . ,band-N). This may also result in performance degradation if the cores write to these bands in a different order. However, epoch skipping optimization mitigates this issue by granting epochs to bands with pending write requests. This issue can be mitigated further by exposing the bands to the programmer. A programmer can then optimize a code's access pattern for band locality—that is consecutive accesses are issued to the same band—for additional performance benefits. This optimization is left for future work.

Table 3. Simulation Parameters

GPU CUs and Clock	8 CUs @ 1 GHz
GPU L1 D-Cache	64 KB (64-way associative)
GPU L1 I-Cache	16 KB (16-way associative)
GPU Shared L2 Cache	512 KB (16-way associative)
GPU L2 Latency	160 GPU cycles roundtrip
GPU Memory Latency	260 GPU cycles roundtrip
Number of Epochs	16 (N = 4 bits)
EMU Clock	2 GHz
EMU Wakeup Cycles	100 GPU cycles
Default SEB	12
BSQ (and coalescing buffer)	256 entries

6 METHODOLOGY

Simulator: We used the gem5 GPU simulator [23] that simulates the GCN3 ISA [7]. The simulated GPU has 8 CUs and each CU hosts a 64 KB private L1 data cache. The 16 KB instruction cache is shared by 4 CUs and the 512KB L2 cache is shared by all CUs. This cache hierarchy is modeled in Ruby [34]. Table 3 lists the remaining simulation parameters.

The baseline GPU is discussed in Section 4.1 and is extensively used in many previous studies [5, 23, 25, 37]. We implemented STC on this baseline GPU and evaluated STC against it. We evaluated four versions of STC:

(1) STC-NV: Naive STC implementation without any optimizations.
(2) STC-ES: STC with epoch skipping optimization.
(3) STC-AB: STC-ES with adaptive band optimization.
(4) STC-MB: STC-AB with multiband optimization. In our implementation, we allowed up to 4 simultaneous bands.

Additionally, we also compare STC against the temporal coherence proposed in the prior work [51]. We also evaluated a cacheless baseline (disabling L1 but keeping L2), but it performed so poorly for some benchmarks (up to 62.13% slower), we decided to instead baseline against a realistic current-generation design [5, 23, 25, 37].

The EMU in our STC implementation wakes up every 100 cycles and changes the epoch. We simulate STC with 16 epochs (4 bits to identify a band/epoch, that is N = 4). The default SEB is 12. The SEB is modified dynamically by the adaptive band optimization. However, since 12 bits indicate the page boundary and 32 bits (4 GB) is the maximum addressable physical memory available in some GPUs, we decided to limit the SEB in the range of 12 to 32.

Workloads: We evaluated STC with benchmarks from AMD Compute App [9], HCC Example App [8], Rodinia [20] and Pannotia benchmark suites [19]. We also developed a fine-grain data sharing benchmark (fg-share) in which all workgroups (thread blocks in CUDA) attempt to enter a critical section, and then read and update a shared data structure in place, emulating a centralized ledger update. With the workgroups contending to enter into the mutually exclusive critical section implemented with atomic compare-and-swap, and with in-place updates, this benchmark significantly pressurizes the coherence mechanism. We also evaluated a cache-reuse benchmark that launches 10 kernels with each kernel reading the same read-only array and updating a second array. Additionally, a write-intensive **vector copy (vec-cpy)** benchmark that copies a buffer to another is also evaluated.

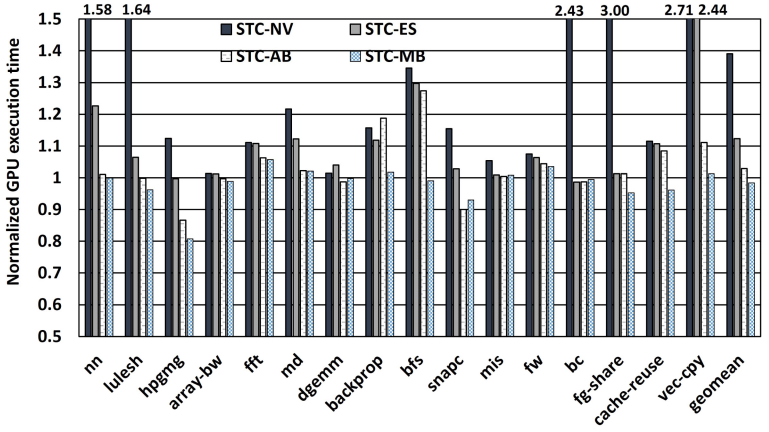

Fig. 8. GPU execution time normalized to the baseline.

