QuickRelease:
A Throughput-oriented Approach to Release Consistency on GPUs

Blake A. Hechtman[†§], Shuai Che[†], Derek R. Hower[†], Yingying Tian[†‡], Bradford M. Beckmann[†], Mark D. Hill[‡†], Steven K. Reinhardt[†], David A. Wood[‡†]

[†]Advanced Micro Devices, Inc.
blake.hechtman@duke.edu

[§]Duke University Electrical and Computer Engineering
{derek.hower, shuai.che, brad.beckmann, steve.reinhardt}@amd.com

[‡]University of Wisconsin-Madison Computer Sciences
{markhill, david}@cs.wisc.edu

[‡] Texas A&M University Computer Science and Engineering
yingyingtian@tamu.edu

Abstract

Graphics processing units (GPUs) have specialized throughput-oriented memory systems optimized for streaming writes with scratchpad memories to capture locality explicitly. Expanding the utility of GPUs beyond graphics encourages designs that simplify programming (e.g., using caches instead of scratchpads) and better support irregular applications with finer-grain synchronization. Our hypothesis is that, like CPUs, GPUs will benefit from caches and coherence, but that CPU-style "read for ownership" (RFO) coherence is inappropriate to maintain support for regular streaming workloads.

This paper proposes QuickRelease (QR), which improves on conventional GPU memory systems in two ways. First, QR uses a FIFO to enforce the partial order of writes so that synchronization operations can complete without frequent cache flushes. Thus, non-synchronizing threads in QR can re-use cached data even when other threads are performing synchronization. Second, QR partitions the resources required by reads and writes to reduce the penalty of writes on read performance.

Simulation results across a wide variety of general-purpose GPU workloads show that QR achieves a 7% average performance improvement compared to a conventional GPU memory system. Furthermore, for emerging workloads with finer-grain synchronization, QR achieves up to 42% performance improvement compared to a conventional GPU memory system without the scalability challenges of RFO coherence. To this end, QR provides a throughput-oriented solution to provide fine-grain synchronization on GPUs.

1. Introduction

Graphics processing units (GPUs) provide tremendous throughput with outstanding performance-to-power ratios on graphics and graphics-like workloads by specializing the GPU architecture for the characteristics of these workloads. In particular, GPU memory systems are optimized to stream through large data structures with coarse-grain and relatively infrequent synchronization. Because synchronization is rare, current systems implement memory fences with slow and inefficient mechanisms. However, in an effort to expand the reach of their products, vendors are pushing to make GPUs more general-purpose and accessible to programmers who are not experts in the graphics domain. A key component of that push is to simplify graphics memory with support for flat addressing, fine-grain synchronization, and coherence between CPU and GPU threads [1].

However, designers must be careful when altering graphics architectures to support new features. While more generality can help expand the reach of GPUs, that generality cannot be at the expense of throughput. Notably, this means that borrowing solutions from CPU designs, such as "read for ownership" (RFO) coherence, that optimize for latency and cache re-use likely will not lead to viable solutions [2]. Similarly, brute-force solutions, such as making all shared data non-cacheable, also are not likely to be viable because they severely limit throughput and efficiency.

Meanwhile, write-through (WT) GPU memory systems can provide higher throughput for streaming workloads, but those memory systems will not perform as well for general-purpose GPU (GPGPU) workloads that exhibit temporal locality [3]. An alternative design is to use a write-back or write-combining cache that keeps dirty blocks in cache for a longer period of time (e.g., until evicted by an LRU replacement policy). Write-combining caches are a hybrid between WT and write-back caches in which multiple writes can be combined before reaching memory. While these caches may accelerate workloads with temporal locality within a single wavefront (warp, 64 threads), they require significant overhead to manage synchronization among wavefronts simultaneously executing on the same compute unit (CU) and incur a penalty for performing synchronization. In particular, write-combining caches require finding and evicting all dirty data written by a given wavefront, presumably by performing a heavy-weight iteration over all cache blocks. This overhead discourages fine-grain synchronization that we predict will be necessary for broader

