Efficiently Enforcing Strong Memory Ordering in GPUs

Abhayendra Singh Shaizeen Aga Satish Narayanasamy
University of Michigan, Ann Arbor
{ansingh, shaizeen, nsatish}@umich.edu

ABSTRACT

GPU programming models such as CUDA and OpenCL are starting to adopt a weaker data-race-free (DRF-0) memory model, which does not guarantee any semantics for programs with data-races. Before standardizing the memory model interface for GPUs, it is imperative that we understand the tradeoffs of different memory models for these devices. While there is a rich memory model literature for CPUs, studies on architectural mechanisms and performance costs for enforcing memory ordering constraints in GPU accelerators have been lacking.

This paper shows that the performance cost of SC and TSO compared to DRF-0 is insignificant for most GPGPU applications, due to warp-level parallelism and in-order execution. For the remaining challenging applications that exhibit significant overhead for SC, we show that commonly employed memory ordering optimizations in CPUs are either expensive or ineffective for GPUs. We propose a GPU-specific non-speculative SC design that takes advantage of high spatial locality and temporally private data in GPU applications. Results show that the proposed design is effective in eliminating the performance gap between SC and DRF-0 in GPUs.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

Keywords

Memory consistency model, GPUs, OpenCL

1. INTRODUCTION

Graphics Processing Units (GPUs) are now deployed across a wide range of systems that vary from mobile platforms to super computers. Modern GPUs are no longer limited to graphics applications, and are also being used as data parallel accelerators for general purpose programs. Programming models such as CUDA [1] and OpenCL [2] have enabled developers to exploit data-parallelism in general-purpose applications using GPUs (referred as GPGPU applications). Most of these GPGPU applications are not purely data-parallel and involve inter-thread communications. Ensuring correctness of these applications in the presence of wild shared-memory communication remains a challenge.

As is the case with any shared-memory multi-processor system, complexity of GPU parallel programming critically depends on the memory model of the GPU platform. Unfortunately, GPU vendors have largely ignored the need for formally defining the memory model semantics of their devices. Even today, GPU vendors' programming guides only provide informal descriptions about their memory model, leaving programmers with no choice but to rely on folklore assumptions. Not surprisingly, researchers are starting to find some serious mismatches between the memory model assumptions commonly made by the GPU programmers and what the hardware implementations end up providing [3].

Recently, there have been efforts to standardize OpenCL's [4] memory model based on C11's [5] data-race-free-0 (DRF-0) [6] model. It guarantees sequential consistency (SC) for data-race-free programs, but no semantics for programs with data-races. Researchers have refined this model for CPU-GPU devices by defining heterogeneous data-races [7], which is based on a revised happens-before model that accounts for different thread-visibility scopes for memory operations.

Before adopting a weaker memory model (DRF-0) for GPUs, it is imperative that we understand the costs of a stronger memory model such as TSO and SC for GPUs. A recent study investigated the cost of various memory models in a high-throughput processor with several in-order cores [8]. However, for a GPU architecture, several questions remain unanswered, which we seek to answer in this paper.

- What are the architectural mechanisms necessary in GPUs to enforce various types of memory ordering constraints (DRF-0, TSO, SC)? What are their performance costs?

MICRO-48, December 05 - 09, 2015, Waikiki, HI, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4034-2/15/12 ...$15.00.
http://dx.doi.org/10.1145/2830772.2830778.

- What is the interplay between the common GPU architectural design choices such as warp-scheduling and write-through caches, and the memory ordering constraints?
- Prefetching and in-window speculation [9] work remarkably well for CPUs. Are they feasible in GPUs? Are they equally effective in mitigating the performance overhead of memory ordering constraints in GPUs?
- How do we engineer a GPU-specific optimization for reducing strong SC memory ordering constraints?

A GPU core is typically assigned to tens of warps, where each warp contains as many threads as the number of SIMD lanes. Memory ordering constraint (RMO, TSO, or SC) can be naïvely implemented in GPUs by restricting the warp scheduling policy. To support SC, a core can issue a memory operation from a warp only if all of its preceding memory operations from the warp have completed. If a warp is stalled due to a memory ordering constraint, the core can issue instructions from another ready warp. Thus, warp-level-parallelism is effective in reducing memory ordering stalls. Furthermore, GPU's in-order core is limited in its capability in exposing intra-thread memory-level-parallelism. In GPUs with write-back caches, these two factors combine together to effectively nullify the performance overhead of SC as compared to RMO (which supports DRF-0 model) for a majority of applications we studied. Surprisingly, for some applications, SC outperformed RMO. We analyze this anomaly in depth in Section 6.1.

However, for a few challenging applications, overhead of SC is quite significant (maximum is 3x). We also found that overhead of TSO was also significant in these applications (maximum is 88%). When we employed store-buffer optimization, maximum TSO overhead reduced to about 50%. This optimization relied on one large store-buffer per warp – a significant area cost (48 KB). Given that the TSO overhead is still significant for the challenging applications, we do not see a compelling case to weaken the SC guarantees and choose TSO.

Replacing write-back caches with write-through caches eliminated the performance gap between SC and RMO for all applications, including the challenging applications we discussed above. We analyze the reasons in Section 6.5.

For GPUs with write-back caches, we need an efficient solution for reducing the SC overhead for the challenging applications noted above. In-window speculation [9] is inapplicable for simple GPU cores. Prefetching [9] cache blocks while waiting to resolve memory ordering constraints is a cheaper alternative, and is commonly employed in CPUs. However, to our surprise, we found that it is ineffective in reducing SC or TSO overhead, and in some cases it even reduced the performance. The reason is that, in a GPU core that multiplexes between numerous threads, there are too many premature prefetches, where a prefetched cache block gets evicted before it is used, that increase contention in memory hierarchy and can also evict useful cache blocks.

To reduce the SC overhead for the applications with high overhead, we devise a non-speculative design based on Shasha and Snir's observation that memory ordering constraints can be relaxed for accesses to locations that are private to a thread or are shared in read-only fashion [10] (referred as safe accesses and locations respectively). Singh et al. [11] exploited this observation to build an efficient SC implementation for CPUs that identified safe accesses by observing data sharing at page granularity. We follow this memory access classification scheme, but at a much larger granularity (16KB). We can afford a higher granularity because data accessed by a GPU core typically exhibit very high spatial locality [12].

Despite the similarity in memory access classification schemes, significant microarchitectural differences between GPUs and CPUs necessitate building a GPU specific solution (Section 4). These differences pertain to virtual memory support, instruction execution, number of concurrent threads per core (SM for GPU), presence of a shared memory in SM and partitioned address space, etc. We show that our proposed solution eliminates almost all the SC overhead in the applications we studied.

Online Material: A correctness proof of the design presented in Section 4 is available online [13].

2. MEMORY CONSISTENCY MODEL OF MODERN GPUS

In this section we describe memory models supported by current GPUs. First, we briefly describe GPU architecture and then discuss concurrency support available to programmers under CUDA and OpenCL programming models.

2.1 GPU Architecture

Figure 4 shows the organization of our baseline GPU architecture. Computation on GPU consists of functions or kernels that have a hierarchy of threads. Following the CUDA terminology, this hierarchy comprises a grid of thread blocks (or NDRange of work-groups in OpenCL terminology). Each thread block contains one or more warps (sub-groups in OpenCL), which is a hardware specific grouping of scalar threads (work-items in OpenCL) that execute in a lock step manner.

Memory space in CUDA follows this thread hierarchy and is partitioned with different levels of sharing among the threads. CUDA provisions for per thread *local* memory space which is private to a thread. Threads within a block cooperate by sharing data using *shared* memory space. Finally, CUDA also provisions for *global* and read-only *constant* and *texture* memory spaces which are visible to all threads in a kernel. While texture memory space is specific to CUDA, local, shared, global and constant spaces are termed private, local, global and constant in OpenCL respectively.

A GPU includes a small number of streaming multiprocessors (SM). Each SM has a large number of in-order compute cores, multiple load/store units and special function units. An SM executes instructions from a thread in the program order. A memory operation is

Table 1: Synchronization functions in GPGPU programming models. Notations: S: synchronization, R: read, W: write

(a) **OpenCL**

Synchronization operations (S)	Memory ordering parameter	Ordering enforced	Visibility scope
Fence, Atomic Operation	relaxed	—	Work-group, Device, System
	acquire	$S \rightarrow R$; $S \rightarrow W$	
	release	$W \rightarrow S$; $R \rightarrow S$	
	acq_rel	$S \rightarrow R$; $S \rightarrow W$ $R \rightarrow S$; $W \rightarrow S$	
	seq_cst	$S \rightarrow R$; $S \rightarrow W$ $R \rightarrow S$; $W \rightarrow S$ (& total order on accesses)	

(b) **CUDA**

Fence	—	$R \rightarrow R$ (code: $R;S;R$) $W \rightarrow W$ (code: $W;S;W$)	Thread block, Device, System
Atomic function	—	—	Device

issued to L1 cache when it reaches at the head of the memory pipeline and is then removed from the pipeline. Thus, a cache miss does not block the memory pipeline and subsequent memory operations can be issued to L1 cache as they reach at the head of the memory pipeline.