7 RESULTS

7.1 Execution Time

Figure 8 shows the GPU execution time of benchmarks normalized to the baseline. It can be seen that the unoptimized STC (STC-NV) performed significantly worse than the baseline with the mean runtime increasing by as much as 39.04% across all evaluated benchmarks. However, the epoch skipping optimization alone regained most of the lost performance and STC-ES was only 12.29% slower than the baseline. STC-AB further brought down the performance penalty to 2.93%. However, the full optimization enabled STC-MB improved the performance by 1.63% compared to the baseline.

STC-NV blocks the writes and invalidates the caches during every epoch. Additionally, the average time to change an epoch was observed to be 36 cycles (including time for the four-way handshake), and writes are blocked during this epoch transition as well. All of these contributed to the execution slow down. STC-ES filters out the epochs without any write requests and mitigates both these problems, thus improving performance. The reduction in the number of epoch transitions as compared to STC-NV is shown in Figure 10. Both STC-ES and STC-AB observed extreme reductions in the number of epoch transitions. With epoch skipping optimization, both STC-ES and STC-AB only transition to epochs that have writes pending to them and this eliminates unnecessary epoch transitions. Figure 10 also shows that the STC-AB optimization reduces epoch transitions by three orders of magnitude, which is also reflected in the execution time reduction shown in Figure 8.

Figure 9 shows the L1 hit rate of the baseline compared to the STC-AB. That figure also shows the hit rate after disabling the acquire operation. The benchmarks that show increased hit rate without acquire operation have read data reuse across kernel launches and can potentially benefit from STC's acquire-less operation. From that figure, lulesh, hpgmg, snapc, bc, and cache-reuse exhibit this behavior. For these benchmarks, it can be seen that STC-AB improved the hit rate compared to the baseline.

Figure 9 also shows that STC-AB restored the hit rate to baseline levels for most benchmarks. However, the performance (Figure 8) was still behind the baseline. This suggests that the cache hit rate has little impact on the performance degradation and the impact was mainly caused by blocked writes as opposed to cache invalidations in an epoch. We mitigated this issue with the STC-MB optimization by allowing concurrent writes from multiple bands, reducing the number of blocked

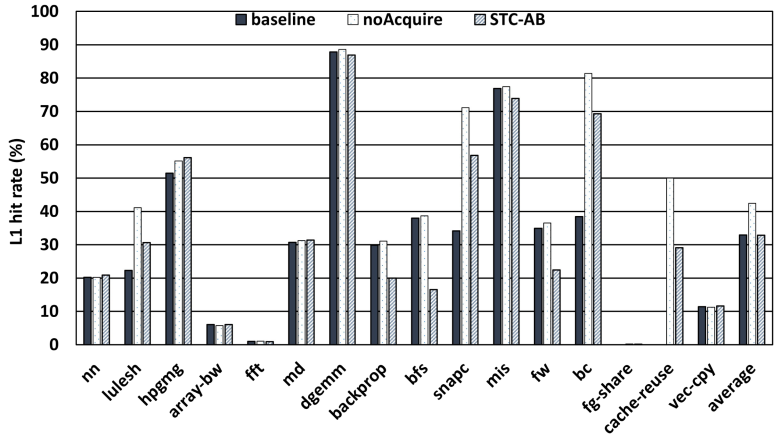

Fig. 9. GPU L1 cache hit rate.

writes waiting on their epoch. This directly translated to performance as seen in Figure 8. With this final optimization applied, STC-MB achieved 1.63% speedup over the baseline across all evaluated workloads. More significantly, the benchmarks with increased cache hit rate (lulesh, hpgmg, snapc, bc, cache-reuse) because of data reuse across kernels observed a geomean speedup of 7.13%. Next, we delve deeper into the performance of a few benchmarks.