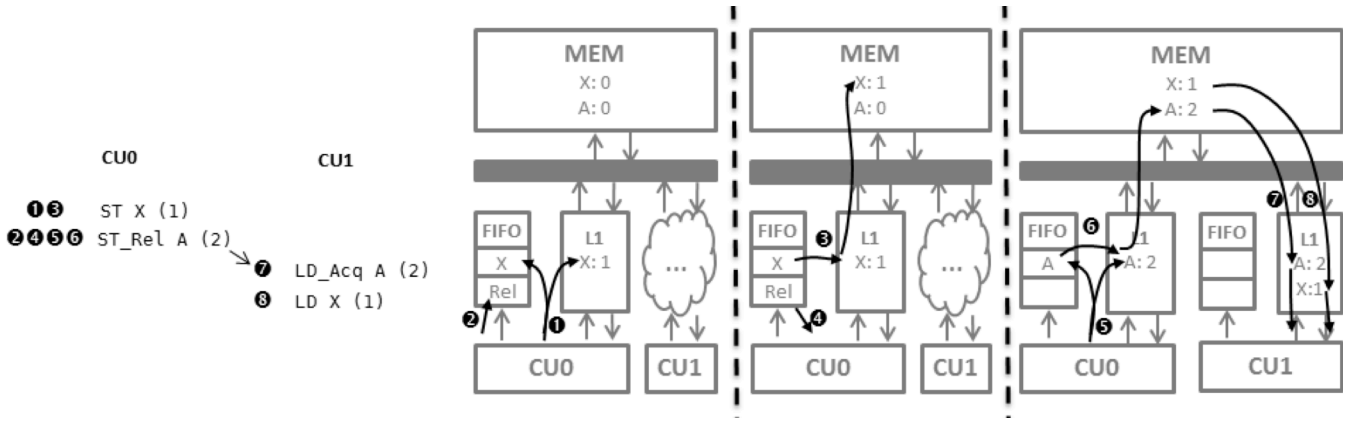

Figure 1. Example of QuickRelease in a simple one-level graphics memory system.

success of GPGPU compute. To this end, no current GPUs use write-combining caches for globally shared data (however, GPUs do use write-combining caches for graphic specific operations such as image, texture, and private writes).

In this paper, we propose a GPU cache architecture called QuickRelease (QR) that is designed for throughput-oriented, fine-grain synchronization without degrading GPU memory-streaming performance. In QR, we "wrap" conventional GPU write-combining caches with a write-tracking component called the synchronization FIFO (S-FIFO). The S-FIFO is a simple hardware FIFO that tracks writes that have not completed ahead of an ordered set of releases. With the S-FIFO, QR caches can maintain the correct partial order between writes and synchronization operations while avoiding unnecessary inter-wavefront interference cause by cache flushes.

When a store is written into a cache, the address also is enqueued onto the S-FIFO. When the address reaches the head of the S-FIFO, the cache is forced to evict the cache block if that address is still present in the write cache. With this organization, the system can implement a release synchronization operation by simply enqueueing a release marker onto the S-FIFO. When the marker reaches the head of the queue, the system can be sure that all prior stores have reached the next level of memory. Because the S-FIFO and cache are decoupled, the memory system can utilize aggressive write-combining caches that work well for graphics workloads.

Figure 1 shows an example of QR. In the example, we show two threads from different CUs (a.k.a. NVIDIA streaming multi-processors) communicating a value in a simple GPU system that contains one level of write-combining cache. When a thread performs a write, it writes the value into the write-combining cache and enqueues the address at the tail of the S-FIFO (time ❶). The cache block then is kept in the L1 until it is selected for eviction by the cache replacement policy or its corresponding entry in the FIFO is dequeued. The controller will dequeue an S-FIFO entry when the S-FIFO fills up or a synchronization event triggers an S-FIFO flush. In the example, the release semantic of a store/release operation causes the S-FIFO to flush. The system enqueues a special release marker into the S-FIFO (❷), starts generating cache evictions for addresses ahead of the marker (❸), and waits for that marker to reach the head of the queue (❹). Then the system can perform the store part of the store/release (❺), which, once it reaches memory, signals completion of the release to other threads (❻). Finally, another thread can perform a load/acquire to complete the synchronization (❼) and then load the updated value of X (❽).

An important feature of the QR design is that it can be extended easily to systems with multiple levels of write-combining cache by giving each level its own S-FIFO. In that case, a write is guaranteed to be ordered whenever it has been dequeued from the S-FIFO at the last level of write-combining memory. We discuss the details of such a multi-level system in Section 3.