Each SM houses shared memory, private L1 data cache and constant cache. All SMs on a GPU share a L2 data cache. Current GPUs (NVIDIA Fermi [14], AMD GCN [15]) do not support coherent L1 data caches. Whereas, ARM's Mali GPU [16] supports coherent L1 caches. Lack of hardware support for coherence burdens the programmer to explicitly manage coherency of shared data (adding volatile keyword in CUDA). In our subsequent discussion we assume that our baseline GPU implements a cache coherent memory hierarchy.

2.2 CUDA and OpenCL Memory Consistency Models

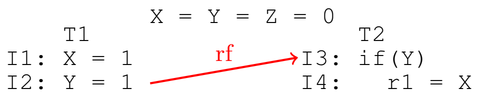
GPGPU programming models such as CUDA and OpenCL support weak consistency models based on data-race-free-0 (DRF-0) [6] memory model. These memory models match weakly ordered [17] memory models supported by GPU hardware. Under DRF-0, a SC execution is guaranteed only if program is data-race free. Concurrent accesses to same location by multiple threads such that at least one of these accesses is a write is termed as a data race. A program is data-race free if all data-races in the program are annotated as synchronization operations. In DRF-0 memory model two memory accesses can be ordered using synchronization operations. Current GPGPU programming models support two kind of synchronization operations: fences and atomic operations. These synchronization operations can be used to construct high level synchronization primitives.

Table 1 lists the synchronization operations provided under OpenCL [4] and CUDA programming models [1] (Section B.5, B.6 and B.12). A synchronization operation in OpenCL has a argument "Memory ordering parameter" which decides the memory ordering constraints enforced by the synchronization operation. The "Ordering enforced" column lists memory ordering constraints enforced by a synchronization operation for a given memory ordering parameter. There are four possible orderings for loads and stores with respect to a synchronization operation (S): $S \rightarrow R$, $S \rightarrow W$, $R \rightarrow S$ and $W \rightarrow S$. The ordering $S \rightarrow W$ indicates that later stores (in the program order) cannot complete before an earlier synchronization operation. In OpenCL, default memory ordering constraint for a synchronization operation is *memory_order_seq_cst* which guarantees that memory accesses before and after the synchronization operation are not reordered with it and there exists a total order on all accesses with this constraint. To support weaker ordering constraints for low level atomic operations, OpenCL allows programmers to specify relaxed constraints such as acquire, release and acq_rel. Note that acq_rel provides read and write ordering relative to the variable, whereas seq_cst provides read and write ordering globally.

CUDA programming model also supports fences and atomic functions to facilitate synchronization among threads, but the memory ordering constraints associated with them are very different from OpenCL. A fence in CUDA only enforces $R \rightarrow R$ and $W \rightarrow W$ memory ordering when these two reads or writes are separated by the fence. This constraint is different from either acquire or release constraints. Furthermore, atomic functions (read-modify-write operations) in CUDA do not have any memory ordering constraints associated with them.

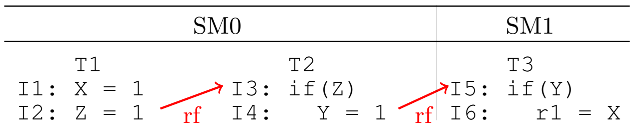
The last column in Table 1 shows different visibility scopes that can be associated with synchronization operations. The visibility scope of a synchronization operation governs which threads observe its effects. Effects of a synchronization operation with device visibility scope are visible to all threads running on a GPU device. On the other hand, the effects of a system scope synchronization operation are visible to threads in the current GPU device, host CPU and other compute devices present in the system. Finally, thread block/work-group scope limits the effects of a synchronization operation to a thread block/work-group of threads only.

Achieving SC in OpenCL and CUDA: In order to guarantee SC, OpenCL requires that all data-races be marked as synchronization operations and recommends that programmer use *memory_order_seq_cst* and system visibility scope for synchronization operations. While achieving SC with relaxed ordering constraints or smaller visibility scope is possible, such optimizations require significant care on part of the programmer and are highly prone to errors. On the other hand, the fence primitive alone as supported in CUDA is not sufficient to build high-level synchronization operation because



```
           X = Y = Z = 0
     T1                  T2
I1: X = 1           I3: if(Y)
I2: Y = 1           I4:   r1 = X
```

(a) Direct synchronization



```
                SM0                       |        SM1
      T1                 T2               |      T3
I1: X = 1          I3: if(Z)              | I5: if(Y)
I2: Z = 1          I4:   Y = 1            | I6:   r1 = X
```

(b) Indirect synchronization

Figure 1: Direct and indirect synchronization. rf is *reads-from* relation that associates a read with a unique write that supplies the value. Locations X and Y are in global memory and Z is in shared memory.

it does not enforce S → R (for acquire) or R → S (for release) ordering constraint which is required for protecting critical sections. To enforce these missing memory ordering constraint, an atomic operation (a read-modify-write operation) could be used before or after the fence instruction for acquire and release semantics respectively.

3. ENFORCING MEMORY ORDERING CONSTRAINTS IN A GPU

In this section we discuss mechanisms for enforcing memory model constraints in GPUs. First, we discuss when and how two memory accesses can appear to have executed in an order different from their program order. Then, we describe how different memory models can be supported in hardware by controlling how warps are executed in an SM. Finally, we also describe how a memory model may affect other architectural features like warp scheduling.

3.1 Violation of Ordering Constraints

In current GPUs, two memory instructions from a thread can complete out-of-order despite their in-order execution. This is possible due to several reasons: data caching at L1 cache, interconnect delivering requests in out-of-order manner, requests going to different partitions. However, in context of memory models, we are interested in how reordering of access can be observed by other threads.

Reordering of memory accesses in one thread could be visible to another thread running on different SM for following reasons:

Caching at L1 data cache: In a cache coherent system, a thread can observe reordering of memory accesses in a thread running on different SM. Figure 1a shows an example wherein two threads $T1$ and $T2$ are synchronizing with each other. Assume that $T1$ and $T2$ are running on streaming processors $SM0$ and $SM1$ which are caching Y and X respectively in their L1 cache. Even though $T1$ executes $I1$ and $I2$ in program order, $I2$ completes before $I1$ as $I1$ misses in L1 cache. The read in $I3$ *reads-from* $I2$ and returns 1. At this time, $T2$ is ready to execute $I4$, but there is no guarantee that $I1$ has been performed yet; $I4$ returns an old value of X. Thus, $T2$ has observed the out-of-order completion of access in $T1$.

Reordering in interconnect: The shared L2 cache in a GPU is typically partitioned into multiple banks. If GPU interconnect delivers two requests from different SMs, but to same L2 bank in an order different from their issue order, reordering of memory access could be visible across different SMs. Continuing with our earlier example, assume that X and Y map to different L2 banks and are not cached in any L1 cache. $T1$ executes two stores which are sent to their L2 banks after missing in L1 cache. In meantime, $T2$ executes $I3$ and issues a read request to L2 cache. Assume that $I3$ *reads-from* $I2$. After the branch in $I3$ has been resolved, $T2$ executes $I4$ and issues a read miss request to L2 cache. If the read request for X reaches the L2 bank containing X before the write request for X (possibly due to different latency to reach the L2 bank), $T2$ will end up with an old value of X, exposing out-of-order completion of accesses in $T1$.

Current NVIDIA and AMD GPUs [18, 15] employ a crossbar as their interconnect. A crossbar interconnect does not reorder two requests going to the same destination. However, if future GPUs adopt a mesh or other interconnects, such reordering of cache requests would make out-of-order completion of memory accesses visible to other SMs. In our designs, we do not rely on such ordering guarantees from the interconnect as we explicitly keep track of pending accesses.