snapc, lulesh: The computation progresses in time steps in snapc, with multiple kernels (2,992 kernels) launched to complete a time step. Similarly, lulesh launches 3,380 kernels. However, many of these kernels have read data reuse. Since the baseline GPU employs software-managed cache coherence, the caches are invalidated at every kernel boundary (acquire operation at the beginning of a kernel) and the baseline cannot take advantage of this reuse. Thus, STC achieves a higher cache hit rate and better performance for these benchmarks.

fg-share, hpgmg: fg-share uses atomic operations to synchronize the workgroups that are contending to enter into the critical section. Inside the critical section, each workgroup reads the updates from the previous workgroup and then modifies them. Since the shared data structure is updated in place, the read and write datasets are not separate. Consequently, the STC-AB could not identify a read-only dataset and performed similar to the STC-ES (Figure 8). The performance loss can be attributed to the write propagation delay of the atomic synchronization. This synchronization has to wait for its turn behind the shared data structure's epoch. But with STC-MB granting epochs for synchronization variable and shared data structure simultaneously, STC achieved 4.75% speedup against the baseline.

The read-modify-write access pattern of the shared data structure does not provide any caching benefit (Figure 9). However, the baseline allocates these loads in the caches, creating allocation/deallocation overhead (tag array lookup and associated port/bank conflicts, data allocation overhead including data array port/bank conflicts, eviction overhead) and thus delaying the load completion [29, 54]. STC-MB issues these loads as uncached, thus completely bypassing the cache (after all cache sets are accessed once for lazy invalidation mechanism, Section 4.2). Thus, loads are completed faster with STC-MB, resulting in speedup against the baseline.

hpgmg benefited from both read-data reuse across kernel launches as seen from its improved hit rate (Figure 9) and also from issuing 33.75% loads as uncached similar to fg-share.

vec-cpy: Each work-item in a vector copy benchmark copies one data element from a source to a destination buffer. This presents a pathological case for STC, because the compute to memory

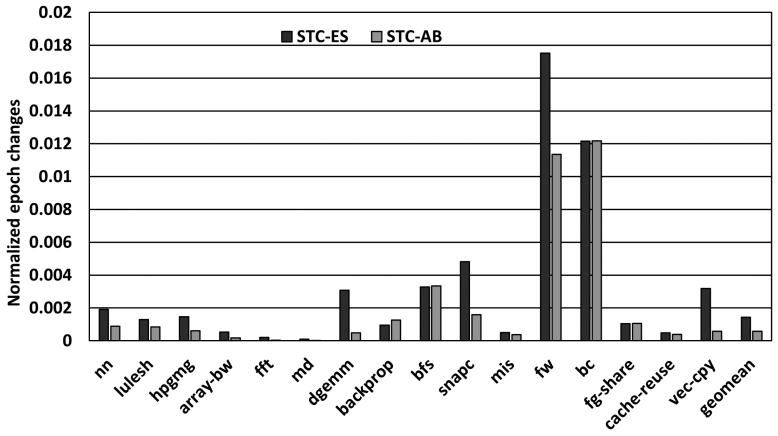

Fig. 10. Decrease in the number of epoch transitions compared to STC-NV.

ratio is low, and the GPU has less opportunity to overlap long-latency memory accesses with work. Since simultaneous writes from multiple work-items across the GPU result in writes to multiple epochs at all times, epoch skipping could not find many epochs to skip as observed from a large number of epoch changes by STC-ES in Figure 10. Consequently, STC-ES performed poorly (Figure 8). However, the adaptive band optimization could take advantage of the read-only source and write-only destination buffers, thus regaining most of the lost performance (within 11.1% of baseline). STC-MB further improved the performance to within 1.3% of the baseline even for this pathological benchmark.

To summarize, the take away is that STC provides performance gains to read-data reuse kernels without negatively impacting the performance of other kernels. It should be noted that the main objective of STC is to simplify the hardware-only coherence mechanism in a GPU while providing comparable performance to that of a software-managed coherence mechanism, and STC has achieved this objective.