Write-combining caches in general, including QR caches, typically incur a significant overhead for tracking the specific bytes that are dirty in a cache line. This tracking is required to merge simultaneous writes from different writers to different bytes of the same cache line. Most implementations use a dirty-byte bitmask for every cache line (12.5% overhead for 64-byte cache lines) and write out only the dirty portions of a block on evictions.

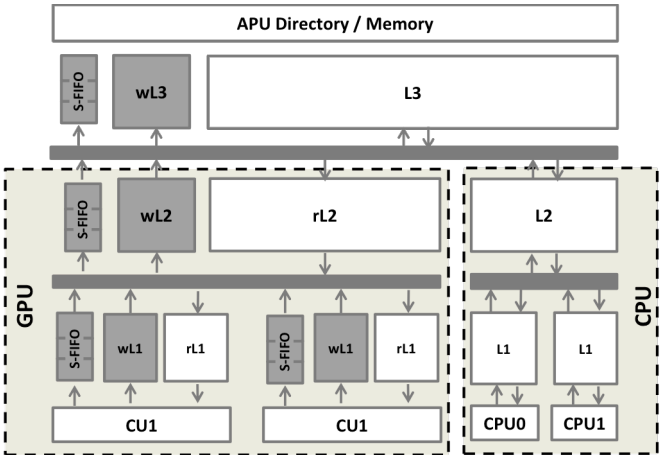

Figure 2: Baseline accelerated processing unit system. QR-specific parts are all S-FIFOs, wL1s, wL2, and wL3 (all smaller than rL1, rL2 and L3).

To reduce the overhead of byte-level write tracking, QR separates the read and write data paths and splits a cache into read-only and (smaller) write-only sub-caches. This separation is not required, but allows an implementation to reduce the overhead of writes by providing dirty bitmasks only on the write-only cache. The separation also encourages data path optimizations like independent and lazy management of write bandwidth while minimizing implementation complexity. We show that because GPU threads, unlike CPU threads, rarely perform read-after-write operations, the potential penalty of the separation is low [4]. In fact, this separation leads to less cache pollution with write-only data.

Experimental comparisons to a traditional GPGPU throughput-oriented WT memory system and to an RFO memory system demonstrate that QR achieves the best qualities of each design. Compared to the traditional GPGPU memory system, bandwidth to the memory controller was reduced by an average of 52% and the same applications ran 7% faster on average. Further, we show that future applications with frequent synchronization can run integer factors faster than a traditional GPGPU memory system. In addition, QR does not harm the performance of current streaming applications while reducing the memory traffic by 3% compared to a WT memory system. Compared to the RFO memory system, QR performs 20% faster. In fact, the RFO memory system generally performs worse than a system with the L1 cache disabled.

In summary, this paper makes the following contributions:

- We augment an aggressive, high-throughput, write-combining cache design with precise write tracking to make synchronization faster and cheaper without the need for L1 miss status handling registers (MSHRs).
- We implement write tracking efficiently using S-FIFOs that do not require expensive CAMs or cache walks, which prevent inter-wavefront synchronization interference due to cache walks.
- Because writes require an additional byte mask in a write-combining cache, we optionally separate the read and write data paths to decrease state storage.

In this paper, Section 2 describes current GPGPU memory systems and prior work in the area of GPGPU synchronization. Section 3 describes QR by describing its design choices and how it performs memory operations and synchronization. Section 4 describes the simulation environment for our experiments and the workloads we used. Section 5 evaluates the merits of QR compared to both a traditional GPU memory system and a theoretical MOESI coherence protocol implemented on a GPGPU.

2. Background and Related Work

This section introduces the GPU system terminology used throughout the paper and describes how current GPU memory systems support global synchronization. Then we introduce release consistency (RC), the basis for the memory model assumed in the next sub-section and the model being adopted by the Heterogeneous System Architecture (HSA) specification, which will govern designs from AMD, ARM, Samsung, and Qualcomm, among others. We also describe the memory systems of two accelerated processing units (APUs—devices containing a CPU, GPU, and potentially other accelerators) that obey the HSA memory model for comparison to QR: a baseline WT memory system representing today's GPUs, and an RFO cache-coherent memory system, as typically used by CPUs, extended to a GPU. Finally, in Section 2.5, we discuss how QR compares to prior art.