Multiple address spaces: Both CUDA and OpenCL provide partitioned address spaces to their programs. Current GPUs implement different pipelines for different address spaces and do not provide ordering across different address spaces. If two threads are synchronizing via accesses in different address spaces, they need to use appropriate fences to achieve correct synchronization. For example, Figure 1b shows an execution in which two threads $T1$ and $T3$, running on different SM synchronize indirectly via thread $T2$. Assume both $T1$ and $T2$ are running on same SM and synchronize through accesses to Z which is in *shared memory*. $T2$ and $T3$ synchronize through accesses to Y in global memory space. Again it is possible for $I6$ to read an old value of X in this program. Under SC, $I1$ happens-before $I6$ due to transitivity of happens-before relation. Under weaker memory models, this example contains a heterogeneous race [7] and requires either fences with device or larger scope or sequentially consistent atomic accesses for Z and Y (in OpenCL).

3.2 Relaxed Memory Ordering

Relaxed memory models mandate memory ordering constraints for synchronization operations which then can be used to order execution of memory accesses. Furthermore, as described in Section 2.2, synchronization operations have concomitant visibility scope which mandates to which threads their effects are visible.

Thread block/work-group visibility scope is naturally supported in GPU hardware because all threads from a thread block are executed in one SM and reordering of memory accesses in a thread is not *observable* in other threads running on *same SM* because all threads share same path to memory hierarchy. While conveying synchronization operations with thread block visibility seem redundant from hardware perspective, they are used by GPU programmers to prevent compiler re-ordering of memory accesses.

Other visibility scopes such as device or system require that a synchronization operation's effect is made visible to all threads running on a device. This can be done by maintaining two per warp counters (for loads and stores) to keep track of pending accesses to global memory address space. Then, based on the memory ordering constraint desired by the synchronization operation, its issue can be stalled till relevant counters become zero. Delaying issue of a synchronization operation is preferable over allowing it to execute and then stalling the memory pipeline. Such delaying prevents this warp from interfering with other warps that share this memory pipeline. Similarly, the aforementioned counters are per warp to avoid a synchronization operation in a warp from interfering with operations of other warps.

To support indirect synchronization (Figure 1b), accesses to Z and Y needs be marked as synchronization operations with device scope. This will ensure that before a synchronizing shared space access is executed, preceding global accesses in its thread have been performed.

3.3 Total Store Order

Today, most common desktop and server processors support total store order(TSO) memory model which provisions for store buffer allowing loads to bypass preceding stores. The key insight behind TSO is that loads are more critical for performance and they should not be delayed by pending stores which are off the critical path. While for desktop and server workloads, TSO represents a good design point and we investigate its efficacy for GPUs.

The naïve approach to support TSO would be to leverage per warp counters to keep track of pending loads and stores. Based on the values of these counters, loads can be stalled at issue for preceding loads while stores can be stalled at issue for preceding loads and stores. Stalling at issue, while preferable, will cause loads following a delayed store to also get delayed. This unnecessarily constrains TSO.

For such situations, CPUs typically employ a store buffer that holds stores which are not yet ready to be issued to cache due to memory ordering constraints. This optimization allows processors to commit stores from reorder buffer (ROB) and allow later loads to get completed. Thus, to avoid stalling of load operations due to an earlier store, we can extend the naïve TSO design with a per warp store buffer.

In our evaluation, we find that TSO does not provide any significant performance advantage (Section 6) over SC. Our naïve SC design, which we describe next, matches the performance of TSO for most benchmarks. For rest of the benchmarks, both SC and TSO perform poorly in comparison to RMO. While TSO has small performance advantage over SC, we believe this advantage is often not large enough to warrant weakening of concurrency semantics.

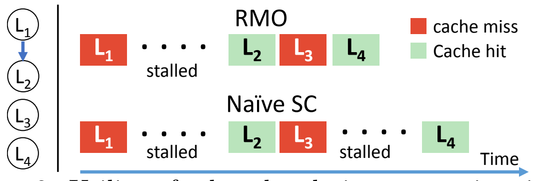


Figure 2: Utility of relaxed ordering constraints in an in-order GPU.

3.4 Sequential Consistency

While SC is the most constrained of all memory models we discuss, it provides concurrency semantics that naturally match with programmer's intuition. This boosts programmability which makes SC an attractive choice for CPUs [19].

SC requires that loads and stores appear to have completed in the program order. It could be trivially achieved by utilizing per warp counters for pending loads/stores to global memory space. To support indirect synchronization (Figure 1b), shared memory accesses wait for preceding global memory accesses to get completed before they can be executed. Though simple in nature, we find that this design performs very close to RMO for most of the benchmarks that we study (Section 6). The primary reason for effectiveness of this naïve design is availability of warp-level-parallelism (WLP) in GPGPU applications.

Unlike CPU applications, where typically only a very small number of threads are executing simultaneously in any core, GPGPU applications have a large number of warps concurrently executing in an SM. Due to large number of available warps, an SM is able to issue memory accesses from different warps to hide overhead of SC ordering constraints. Even though a single warp would observe high cost of SC ordering, an SM incurs only a small overhead.

Although higher WLP helps SC by having multiple warps to issue memory accesses from, it could also result in poorer SC performance. This is because increasing the number of threads per SM typically reduces available per thread cache capacity and this could result in higher cache miss rates. While high cache miss rates will affect performance of both SC and RMO, if the application has intra-thread MLP, RMO can exploit that by overlapping cache misses thus hiding their latency. However, even RMO is constrained in completely exploiting intra-thread MLP due to in-order execution in GPU which stalls an instruction if it has unsatisfied data dependencies.

Figure 2 shows an example that depicts this scenario. In this example three independent loads L_1, L_3, and L_4 can be issued in parallel by an out-of-order processor, but L_2 cannot be issued before L_1 completes due to its true dependency on L_1. Due to in-order execution, loads L_3 and L_4 also get stalled until L_2 is executed. Therefore, even RMO may not able to exploit all intra-thread MLP available in a program.

However in absence of such data dependency induced stalls, RMO can issue multiple memory requests, whereas SC can issue just one request per thread. This inadequacy of SC is exhibited as huge performance overhead (up to 3x) for SC in some of our experiments.

3.5 Common Memory Ordering Optimizations

In-window speculation [9] is commonly employed in CPUs to reduce TSO and SC overhead, but it is not suitable for GPUs as execution window contains only small number memory instructions from a warp.

Prefetching [9] is another common optimization to reduce the cost of memory ordering overhead. While a memory access is waiting for a memory ordering constraint to be resolved, its cache block can be prefetched. We can support this optimization in GPUs by extending our naïve SC design with a per warp FIFO buffer to hold memory accesses while they wait for their ordering constraints to get resolved. Meanwhile, an SM can issue prefetch requests for these waiting memory accesses to lower the miss latency. As we will see in Section 6, this does not help much in bringing down the overhead of SC in GPUs because an SM switches frequently between different warps, and many prefetched blocks end up getting evicted before they are used.

3.6 Impact of GPU Architectural Features

We have discussed how threads issue individual memory operations under different memory models. Here we discuss how other common GPU architectural features impact the cost of memory ordering constraints.

Warp Scheduling: A warp scheduling policy's performance is greatly affected by memory model because the memory model determines when a thread can execute memory operations. This interplay of thread scheduling and memory model is largely absent from CPU memory model tradeoffs because each core has only small number of concurrent threads, often one or two.

The simplest warp scheduling policies include round-robin and greedy-then-oldest (GTO). Under GTO, a warp is run until it stalls and then the oldest ready warp is selected to execute next. Current GPUs use warp scheduling policy similar to GTO [20]. Recent proposals [21, 22, 23, 24, 25] improve upon these simple scheduling policies by leveraging intra-thread MLP and weak consistency model of GPUs. Stronger memory model implementations (e.g. naïve SC or TSO) could reduce efficacy of these proposals by restricting threads from exploiting intra-thread MLP. Nevertheless, optimizations that allow threads to expose intra-thread MLP while preserving SC could regain most of the benefits of these proposals. Section 4 describes one such implementation of SC in GPU.

Cache-Write Policy: Current GPUs typically include write-through L1 caches with no-write allocate policy. To understand how memory ordering constraints are affected by cache write policy, we evaluate performance of different memory models on a cache coherent GPU system with write-through L1 caches that use no-write allocate policy. Performance impact of cache write policy depends on store locality present in the program. If a large fraction of stores are streaming (i.e., no reuse, e.g., Figure 3a), no-write allocate policy results in performance improvement due to an increase in effective cache capacity for loads. However, if a majority of stores are non-streaming (e.g., Figure 3b), sending all writes to L2 cache will increase pressure on memory bandwidth and could result in a performance loss.

```
for (each edge e of a graph node)
 if (!g_graph_visited)
   g_cost[e] = g_cost[tid] + 1;
```

(a) Streaming stores in bfs.

```
for (k=0; k < NK; k++) // NK = 512
  c[tid] += ALPHA * a [i*NK+k] * b[k*NJ+j];
```

(b) Non-streaming stores in gemm

Figure 3: Example of streaming and non-streaming stores in GPGPU programs

Performance gap between RMO and SC is expected to be higher for write-through caches [26], but our evaluation of GPGPU programs shows evidence contrary to this expectation (Section 6.5).