7.2 Sensitivity Analysis

Sensitivity to Epoch Configurations and Cache Size: We experimented with 50-, 100-, 150-, 300-, and 450-cycle epochs, and the performance varied by 3%. The best performing 100-cycle epoch was used for our evaluations as mentioned in Table 3. The epoch duration primarily impacts the latency of the stores blocked by STC. Store completion delay can stall a wavefront if either (a) dependent reads are delayed because of pending older writes or (b) wavefront is waiting on a release operation. Even stalling one wavefront will not impact performance, because the GPU pipeline has the ability to switch execution to the next ready wavefront. Because of these GPU characteristics, store latency, and consequently epoch cycle duration had little impact on the GPU performance. Also, we ran experiments with 8, 16, and 32 epochs, and the performance variation was less than 2%. With 16 kB private cache, STC outperformed the baseline by 1.97%.

Sensitivity to BSQ Size: The max occupancy of the 256-entry BSQ was found to be between 5 and 250 entries and the mean occupancy was between 0.5 and 102 entries for the evaluated benchmarks. The 16-banked BSQ is capable of draining up to 32 entries in 1 cycle (2 entries drained per bank per cycle, 8 cycles for draining all 256 entries) and most of the benchmarks (14 of 16) have a mean occupancy of less than 32. Thus, these benchmarks were able to cycle through the BSQ in 1 cycle on average. Even for the benchmark with a mean occupancy of 102 entries (fft), the draining

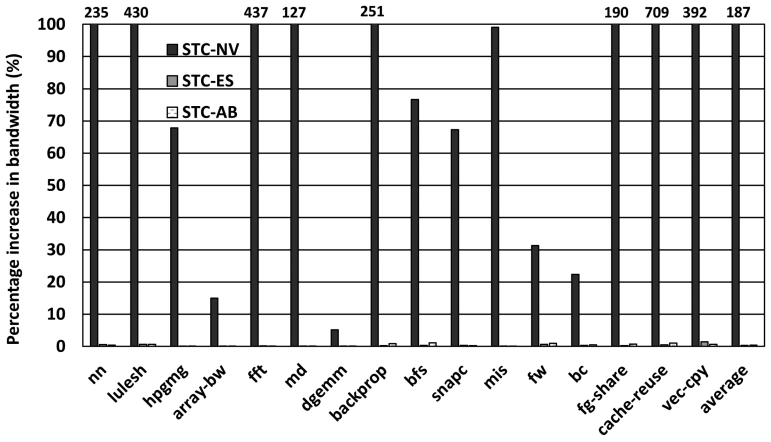

Fig. 11. STC bandwidth consumption compared to the baseline.

overhead was 4 cycles, indicating that draining BSQ is not a bottleneck during epoch transitions. Additionally, only entries to the current epoch are issued to the cache. Thus, the issue rate to cache is much less.

We compared the performance of benchmarks for BSQ sizes ranging from 64 entries to the baseline 256 entries. The performance was relatively insensitive to BSQ size with the geomean across all benchmarks comparable to the 256-entry baseline. As mentioned earlier, 14 of 16 benchmarks only had a mean BSQ occupancy of less than 32 entries and hence the benchmarks remained insensitive to the evaluated BSQ sizes. A BSQ size of less than 32 entries could have made a performance impact. However, the evaluated GPU has a 64 thread wavefront and hence the BSQ must have atleast 64 entries to accommodate up to 64 uncoalesced stores originating from the same wavefront instruction.

7.3 Bandwidth and Hardware Cost

Figure 11 shows the increase in bandwidth consumption due to epoch requests. STC-NV consumes significant bandwidth, because it changes the epochs even when the GPU is idle (e.g., between two kernel launches). The figure also shows the bandwidth consumption of STC-ES and STC-AB. With STC-ES, epochs are changed only when the GPU demands an epoch and hence the epoch change is naturally stopped when the GPU is inactive, resulting in the reduction of bandwidth. STC-AB further reduces the epoch transitions by changing the bands to epoch mapping. As such, the GPU bandwidth increased by only 0.43%, despite these optimizations sending additional messages. Hence, the impact of the optimized STC implementations on bandwidth can be considered a nonissue.

The bandwidth cost of a purely hardware-based implementation for a similar set of benchmarks (8 of the 16 benchmarks are common) is discussed in prior work [40]. Compared to the 17% increase in GPU bandwidth reported there, STC is far better with only 0.43% increase in bandwidth consumption over the baseline.