2.1. GPU Terminology

The paper uses AMD and OpenCL™ terminology [5] to describe GPU hardware and GPGPU software components. The NVIDIA terminology [6] is in parentheses.

- **Work-item** (thread): a single lane of GPU execution.
- **Wavefront** (warp): 64 work-items executing a single instruction in lock-step over four cycles on a 16-wide SIMD unit with the ability to mask execution based on divergent control flow. This now is known as a sub-group in OpenCL 2.0.
- **Compute unit** (streaming multi-processor): a cluster of four SIMD units that share a L1 cache and multiplexes execution among 40 total wavefronts.
- **Work-group** (thread block): a group of work-items that must be scheduled to a single CU.
- **NDRange** (grid): a set of work-groups.
- **Kernel**: a launched task including all work-items in an NDRange.
- **Barrier**: an instruction that ensures all work-items in a work-group have executed it and that all prior memory operations are visible globally before it completes.
- **LdAcq**: Load acquire, a synchronizing load instruction that acts as downward memory fence such that later operations (in program order) cannot become visible before this operation.
- **StRel**: Store release, a synchronizing store instruction that acts like an upward memory fence such that all prior memory operations (in program order) are visible before this store.

2.2. Current GPU Global Synchronization

Global synchronization support in today's GPUs is relatively simple compared to CPUs to minimize microarchitecture complexity and because synchronization primitives currently are invoked infrequently. Figure 2 illustrates a GPU memory system loosely based on current architectures, such as NVIDIA's Kepler [7] or AMD's Southern Islands [8], [9]. Each CU has a WT L1 cache and all CUs share a single L2 cache. Current GPU memory models only require stores to be visible globally after memory fence operations (barrier, kernel begin, and kernel end) [5]. In the Kepler parts, the L1 cache is disabled for all globally visible writes. Therefore, to implement a memory fence, that architecture only needs to wait for all outstanding writes (e.g., in a write buffer) to complete. The Southern Islands parts use the L1 cache for globally visible writes; therefore, the AMD parts implement a memory fence by invalidating all data in the L1 cache and flushing all written data to the shared L2 (via a cache walk) [8].

2.3. Release Consistency on GPUs

RC [10] has been adopted at least partially by ARM [11], Alpha [12], and Itanium [13] architectures and seems like a reasonable candidate for GPUs because it is adequately weak for many hardware designs, but strong enough to reason easily about data races. In addition, future AMD and ARM GPUs and APUs will be compliant with the HSA memory model, which is defined to be RC [1]. The rest of this paper will assume that the memory system implementation must obey RC [14].

The HSA memory model [15] adds explicit LdAcq and StRel instructions. They will be sequentially consistent. In addition, they will enforce a downward and upward fence, respectively. Unlike a CPU consistency model, enforcing the HSA memory model is not strictly the job of the hardware; it is possible to use a finalizer (an intermediate assembly language compiler) to help enforce consistency with low-level instructions. In this paper, we consider hardware solutions to enforcing RC.

2.4. Supporting Release Consistency

In this section, two possible baseline APU implementations of RC are described. The first is a slight modification to the system described in Section 2.2. The second is a naïve implementation of a traditional CPU RFO cache-coherence protocol applied to an APU. Both support RC as specified.

2.4.1. Realistic Write-through GPU Memory System

The current GPU memory system described in Section 2.2 can adhere to the RC model between the CPU and GPU requests by writing through to memory via the APU directory. This means that a release operation (kernel end, barrier, or StRel) will need to wait for all prior writes to be visible globally before executing more memory operations. In addition, an acquiring memory fence (kernel begin or LdAcq) will invalidate all clean and potentially stale L1 cache data.

2.4.2. "Read for Ownership" GPU Memory System

Current multi-core CPU processors implement shared memory with write-back cache coherence [16]. As the RFO name implies, these systems will perform a read to gain ownership of a cache block before performing a write. In doing so, RFO protocols maintain the invariant that at any point in time only a single writer or multiple readers exist for a given cache block.

To understand the benefit an RFO protocol can provide GPUs, we added a directory to our baseline GPU cache hierarchy. It is illustrated in Figure 2, where the wL2 and wL3 are replaced by a fully mapped directory with full sharer state [17]. The directory's contents are inclusive of the L1s and L2, and the directory maintains coherence by allowing a single writer or multiple readers to cache a block at any time. Because there is finite state storage, the directory can recall data from the L1 or L2 to free directory space. The

protocol here closely resembles the coherence protocol in recent AMD CPU architectures [18].