4. EFFICIENT SC FOR GPU

This section describes an optimized non-speculative solution to support SC in GPUs while achieving performance close to RMO. First, it provides an overview of key ideas behind this solution, then discusses implementation details.

4.1 Overview

To improve performance of the naïve SC design, we exploit the observation that memory ordering constraints can be relaxed for accesses to locations that are either private to a thread or shared with other threads in read-only fashion [10]. Memory reordering of such accesses cannot be observed or altered by other threads. We refer to accesses with this property as safe accesses and rest as unsafe in the following discussion.

Safe accesses can be freely reordered with other accesses provided intra-thread data dependencies are preserved. Unsafe accesses, however, are completed in the program order to ensure SC. To ensure correctness, accesses must be classified as either safe or unsafe in their program order.

We can identify safe accesses by observing thread-level data sharing at runtime. First, we classify memory locations at word granularity as safe or unsafe; then accesses are classified as safe if they access safe locations [11]. GPGPU applications typically exhibit high spatial locality [12] in memory accesses. We exploit this observation by keeping track of sharing of memory locations at a large granularity (referred to as sectors), thereby saving required hardware space.

Recording sharing information at thread granularity is prohibitively expensive in GPUs because of huge number of concurrent threads in GPUs. To keep hardware cost under control, we record this information for each SM instead of each thread. This gives us a set of potentially safe accesses (referred to as SM-safe). Similarly, shared memory accesses can also be treated as SM-safe accesses. These accesses are only potentially safe because multiple threads from same SM could be performing conflicting accesses to an SM-safe location. We treat SM-safe accesses as "safe" accesses and relax ordering constraints for them, but enforce strong ordering on conflicting accesses to an SM-safe location.

Classification of memory accesses: In CPUs, accesses are classified during virtual to physical address translation [27, 28, 11] by leveraging page tables and translation lookaside buffer (TLB). Recent GPU virtual memory proposals [12, 29] also provide a CPU like virtual memory support for GPUs. These proposals can be extended to also provide classification of accesses during address translation stage.

Although current GPUs implement virtual memory support, there is very little public information about them [30, 31]. It is not clear whether address translation is carried out prior to L1 cache access or not. Intel and AMD GPUs have Input Output Memory Management Units (IOMMU) [32, 33, 34] equipped with their own page tables and TLBs, but these IOMMUs are placed in memory controller and are accessed only after a miss in L1 cache. In this paper we have assumed a baseline GPU that does not perform address translation prior to accessing L1 data cache. Therefore, extending TLB and page tables to support access classification is not very useful for such GPUs.

To enable access classification prior to accessing L1 cache, we add a memory access classifier (MAC) at each memory partition that keeps track of how sectors are shared among SMs. These classifiers are statically partitioned; all requests for a given address go to a single classifier. A sector is said to be unsafe if it is being accessed by multiple SMs and at least one of them is writing to it. Sector level classification is also cached at each SM in a Type-cache to quickly determine SM-safe or unsafe nature of memory accesses.

Preserving SC during a sector's transition to unsafe state: A sector starts in one of safe states and may eventually transition to the unsafe state (labeled as $\langle shared, rw\rangle$, Figure 5). During transition to unsafe state, a sector cannot immediately complete the transition because there could still be pending unsafe accesses that precede already completed safe accesses to this sector in program order. In order to satisfy SC ordering constraints, these unsafe accesses must be completed before transition to the unsafe state is completed. During this transition, all new requests to this sector are delayed till the sector has completed its state transition.

Preserving SC for SM-safe accesses: In SM level classification of accesses, it is possible that an SM-safe location is not "safe" at thread level. Therefore, freely reordering SM-safe accesses could result in an SC violation if multiple threads are performing conflicting accesses to SM-safe locations. Consider our earlier example from Figure 1b and assume that all locations are in global memory with X, Y as unsafe and Z as SM-safe. Since accesses to Z can get completed before preceding unsafe accesses, an SC violation occurs when $T3$ receives old value of X after reading latest value of Z because unsafe accesses from different threads get completed independently. This violation also occurs when Z is in shared memory because accesses to global and shared memory are not ordered with each other.

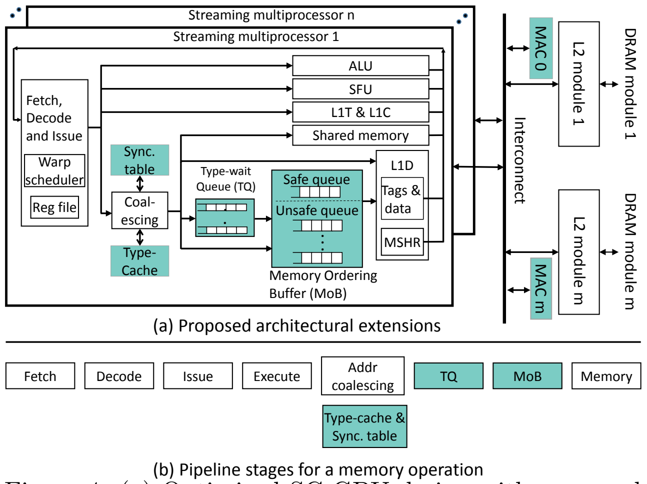


Figure 4: (a) Optimized SC GPU design with proposed changes highlighted in blue, (b) Pipeline stages for memory operations in proposed design.

To guarantee SC execution, we need to identify when threads are performing conflicting accesses to an SM-safe location and delay accesses such that preceding unsafe accesses have been completed. In our previous example, we delay the read of Z in $T2$ till unsafe accesses in $T1$ prior to $I2$ (previous access of Z) are completed. To this end, we envision a small table (referred to as *synchronization table*) that keeps track of SM-safe accesses that have bypassed earlier unsafe accesses from same warp. This table is used to delay later conflicting accesses from getting performed until it is safe to do so.

Synchronization table only keeps track of SM-safe accesses that have gotten reordered with preceding unsafe accesses. We do not need to consider earlier SM-safe accesses (e.g. if X is classified as SM-safe) because in the proposed design all threads have same path to memory hierarchy (either to L1 cache or to shared memory banks). Due to this common path, threads running on an SM cannot observe reordering of two SM-safe accesses from another thread running on the same SM. Furthermore, if a thread from another SM attempts to access a SM-safe location, it will trigger state transition for the sector; resulting in completion of pending SM-safe accesses to this sector, and thereby preserving SC.

4.2 Implementation

Figure 4a shows proposed extensions to a baseline GPU architecture. These changes include addition of Type-cache and memory access classifier (MAC) for classifying memory accesses, a synchronization table to ensure correct ordering for intra-SM conflicting accesses to an SM-safe location, and a memory ordering buffer to guarantee in-order commit of unsafe accesses. Fig-

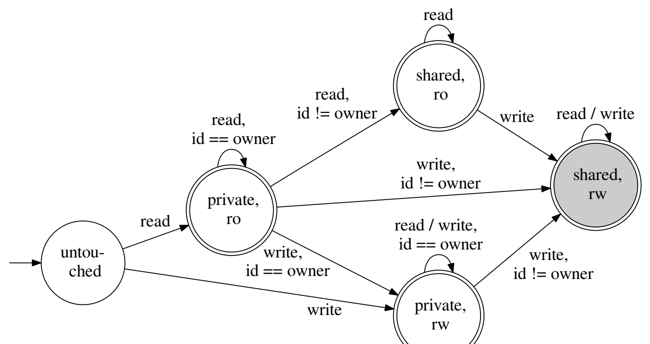


Figure 5: State transition diagram of sectors in MAC

ure 4b highlights how memory pipeline is extended to accommodate different functionalities.

Memory Access Classification: As a first step, an SM needs to determine the type of a memory access prior to accessing L1 cache. Access type resolution cannot be carried out in parallel with the data cache access because this could lead to a SC violation. This can happen if the sector for this access needs to undergo a state transition. Such a transition has to happen before the cache access is done. Note that data cache is decoupled from access classification mechanism.