STC's low bandwidth consumption even for fine-grained data sharing benchmarks can be explained by a comparison of its operation to a traditional coherence protocol. Suppose few cores are performing a producer-consumer data sharing via a data structure. In a traditional protocol, the modifying cores send write requests to the directory, the directory then sends invalidations to the sharers and the owners forwards the latest data along with exclusive permissions to the

Table 4. Hardware Cost

EMU	4-bit epoch and 8-bit wakeup counters 16-entry ERV (33 bits per entry) 16-entry ETV (1 bit per entry)
BSQ	1 per CU, 256-entry, 32 bits per entry

requestors. This sequence is performed for every single address being modified in that data structure and most importantly repeats for every single producer-consumer exchange. With STC, an epoch or few epochs (if the entire data structure cannot be grouped into the same band) are granted to this data structure in response to epoch request(s), and the producer-consumer exchange can continue without any further coherence messages. With multiband optimization, the epoch(s) to this data structure can be always granted and these cores can perform repeated producer-consumer exchanges without issuing any further coherence messages for all addresses in that data structure. This is the reason STC's bandwidth overhead was only 0.43% even with fine-grain data sharing benchmarks (fg-share, bc), and coarse-grain data sharing benchmarks (snapc, fw, mis).

Table 4 lists the hardware cost for implementing the optimized STC-AB protocol with 16 epochs and the EMU waking up every 100 cycles. It can be seen from the table that the hardware cost of implementing the EMU is negligible.

7.4 Scalability

Figure 12 shows the performance of benchmarks for 8, 16, and 32 CUs normalized to their respective baselines. The geomean1 reports the mean performance across all evaluated benchmarks whereas geomean2 reports the mean performance of STC friendly benchmarks with data reuse across kernels (lulesh, hpgmg, snapc, bc). The mean performance was observed to be relatively similar across configurations with different numbers of CUs. However, benchmarks like nn, dgemm and vector-copy observed a performance slowdown due to the increase in the epoch synchronization overhead (increased from 36 GPU cycles with 8 CUs to 79 GPU cycles with 32 CUs at steady state after the read-only bands are separated by the adaptive band optimization). The STC friendly benchmarks still observed performance benefits across the range, because they were able to exploit STC's caching benefits and nullify the negative impact of the higher epoch synchronization overhead. Thus, it can be concluded that STC scales well for workloads with read-data reuse across kernels. The two benchmarks fg-share and cache-reuse did not have enough workgroups to scale to 32 CUs with the same occupancy (number of wavefronts simultaneously active on a CU) as the baseline. Running GPU with a lower occupancy will hamper its ability to hide memory latency and hence these benchmarks are omitted from the results. In summary, since the epoch synchronization overhead increases with CUs in general, STC is better suited for small GPUs with limited numbers of CUs such as the ones used in embedded platforms and mobile SoCs.

7.5 Comparison with Prior Work

We compared STC with the strong variant of the timestamp-based temporal coherence protocol (temporal coherence strong - TCS) [51], because it also provides write-atomicity similar to STC. Across all evaluated benchmarks, the TCS was slower than the STC by 3.46%. Although both STC and TCS avoid invalidation messages, TCS's exclusive modifier requirement delays a write till the global lease on a block expires. This makes the TCS slower than STC, especially for fine-grain data sharing operations.

While performance advantage is not significant, STC has storage overhead and hardware complexity advantages. TCS requires every cache block to store additional 32-bit lease information.

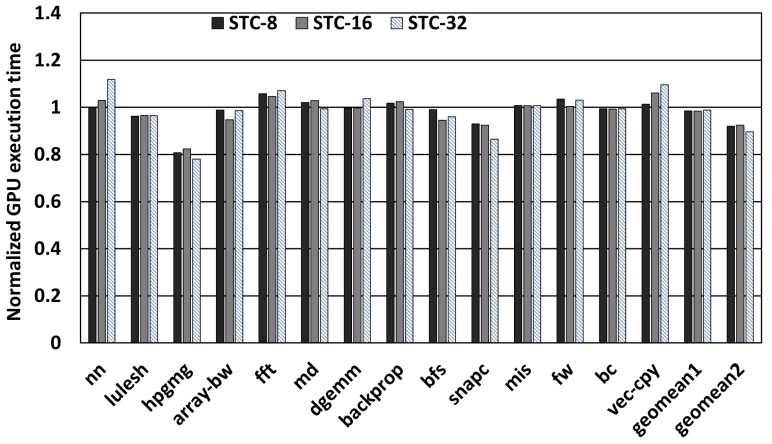

Fig. 12. Scalability.

Additionally, to store the lease information of evicted blocks from the shared L2, the TCS employs MSHRs (we used 128-entry MSHRs per L2 bank). Thus, for the simulated system with 8,192 (8 CUs and 1,024 cache blocks per CU) cache blocks in both GPU L1 and L2, and with 512 MSHR entries across four L2 banks, the TCS requires 66 kB of storage to store lease information whereas STC requires only 8 kB additional storage for its BSQs (8 BSQs and 256 32-bit entries per BSQ). The MSHRs not only add storage overhead but these associatively looked-up structures are difficult and costly to implement beyond a few entries [37, 39]. STC does not require metadata storage and associative structures for its implementation. Additionally, TCS sends 32-bit timestamp information with every read request resulting in a bandwidth increase of 3.57% whereas STC's epoch requests increased it by only 0.43% across all evaluated benchmarks.

8 RELATED WORK

The reduction of coherence traffic in GPUs has been the subject of many studies in the past [5, 12, 25, 37, 42, 51, 53]. Prior work has also explored coherence across multiple GPUs [43, 57]. Power et al. and Basu et al. proposed tracking addresses at larger granularities called regions to reduce the number of invalidation messages [12, 37]. Both these techniques track coherence permissions over a larger region granularity. STC bands are different from regions in three ways: (a) regions track coherence permissions whereas bands do not, (b) regions are modified by processors with exclusive/private permissions but STC bands are modified exclusively in its epoch (processor versus epoch centric write permissions), and (c) regions are contiguous addresses whereas grouping addresses into a band is entirely implementation-specific. Denovo protocol [21] also identifies read-only regions similar to STC but STC does it without any software assistance. Sinclair et al. provided a taxonomy of commonly used CPU/GPU coherence protocols based on (a) invalidation initiator, and (b) tracking up-to-date copy of the data [47]. Since invalidation initiator is an epoch in STC and the up-to-date copy is maintained by write-through operation, STC differentiates itself from those protocols. Finally, Alisafaee proposed Spatiotemporal Coherence Tracking directory [4] that tracks private data at region granularity. However, this proposal is an efficient directory design to reduce the directory capacity, not a coherence protocol.

The timestamp-based coherence protocols [42, 51, 53, 59] eliminate invalidation messages by self-invalidation based on expiring time lease. Although this method eliminates invalidation traffic, it suffers from scalability and storage overheads as previously discussed in Section 5. STC does

not have these drawbacks. Finally, the Racer protocol by Ros and Kaxiras provides store atomicity without invalidations and per-processor write permission [44]. Racer achieves this by dynamically detecting read-after-write data races and self invalidating the racing reader's cache on a data race. STC avoids the need for dynamic race detection and self invalidation by restricting writes to an epoch and making the potentially modifiable data non-cacheable in private caches.

STC simplifies the coherence mechanism by establishing a mutual agreement among all the processors about the modification of addresses. Ros and Kaxiras also leveraged a similar concept for non-speculative store coalescing without deadlocks [45]. The globally agreed lexicographical order for acquiring cache permissions prevents cyclic cache permission dependencies making the implementation deadlock-free. Ibrahim et al. [28] proposed a shared L1 architecture for GPUs that obviates the need for coherence among the L1-sharer CUs. However, when such a system scales with shared L1 clusters, the STC can provide coherence across these clusters with negligible hardware cost.

9 CONCLUSION

We introduce turn-based STC in this article. STC grants modification rights to an address to epochs as opposed to processor cores. This novel way of assigning write permissions enables STC to behave like traditional hardware cache coherence but with negligible coherence traffic. Evaluation of our implementation of optimized STC shows that our implementation improves performance compared to the baseline GPU RC coherence while providing write atomicity, without needing any software assistance.

ACKNOWLEDGMENTS

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (Dec. 1996), 66–76. https://doi.org/10.1109/2.546611

[2] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2012. Software verification for weak memory via program transformation. Retrieved from http://arxiv.org/abs/1207.7264.