As mentioned earlier, we cannot rely on current GPU's virtual memory support for access classification. To facilitate access classification, we have extended each memory partition with a small memory access classifier (MAC). These classifiers keep track of SM level data sharing at sector granularity. Each entry in MAC stores identifier of previous accessor and state of the sector. An entry in MAC can be in one of following states: $\langle private, ro\rangle$, $\langle private, rw\rangle$, $\langle shared, ro\rangle$, and $\langle shared, rw\rangle$ (Figure 5). In $\langle shared, ro\rangle$ state, individual sharers are not tracked. Only the $\langle shared, rw\rangle$ state is treated as unsafe state. This classification scheme is similar to schemes in prior proposals [27, 28, 11] that classified accesses at page granularity.

Accessing MAC incurs huge latency as requests have to cross interconnect. To make obtaining classification faster, we also cache a sector's information in a Type-cache in each SM. A Type-cache is analogous to TLB in CPUs. Besides a sector's classification, an entry in Type-cache also records an SM's write eligibility for this sector. If an SM does not have write privilege for a sector, it needs to send an request to MAC asking the same. These Type-caches are kept coherent through explicit messages on a sector's safe to unsafe state transition (*invalidation* message) and its eviction from MAC (*recall* message).

After a memory instruction is executed, addresses from different threads in a warp are coalesced before being sent to L1 cache to minimize L1 cache accesses. In parallel with address coalescing, an SM also accesses its Type-cache with address from a thread in the warp (Figure 4(b)). In the common case all threads are accessing only one cache block, their access type is already resolved without any additional delay. If multiple cache accesses are required for a memory instruction, their type resolution is also pipelined in the same way these accesses get sent to L1 cache. If an access misses in Type-cache, the SM sends a request to MAC to obtain the classification of the accessed sector.

On a miss in Type-cache, an access is moved to a type-wait queue (TQ stage) instead of blocking the pipeline. Type-wait queue is implemented as a per warp FIFO queue to avoid interference among different warps. FIFO queues ensure that accesses get their classification in program order. If a warp's type-wait queue is not empty, accesses from this warp are also sent to the type-wait queue even if they hit in Type-cache.

Ordering constraints for unsafe accesses: Once an access has been classified as SM-safe or unsafe, it can proceed with L1 cache access. Unsafe accesses are completed in their program order to guarantee SC. If an unsafe access cannot get issued to data cache because of ordering constraints, it is moved to unsafe FIFO queue for its warp in memory ordering buffer (MoB) to avoid pipeline stall (MoB pipeline stage). Once an access reaches at the head of its queue, it can be issued to cache.

Once an unsafe access has been issued to L1 cache, its entry is immediately reclaimed. To keep track of pending unsafe accesses, we can simply use per warp counters instead of keeping issued entries in unsafe queues. If a warp instruction accesses multiple cache blocks, these accesses can be issued to L1 cache in parallel because there is no ordering requirement on accesses from different threads running in a lock-step manner. However, unsafe accesses from later instructions are not allowed to access L1 cache till all preceding unsafe accesses have been completed.

Ordering constraints for SM-safe accesses: Ordering constraints for SM-safe accesses are relaxed except for conflicting accesses to an SM-safe location. We use *synchronization table* to identify when SM-safe accesses require strong ordering constraints. An SM-safe access at the head of type-wait queue searches synchronization table for conflicting accesses. If a match is found, this access needs to wait until the matching entry is deleted from the table. Otherwise, it can proceed with its L1 cache access. If an access is delayed due to a conflicting entry in the synchronization table, its warp is also restricted from issuing further memory requests till this wait is over.

Synchronization table is a small fully associative cache in each SM and supports a single writer or multiple readers for a cache block. An entry in this table has a pending count, access type (load or store) and warp identifier of previous accessor. An entry in the table is created when an SM-safe access bypasses pending unsafe accesses. An SM-safe access is said to have bypassed preceding unsafe accesses when it leaves the type-wait-queue and prior unsafe accesses in its warp are still pending. Furthermore, a marker entry is also inserted in the unsafe queue of this warp to determine when these unsafe accesses have been completed. This marker is deleted from the unsafe queue when all earlier pending unsafe accesses are completed. At this time, corresponding entry from synchronization table is also deleted because subsequent conflicting accesses do not need to wait anymore.

Synchronization table is also used to enforce ordering constraints for conflicting accesses in shared memory because shared memory accesses are akin to SM-safe accesses. Similar to SM-safe accesses in global memory space, shared memory accesses also search this table for conflicting entry and are delayed accordingly if a match is found. In our baseline GPU, shared memory accesses do not go through address coalescing stage. However, in our proposed design we also send shared memory accesses through an address coalescing stage to get 128 byte level unique addresses which are then used to index in the synchronization table. If a shared memory access needs to wait for a conflicting entry to get deleted from the table, its warp is also prevented from issuing further memory accesses to preserve intra-thread data dependencies.

Unlike for SM-safe accesses in global memory, additional markers are not inserted in an unsafe queue for shared memory accesses to get apprised of completion of pending unsafe stores. This avoids filling up of the unsafe queue with marker entries and keeps it available for unsafe accesses. Instead, we record an identifier for the latest unsafe access from this warp. This identifier is unique across all type-wait queues and unsafe queue entries. Once this unsafe entry is completed, synchronization table entries with this identifier are deleted from the table. In the case of multiple readers, each entry records one identifier per warp that has accessed the location pointed by this entry.

MoB also includes a safe queue for SM-safe accesses that could not be issued to L1 cache in this cycle due to cache port unavailability. The cache port could be unavailable if an access from either memory pipeline or from an unsafe queue wins arbitration for cache port. In the proposed design the safe queue is modeled as a FIFO queue even though there is no ordering requirement. We opted FIFO buffer to keep complexity low for the safe queue.

A memory operation can bypass TQ or MoB pipeline stages, if it hits in the Type-cache and there are no preceding pending unsafe accesses in its warp.

Preserving SC execution state on transition from safe to unsafe: When MAC detects a transition to the $\langle shared, rw \rangle$ state, it sends an *invalidation* message to the SM that had previously accessed this sector (or a broadcast if the sector was in $\langle shared, ro \rangle$ state). On receiving an *invalidation* request from MAC, an SM updates its Type-cache entry and other entries to this sector in its type-wait queue. It also needs to complete outstanding SM-safe accesses in the safe queue and drain its unsafe queues to ensure that all unsafe accesses that were bypassed by an SM-safe access to the sector indicated in the invalidation message have been completed. Once these queues have been drained, the SM sends an acknowledgment to MAC.

Draining all unsafe queues in MoB naïvely could be fairly expensive as there can many pending unsafe accesses from different warps that are not required to be completed in response to an invalidation message. In order to efficiently determine unsafe accesses that need to be completed, we reuse markers that are inserted in unsafe queues for SM-safe accesses. On receiving an *invalidation* request, an SM searches for markers to this sector in its unsafe queues. The SM needs to drain unsafe queues with a matching marker before sending an acknowledgment.

After receiving all acknowledgments (multiple acknowledgments for $\langle shared, ro \rangle$ state), MAC completes the safe to unsafe transition. To avoid races during this transition, all subsequent requests to this sector undergoing safe to unsafe state transition are not processed till this transition is completed.

Table 2: Simulator Configuration

GPGPU-sim Core Model	
# GPU Cores	16
Core Config	48 Wavefronts/core, 32 threads/wavefront, 1.4Ghz, Pipeline width:32, #Reg: 32768 Scheduling: Loose Round Robin. Shared Mem.: 48KB
Ruby Memory Model	
L1 Private Data$	32KB, 128B line, 4-way assoc. 128 MSHRs
L2 Shared Bank	128KB, 8-way, 128B line, 128 MSHRs. Minimum Latency: 340 cycles, 700 MHz
# Mem. Partitions	8
Interconnect	1 Crossbar/Direction. Flit: 32bytes Clock: 700 MHz. BW: 32 (Bytes/Cycle). (175GB/s/Direction)
# Virtual Networks	MESI: 5
GDDR Clock	1400 Mhz
Memory Channel BW	8 (Bytes/Cycle) (175GB/s peak). Minimum Latency: 460 cycles
Type-aware SC parameters	
Type-cache	64 entries, 4 way associative
Memory access classifier (MAC)	512 entries, 8 way associative, sector size: 16384 Bytes
Type-wait queue	48 FIFO queues with 4 entry per queue
Memory ordering buffer (MoB)	Safe queue: 16 entry FIFO; Unsafe queues: 48 FIFO queues with 4 entry per queue
Outstanding Stores in queues	64
Synchronization Table	64 entries

5. EXPERIMENTAL METHODOLOGY

Simulation environment: We used GPGPU-sim [35] v3.2.1 to simulate the baseline NVIDIA Fermi architecture (GTX 480) and our proposed design for efficient SC. To simulate cache hierarchy of data caches we used Ruby memory systems from GEMS [36]. The baseline coherence protocol is MESI with write-allocate policy. The MESI coherence protocol we simulate is from gem5 [37]. We also evaluate a system configuration with write-through protocol with no-write allocate policy. This setup is based on the simulation infrastructure used in GPU-coherence [38] work.

Our baseline GPU system includes 16 multi-processors each with 32 KB of L1 data cache. The default warp scheduling policy is greedy-then-oldest-first ("GTO") present in GPGPU-sim as it performs better than loose round-robin ("LRR") policy. The optimized SC design uses GTO and gives higher priority to warps that do not have pending unsafe accesses.

We modeled a crossbar interconnect network using detailed fixed pipeline network model in Garnet [39]. Each SM is connected with the crossbar interconnect through private ports. We modeled minimum L2 and DRAM access latencies to be 340 and 460 cycles (in core cycles) respectively. These latencies are in accordance with latencies used in earlier work [38] and observed latencies on Fermi GPU via micro-benchmarks by Wong et al. [40]. Other relevant simulator parameters are listed in Table 2.

The proposed Type-cache and MAC are small caches and are connected through the interconnect. Messages for resolving an access' type can use the three existing virtual channels (request, forward, and reply) to achieve deadlock freedom.

Queues in type-wait queue and MoB hold both loads and stores. However, an store entry could be as large as a cache block size (128 bytes). To keep area requirements of these queues low, we limit the number of outstanding stores in these queues to 64 stores and keep the store data in a separate data array. This allows entries in queues to not provision for large store data. Since, there is no load-to-store forwarding, keeping store data in a separate buffer does not pose any additional challenge.

Benchmarks: To evaluate different SC/TSO designs, we studied applications from Rodinia [41] and Polybench [42] benchmarks suites, an GPU implementation of memcached [43]. Additionally, we also studied benchmarks used by Singh et al. [38] as these benchmarks exhibit a higher degree of inter thread block communication.

6. EXPERIMENTAL RESULTS

In this section we compare the performance of TSO and SC (both naïve and optimized) against the baseline RMO memory model. We also show the impact of write-policy, warp scheduling policy on observed overhead of a given memory model.

6.1 Comparison of naïve SC, naïve TSO and RMO

Figure 6 shows normalized execution time of naïve SC and TSO designs with respect to a baseline GPU that implements RMO memory model. This graph reveals that for a majority of applications, naïve SC or TSO designs perform as good as baseline. There are two main reasons for this low performance gap between RMO and SC: warp-level-parallelism (WLP) and in-order execution. WLP helps in reducing the overhead of memory ordering constraints by allowing an SM to continue execution with different warps. Furthermore, the in-order execution could also limit RMO's ability to exploit intra-thread MLP (Section 3.4).

Although a majority of applications exhibit small performance difference for stronger memory models, there are benchmarks that incur huge performance overheads (up to 1.84x under TSO and up to 2.93x under SC for `gemm`). The primary sources of this performance gap is RMO's ability to take advantage of intra-thread MLP to overcome high cache miss rates. If a program has fairly low cache miss rates or lacks intra-thread MLP, SC can perform on par with RMO. For example, if we restrict each SM to execute only one thread block, these two benchmarks exhibit low cache miss rates (about 3%) which result in much smaller performance gap between SC and RMO. When an SM is allowed to execute up to 8 thread blocks, cache miss rates increase drastically (about 35%) due to higher contention at L1 cache. Under high cache miss rates, RMO performs better by exploiting intra-thread MLP in addition to WLP. Whereas, naïve SC is limited to exploiting just WLP.

Intuitively we expect RMO to outperform SC because it places fewer restrictions on execution. However, this may not always be true for GPGPU programs because there are many threads that could cause contention in memory hierarchy. For example, naïve SC and TSO outperform RMO for `streamcluster` (Figure 6).

To understand why SC outperforms RMO, consider an execution of code from `streamcluster` (Figure 7), where two warps W_1 and W_2 execute three independent loads and one store. Assume that all these warps access two distinct cache blocks that are not present in L1 cache. The first load from W_1 misses in the L1 cache and allocates an entry in MSHR. In RMO baseline, the GTO warp scheduler executes the second load from W_1 that hits in MSHR and gets merged with the existing entry in MSHR. If each MSHR entry can merge only two memory requests, the third load from W_1 gets stalled because it cannot be merged into the existing MSHR entry, resulting in a stall in the memory pipeline. At this time, the SM cannot issue any new memory access until this stall is resolved even if it switches to other warps (e.g. W_2). Once the first cache miss satisfied, the SM can issue the third load and complete it. However, it runs into the same problem again when the third load from W_2 is not able to get merged into a existing MSHR entry.

In contrast, under SC the warp scheduler switches to warp W_2 after the load from W_1 has missed in the cache. However, after issuing one load from each warp, execution is stalled till these caches misses gets satisfied. In this case, the execution is stalled due to memory ordering constraints instead of resource contention. Once these loads have been completed, all subsequent loads accesses hit in the cache and get completed without any delay.

This example demonstrates how unrestricted issue of memory accesses can cause RMO to perform worse than SC (`streamcluster`). When we increase the limit on number of accesses that can be merged into an MSHR entry from 32 to 1024, RMO and SC have almost same performance. This is also the reason why our optimized Type-aware SC too performs worse than naïve SC.

6.2 Benefits of TSO over SC are Small

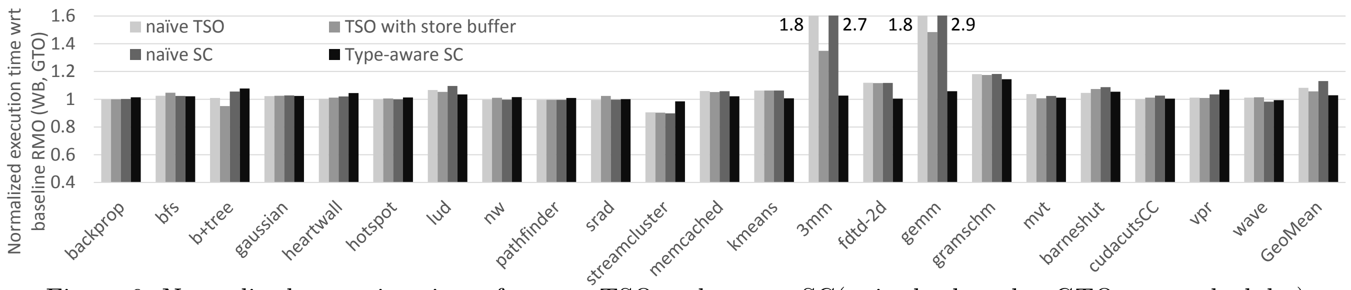


Figure 6: Normalized execution time of naïve TSO and naïve SC(write-back cache, GTO warp scheduler)

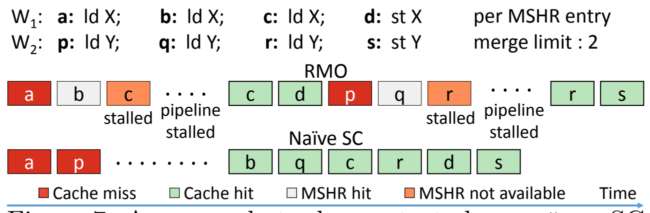

W1: a: ld X; b: ld X; c: ld X; d: st X
W2: p: ld Y; q: ld Y; r: ld Y; s: st Y
per MSHR entry merge limit : 2
RMO
stalled
pipeline stalled
Naïve SC
Cache miss
Cache hit
MSHR hit
MSHR not available
Time

Figure 7: An example to demonstrate how naïve SC could perform better than RMO

TSO enables store buffer optimization by allowing loads to bypass preceding stores. A store buffer can be organized either as an unified buffer or as a set per warp buffers. We find that unified store-buffer optimization performs significantly worse than per warp configuration due to contention from different warps. Furthermore, Figure 6 shows that adding a large per warp store buffer (48 queues with 8 entries each with 128 bytes data) improves naïve TSO's performance only marginally.

For a majority of benchmarks both naïve SC and naïve TSO perform as good as RMO. Though TSO with store buffer performs better than naïve SC for other benchmarks, its overhead is still quite high in comparison to RMO. Marginal performance benefits of TSO for a few programs does not justify weakening SC guarantees.

6.3 Prefetching is Ineffective

In Section 3.5 we described an extension to naïve SC to take advantage of prefetching for accesses that cannot be issued to cache due to ordering constraints. Figure 8 shows the result for such a design. We find that prefetching helps very little in improving SC's performance. Whereas, some benchmarks (e.g., `streamcluster`, `fdtd-2d`) incur a performance loss due to prefetches being premature.