[3] Jade Alglave and Luc Maranget. 2011. Stability in weak memory models. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*.

[4] M. Alisafaee. 2012. Spatiotemporal coherence tracking. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 341–350.

[5] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy release consistency for GPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE Press, Piscataway, NJ, Article 26, 13 pages. Retrieved from http://dl.acm.org/citation.cfm?id=3195638.3195669.

[6] AMD. 2012. AMD Graphics Cores Next (GCN) Architecture. Retrieved from https://goo.gl/GPvy8R.

[7] AMD. 2016. AMD GCN3 ISA Architecture Manual. Retrieved from https://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual.

[8] AMD. 2016. HCC Example Apps. Retrieved from https://github.com/ROCm-Developer-Tools/HCC-Example-Application.

[9] AMD. 2019. Compute Apps. Retrieved from https://github.com/AMDComputeLibraries/ComputeApps.

[10] AMD. 2019. User Guide for AMDGPU Backend. Retrieved from https://llvm.org/docs/AMDGPUUsage.html.

[11] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*. ACM, New York, NY, 7–18. https://doi.org/10.1145/1706299.1706303

[12] Arkaprava Basu, Sooraj Puthoor, Shuai Che, and Bradford M. Beckmann. 2016. Software assisted hardware cache coherence for heterogeneous processors. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS'16)*. ACM, New York, NY, 279–288. https://doi.org/10.1145/2989081.2989092

[13] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A dependence-aware task-based execution framework for GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. ACM, New York, NY, 54–67. https://doi.org/10.1145/3178487.3178492

[14] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. 2011. DAGuE: A generic distributed DAG engine for high-performance computing. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. 1151–1158. https://doi.org/10.1109/IPDPS.2011.281

[15] D. Bouvier and B. Sander. 2014. Applying AMD's kaveri APU for heterogeneous computing. In *Proceedings of the IEEE Hot Chips 26 Symposium (HCS'14)*.

[16] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 12–21. https://doi.org/10.1145/1250734.1250737

[17] Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective program verification for relaxed memory models. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*. Springer-Verlag, Berlin, 107–120. https://doi.org/10.1007/978-3-540-70545-1_12

[18] Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing concurrent programs on relaxed memory models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*. ACM, New York, NY, 122–132. https://doi.org/10.1145/2001420.2001436

[19] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'13)*.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*.

[21] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. 2011. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 155–166.

[22] HSA Foundation. 2016. HSA platform system architecture specification 1.1. Retrieved from http://www.hsafoundation.com/?ddownload=5114.

[23] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. 2018. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 608–619. https://doi.org/10.1109/HPCA.2018.00058

[24] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 184–195. https://doi.org/10.1145/1555754.1555779

[25] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 189–200. https://doi.org/10.1109/HPCA.2014.6835930

[26] Mark D. Hill. 1998. Multiprocessors should support simple memory-consistency models. *Computer* 31, 8 (Aug. 1998), 28–34. https://doi.org/10.1109/2.707614

[27] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 427–440. https://doi.org/10.1145/2541940.2541981

[28] Mohamed Assem Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. 2020. Analyzing and leveraging shared L1 caches in GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT'20)*. Association for Computing Machinery, New York, NY, 161–173. https://doi.org/10.1145/3410463.3414623

[29] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 272–283. https://doi.org/10.1109/HPCA.2014.6835938

[30] Mahmoud Khairy, Mohamed Zahran, and Amr G. Wassal. 2015. Efficient utilization of GPGPU cache hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15)*. ACM, New York, NY, 36–47. https://doi.org/10.1145/2716282.2716291

[31] G. Krishnan, D. Bouvier, and S. Naffziger. 2016. Energy-efficient graphics and multimedia in 28-nm Carrizo accelerated processing unit. *IEEE Micro* 36, 2 (2016), 22–33.

[32] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2018. ComP-net: Command processor networking for efficient intra-kernel communications on GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. ACM, New York, NY, Article 29, 13 pages. https://doi.org/10.1145/3243176.3243179

[33] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*.