In CPUs, prefetches are used with in-window speculation to reduce memory ordering overhead. In window-speculation allows loads to consume prefetched data even if ordering constraints are not satisfied yet. However, in GPUs loads must still wait for ordering constraints to be satisfied due to lack of in-window speculation support. Furthermore, the probability of a prefetched data getting evicted before being read is also higher in GPUs due to large number of threads running on an SM.

Premature prefetches do not provide much benefit; at the same time they still consume limited resources like MSHR entries, space in L1 and L2 caches and bandwidth between L1 and L2 caches. Therefore, premature prefetches leads to poorer performance.

6.4 Type-aware design is Effective

Now, we evaluate performance of access type aware design presented in Section 4. Figure 6 shows normalized execution time of Type-aware SC with respect to a RMO baseline. From this graph, we find that Type-aware SC successfully brings down large naïve SC overheads in `3mm` and `gemm`. We also observe that the proposed design is not only able to reduce overheads of SC but on average it performs very close to RMO. The primary source of this performance improvement is relaxing ordering constraints for SM-safe accesses.

Although Type-aware SC typically improves performance over naïve SC, it could also end up hurting performance. For example, for `b+tree` benchmark Type-aware design performs worse than a naïve design. The primary reason for this slowdown is unsafe queues being too small. A full unsafe queue can lead to memory pipeline stall if type-wait queue also become full. Increasing the size of unsafe queues alleviates this problem and, Type-aware SC performs very close to RMO. In case of `streamcluster`, Type-aware SC also suffers from the same resource contention in MSHR that is present for baseline RMO.

Our sensitivity studies (not shown in the paper) show that increasing the size of type wait queues and unsafe queues to more than four entries per warp provides very little performance improvement. Similarly, MAC and Type-cache also benefit very little from larger size due to bigger sectors and small shared L2 cache. The shared L2 cache in GPUs is small (less than 1 MB) in comparison to CPUs with even small core count. A smaller L2 cache results in a smaller set of locations that need to be tracked in MAC at any time.

6.5 Impact of cache-write policy

Current GPUs support write-through L1 data caches. To understand how a cache write policy affects the cost of memory ordering constraints, we evaluate performance of different memory models on a cache coherent GPU system with write-through and write no-allocate L1 caches.

Figure 9 shows performance impact of tradeoffs for write-through cache policy which are described earlier in Section 3.6. All execution times have been normalized to a system with RMO memory model and write-back caches. For the baseline RMO memory model, applications with streaming stores enjoy performance boost due to increased effective cache capacity for loads. However, some applications with non-streaming stores end

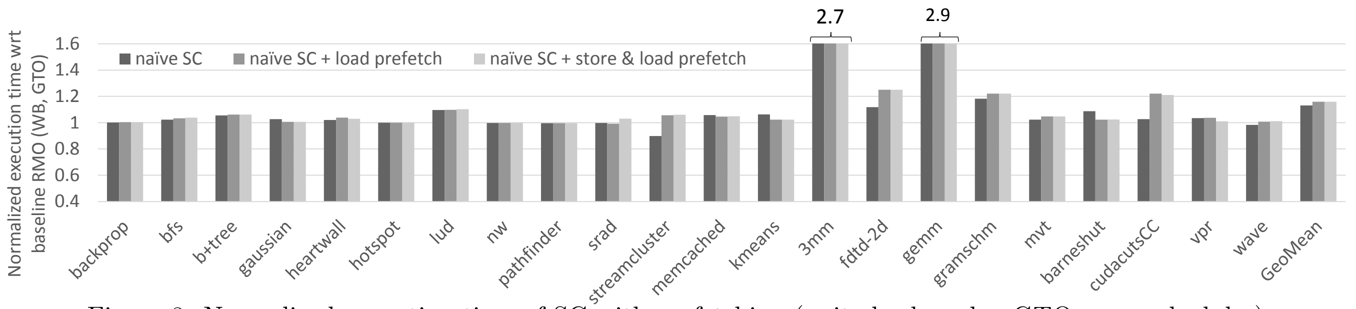


Figure 8: Normalized execution time of SC with prefetching (write-back cache, GTO warp scheduler)

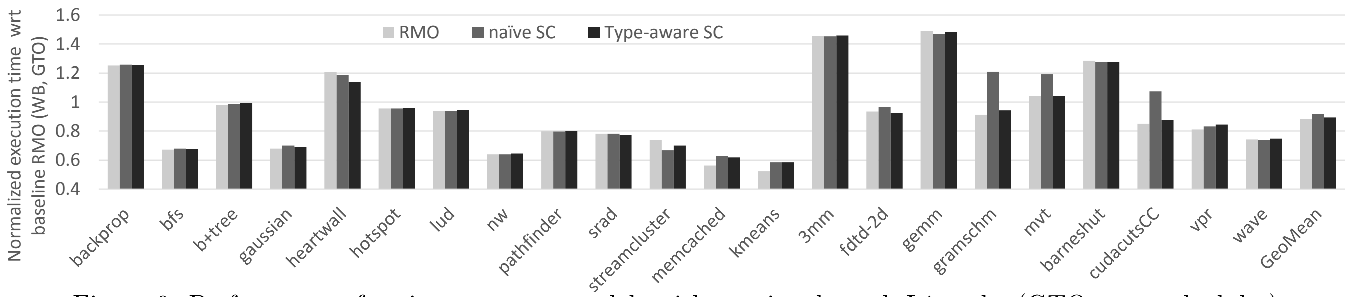


Figure 9: Performance of various memory models with a write-through L1 cache (GTO warp scheduler)

up loosing performance due to increased contention in memory hierarchy (e.g., `gemm`).

The write-no allocate policy increases effective cache capacity because stores do not take any space in the cache. Both the baseline RMO and the naïve SC benefit from this increase in cache capacity and incur fewer load misses in comparison to write-back caches.

Intuitively, we expect performance gap between RMO and SC to be wider in write-through caches than the gap in write-back caches because stores have larger latency in write-through caches [26]. However, Figure 9 shows that this gap becomes narrower for write-through caches, e.g. naïve SC performs very close to RMO for `gemm` and `heartwall` benchmarks. Whereas, for write-back L1 cache, naïve SC has much higher overhead for these applications (Figure 6).

In applications with non-streaming stores, naïve SC still issues memory accesses in a restricted manner and typically experiences very small contention in memory hierarchy. Whereas, RMO could suffer from higher contention in memory hierarchy caused by additional stores being sent to L2 cache. This is indeed the case for `gemm` where RMO performs worse with write-through cache than with write-back cache. Due to these reasons there is very little difference in performance of RMO and SC for `gemm`. On the other hand, applications like `mvt` are hurt by higher store latency and exhibit larger difference between RMO and naïve SC under write-through caches.

For applications with streaming stores, sending stores to L2 cache on a write does not necessarily result in a performance loss because most of these stores will miss in L1 cache and get sent to L2 cache even with write-back L1 caches. On the other hand, both RMO and SC enjoy an increase in effective cache capacity. Therefore, relative performance difference between RMO and SC remains similar to that under write-back L1 caches.

7. RELATED WORK

Prior work has not studied mechanisms for enforcing memory ordering constraints in GPUs and their performance cost for various memory models. Also, to our knowledge this is the first work examining efficient solution for providing SC in GPUs.

For throughput oriented processors with in-order cores, Hechtman et al. [8] observe that thread-level parallelism can mask the stalls due to strict memory ordering constraints. While this observation is valid for many GPU applications, we report several applications with significant SC overhead. Also, we study the interplay between memory ordering constraints and features common in GPUs such as write-through caches and warp-scheduling policies. Furthermore, in contrast to MTTOP systems they study, presence of a shared memory in GPUs allows threads in a thread-block to synchronize efficiently without increasing memory contention. Their observations on increased frequency of synchronization operations and memory contention issues due to them are less applicable for GPGPU programs.

Finally, we observe the ineffectiveness of prefetching optimization in reducing SC overhead. We propose an alternative optimization for GPUs that exploit spatial locality and "GPU-core locality" (data private to a core).

Speculative optimizations for memory ordering constraints: In the past, there have been a gamut of speculative proposals [9, 44, 45, 46, 47, 48, 49, 50] which help bring down the cost of stronger memory models like SC for multi-cores. The key tenet of these schemes is to speculatively commit memory operations and buffer them locally. They then rely on a register checkpoint and rollback mechanism to trigger recovery if a misspeculation is detected.

Implementing such techniques for GPUs is expensive for many reasons. GPUs employ large register files and their scheduling unit is a group of threads or warp. They also employ a shared memory for high bandwidth and low latency access to input data which is shared amongst group of threads. Also, branches in GPUs require special handling using branch synchronization stack. These architectural differences make checkpointing processor state far more complicated in GPUs. The complexity is further compounded by the fact that GPUs have 1000s of more threads than CPUs which translates to considerable energy and area overheads to track both processor and speculative state. Finally, given their in-order pipeline, GPUs stand less to gain from speculative execution.

Non-speculative techniques for improving SC: Recently there have been a few proposals [51, 11, 52, 53] that reduce SC overhead without incurring cost of out-of-window speculation. These proposals, however, optimize only for stalls due to write misses in L1 cache. In contrast, our proposed design also optimizes for both loads and stores.

The proposed SC design is based on "End-to-end SC" work [11] which is targeted towards multi-core CPU systems. As a consequence, to design efficient SC for GPU's our proposed design has to factor in the micro architectural differences between CPU and GPU with regards to virtual memory support, number of concurrent threads per processor, shared memory & partitioned address space and the execution model. In Section 4.1, we have discussed how virtual memory support in current GPUs might require us to engineer Type-cache and memory access classifier in order to classify memory accesses. Furthermore, Section 4.2 describes how large number of threads and a shared memory per SM in GPUs present additional challenge in ensuring system's SC execution state during intra SM synchronization and during safe to unsafe state transition of a sector. Typically GPGPU applications exhibit more spatial locality than CPU applications. Therefore, classifying memory accesses at a larger granularity is more suitable for GPU benchmarks. In fact, our design uses a 16KB sector size that is much larger than page size classification used in [11]

Other non-speculative proposals Conflict-ordering [51], Atomic-SC [52] and zFence [53] leverage low communication latency for L2 cache to obtain pending write sets, mutexes, or coherence permissions respectively. However, given very high L1 miss rates for GPGPU applications and huge L2 access latency in GPUs, these proposals may not be very effective in reducing memory ordering overhead for GPUs. On the contrary, our proposed design incurs large latency cost only for communicating with memory access classifier (MAC) once per 16KB data. Coupled with spatial locality of accesses in GPUs, this translates to very small overhead for accessing MAC.

8. CONCLUSION

In this paper, we describe how stronger memory models (e.g. SC, TSO) can be implemented in GPUs. Despite GPUs executing instructions in order, they can violate memory ordering constraints by allowing out-of-order completion of memory accesses in a cache coherent system. We show that the inherent thread level parallelism prevalent in GPGPU programs helps close the performance gap between various memory models for many if not all benchmarks. We demonstrate using experimental evaluation, benchmarks which exhibit considerable performance overhead for stronger memory models and thus motivate the need to design optimizations which will help obviate this overhead. Furthermore, we also demonstrated that TSO memory model may not be an attractive alternative for GPUs as it does not provide significant benefits over SC. We evaluate efficacy of prefetching in bridging the performance gap of stronger memory models and find it to be insufficient. Finally, we adapt a non-speculative technique proposed for multi-cores to GPU architectures and show that it successfully eliminates the performance overhead for stronger memory models.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their comments which helped improve this paper. This work is supported by the National Science Foundation under the CAREER-1149773 award.

10. REFERENCES

[1] "NVIDIA CUDA v7.0 Developer Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] A. Munshi *et al.*, "The OpenCL specification," *Khronos OpenCL Working Group*, vol. 1, pp. l1–15, 2009.

[3] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "Gpu concurrency: Weak behaviours and programming assumptions," in *ASPLOS*, 2015, pp. 577–591.

[4] A. Munshi *et al.*, "The OpenCL specification," *Khronos OpenCL Working Group*, vol. 2.0, 2013.

[5] ISO/IEC 9899:2011, "Programming language C," http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853, 2011.

[6] S. V. Adve and M. D. Hill, "Weak Ordering—A New Definition," in *ISCA*, 1990.

[7] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *ASPLOS*, 2014.

[8] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," in *ISCA*, 2013, pp. 201–212.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *ICPP*, 1991, pp. 355–364.

[10] D. Shasha and M. Snir, "Efficient and Correct Execution of Parallel Programs that Share Memory," *ACM TOPLAS*, vol. 10, no. 2, pp. 282–312, Apr. 1988.

[11] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end Sequential Consistency," in *ISCA*, june 2012, pp. 524 –535.

[12] J. Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *HPCA*, 2014, pp. 568–578.

[13] "Online companion material." [Online]. Available: http://www.eecs.umich.edu/˜ansingh/micro15

[14] NVIDIA, " NVIDIA's next generation CUDA compute architecture: Fermi, White Paper," http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2009.

[15] AMD, "AMD Graphics Core Next," http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2011.

[16] S. Steele, "ARM GPUs: Now and in the Future," 2011. [Online]. Available: http://www.arm.com/files/event/8_steve_steele_arm_gpus_now_and_in_the_future.pdf

[17] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory access buffering in multiprocessors," in *ISCA*, 1986, pp. 434–442.

[18] NVIDIA, "Nvidia's next generation cuda compute architecture: Kepler gk110," *whitepaper*, 2012. [Online]. Available: www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[19] M. D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *IEEE Computer*, vol. 31, pp. 28–34, 1998.

[20] P. Mills, J. Lindholm, B. Coon, G. Tarolli, and J. Burgess, "Scheduler in multi-threaded processor prioritizing instructions passing qualification rule," May 24 2011, uS Patent 7,949,855. [Online]. Available: http://www.google.com/patents/US7949855

[21] T. Rogers, M. O'Connor, and T. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO*, 2012, pp. 72–83.

[22] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *MICRO*, 2013, pp. 99–110.

[23] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *ASPLOS*, 2013, pp. 395–406.

[24] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *HPCA*, 2014, pp. 272–283.

[25] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *HPCA*, 2015, pp. 174–185.

[26] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton, "An evaluation of memory consistency models for shared-memory systems with ilp processors," in *ASPLOS*, 1996, pp. 12–23.

[27] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *ISCA*, 2009, pp. 184–195.

[28] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks," in *ISCA*, 2011.

[29] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *ASPLOS*, 2014, pp. 743–758.

[30] R. Danilak, "System and method for hardware-based gpu paging to system memory," Nov. 24 2009, uS Patent 7,623,134. [Online]. Available: http://www.google.com/patents/US7623134

[31] P. Tong, S. Yeoh, K. Kranzusch, G. Lorensen, K. Woo, A. Kaul, C. Case, S. Gottschalk, and D. Ma, "Dedicated mechanism for page mapping in a gpu," Jan. 31 2008, uS Patent App. 11/689,485. [Online]. Available: http://www.google.com/patents/US20080028181

[32] Intel, "Intel Virtualization Technology for Directed I/O Architecture Specification," 2006.

[33] AMD, "AMD I/O Virtualization Technology (IOMMU) Specification," 2006.

[34] N. Amit, M. B. Yehuda, and B.-A. Yassour, "Strategies for mitigating the iotlb bottleneck," in *WIOSCA*, 2010.

[35] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009, pp. 163–174.

[36] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, pp. 92–99, 2005.

[37] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[38] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt, "Cache coherence for gpu architectures," in *HPCA*, 2013, pp. 578–590.

[39] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009, pp. 33–42.

[40] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS*, 2010, pp. 235–246.

[41] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.

[42] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–10.

[43] T. Hetherington, T. Rogers, L. Hsu, M. O'Connor, and T. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *ISPASS*, April 2012, pp. 88–98.

[44] P. Ranganathan, V. Pai, and S. Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models," in *ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 199–210.

[45] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP=RC?" in *ISCA*, 1999, pp. 162–171.

[46] C. Gniady and B. Falsafi, "Speculative Sequential Consistency with Little Custom Storage," in *PACT*, 2002.

[47] C. Blundell, M. Martin, and T. Wenisch, "InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors," in *ISCA*, 2009, pp. 233–244.

[48] T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors," in *ISCA*, 2007, pp. 266–277.

[49] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, 2007, pp. 278–289.

[50] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with Transactional Coherence and Consistency (TCC)," in *ASPLOS*, 2004, pp. 1–13.

[51] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient Sequential Consistency via Conflict Ordering," in *ASPLOS*, 2012, pp. 273–286.

[52] D. Gope and M. Lipasti, "Atomic sc for simple in-order processors," in *HPCA*, 2014, pp. 404–415.

[53] S. Aga, A. Singh, and S. Narayanasamy, "zfence: Data-less coherence for efficient fences," in *ICS*, 2015, pp. 295–305.