[34] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 92–99. https://doi.org/10.1145/1105734.1105747

[35] NVIDIA. 2009. Nvidia Tesla V100 GPU Architecture. Retrieved from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[36] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain task aggregation and coordination on GPUs. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 181–192. https://doi.org/10.1145/2678373.2665701

[37] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 457–467. https://doi.org/10.1145/2540708.2540747

[38] Sooraj Puthoor, Ashwin M. Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M. Beckmann, and Gregory Rodgers. 2016. Implementing directed acyclic graphs with the heterogeneous system architecture. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU'16)*. ACM, New York, NY, 53–62. https://doi.org/10.1145/2884045.2884052

[39] Sooraj Puthoor and Mikko H. Lipasti. 2018. Compiler assisted coalescing. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. ACM, New York, NY, Article 11, 11 pages. https://doi.org/10.1145/3243176.3243203

[40] Sooraj Puthoor and Mikko H. Lipasti. 2021. Systems-on-chip with strong ordering. *ACM Trans. Archit. Code Optim.* 18, 1, Article 15 (Jan. 2021), 27 pages. https://doi.org/10.1145/3428153

[41] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M. Beckmann. 2018. Oversubscribed command queues in GPUs. In *Proceedings of the 11th Workshop on General Purpose GPUs (GPGPU'18)*. ACM, New York, NY, 50–60. https://doi.org/10.1145/3180270.3180271

[42] X. Ren and M. Lis. 2017. Efficient sequential consistency in GPUs via relativistic cache coherence. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 625–636. https://doi.org/10.1109/HPCA.2017.40

[43] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans. 2020. HMG: Extending cache coherence protocols across modern hierarchical multi-GPU systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*. 582–595.

[44] A. Ros and S. Kaxiras. 2016. Racer: TSO consistency via race detection. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783736

[45] A. Ros and S. Kaxiras. 2018. Non-speculative store coalescing in total store order. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 221–234. https://doi.org/10.1109/ISCA.2018.00028

[46] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas. 2011. Library cache coherence. http://hdl.handle.net/1721.1/62580.

[47] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, 647–659. https://doi.org/10.1145/2830772.2830821

[48] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Association for Computing Machinery, New York, NY, 161–174. https://doi.org/10.1145/3079856.3080206

[49] A. Singh, S. Aga, and S. Narayanasamy. 2015. Efficiently enforcing strong memory ordering in GPUs. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 699–712. https://doi.org/10.1145/2830772.2830778

[50] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 524–535. Retrieved from http://dl.acm.org/citation.cfm?id=2337159.2337220.

[51] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 578–590. https://doi.org/10.1109/HPCA.2013.6522351

[52] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[53] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. 2018. G-TSC: Timestamp based coherence for GPUs. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 403–415.

[54] Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez. 2015. Adaptive GPU cache bypassing. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*.

[55] D. Vantrease, M. H. Lipasti, and N. Binkert. 2011. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*. 132–143.

[56] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. 2015. Hierarchical DAG scheduling for hybrid distributed systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. 156–165. https://doi.org/10.1109/IPDPS.2015.56

[57] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Oreste Villa. 2018. Combining HW/SW mechanisms to improve NUMA performance of multi-GPU systems. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE Press, 339–351. https://doi.org/10.1109/MICRO.2018.00035

[58] Xiangyao Yu and Srinivas Devadas. 2015. Tardis: Time traveling coherence algorithm for distributed shared memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE Computer Society, Washington, DC, 227–240. https://doi.org/10.1109/PACT.2015.12

[59] Xiangyao Yu, Hongzhe Liu, Ethan Zou, and Srinivas Devadas. 2016. Tardis 2.0: Optimized time traveling coherence for relaxed consistency models. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. IEEE, 261–274.

[60] Sizhuo Zhang, Arvind, and Muralidaran Vijayaraghavan. 2016. Taming weak memory models. Retrieved from http://arxiv.org/abs/1606.05416.

[61] Sizhuo Zhang, Muralidaran Vijayaraghavan, Andrew Wright, Mehdi Alipour, and Arvind. 2018. Constructing a weak memory model. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE Press, Piscataway, NJ, 124–137. https://doi.org/10.1109/ISCA.2018.00021

Received 4 August 2022; revised 26 January 2023; accepted 12 March 2023