2.5. Related Work

Recent work by Singh et al. in cache coherence on GPUs has shown that a naïve CPU-like RFO protocol will incur significant overheads [2]. This work does not include integration with CPUs.

Recent work by Hechtman and Sorin also explored memory consistency implementations on GPU-like architectures and showed that strong consistency is viable for massively threaded architectures that implement RFO cache coherence [4]. QR relies on a similar insight: read-after-write dependencies through memory are rare on GPU workloads.

Similar to the evaluated WT protocol for a GPU, the VIPS-m protocol for a CPU lazily writes through shared data by the time synchronization events are complete [19]. However, VIPS-m relies on tracking individual lazy writes using MSHRs, while the WT design does not require MSHRs and instead relies on in-order memory responses to maintain the proper synchronization order.

Conceptually, QR caches act like store queues (also called load/store queues, store buffers, or write buffers) that are found in CPUs that implement weak consistency models [20]. They have a logical FIFO organization that easily enforces ordering constraints at memory fences, thus leading to fast fine-grain synchronization. Also like a store queue, QR caches allow bypassing from the FIFO organization for high performance. This FIFO organization is only a logical wrapping, though. Under the hood, QR separates the read and write data paths and uses high-throughput, unordered write-combining caches.

Store-wait-free systems also implement a logical FIFO in parallel with the L1 cache to enforce atomic sequence order [21]. Similarly, implementations of transactional coherence and consistency (TCC) [22] use an address FIFO in parallel with the L1. However, TCC's address FIFO is used for transaction conflict detection while QR's address FIFO is used to ensure proper synchronization order.

3. QuickRelease Operation

In this section, we describe in detail how a QR cache hierarchy operates in a state-of-the-art SoC architecture that resembles an AMD APU. Figure 2 shows a diagram of the system, which features a GPU component with two levels of write-combining cache and a memory-side L3 cache shared by the CPU and GPU. For QR, we split the GPU caches into separate read and write caches to reduce implementation cost (more detail below). At each level, the write cache is approximately a quarter to an eighth the size of the read cache. Additionally, we add an S-FIFO structure in parallel with each write cache.

A goal of QR is to maintain performance for graphics workloads. At a high level, a QR design behaves like a conventional throughput-optimized write-combining cache: writes complete immediately without having to read the block first, and blocks stay in the cache until selected for eviction by a replacement policy. Because blocks are written without acquiring either permission or data, both write-combining and QR caches maintain a bitmask to track which bytes in a block are dirty, and use that mask to prevent loads from reading bytes that have not been read or written.

The QR design improves on conventional write-combining caches in two ways that increase synchronization performance and reduce implementation cost. First, QR caches use the S-FIFO to track which blocks in a cache might contain dirty data. A QR cache uses this structure to eliminate the need to perform a cache walk at synchronization events, as is done in conventional write-combining designs. Second, the QR design partitions the resources devoted to reads and writes by using read-only and write-only caches. Because writes are more expensive than reads (e.g., they require a bitmask), this reduces the overall cost of a QR design. We discuss the benefits of this separation in more detail in Section 3.2, and for now focus on the operation and benefits of the S-FIFO structures.

When a conventional write-combining design encounters a release, it initiates a cache walk to find and flush all dirty blocks in the cache. This relatively long-latency operation consumes cache ports and discourages the use of fine-grain synchronization. This operation is heavy-weight because many threads share the same L1 cache, and one thread synchronizing can prevent other threads from re-using data. QR overcomes this problem by using the S-FIFO. At any time, the S-FIFO contains a superset of addresses that may be dirty in the cache. The S-FIFO contains at least the addresses present in the write cache, but may contain more addresses that already have been evicted from the write cache. It is easy to iterate the S-FIFO on a release to find and flush the necessary write-cache data blocks. Conceptually the S-FIFO can be split into multiple FIFOs for each wavefront, thread, or work-group, but we found such a split provides minimal performance benefit and breaks the transitivity property on which some programs may rely [23]. Furthermore, a strict FIFO is not required to maintain a partial order of writes with respect to release operations, but we chose it because it is easy to implement.

In the following sub-sections, we describe in detail how QR performs different memory operations. First, we document the lifetime of a write operation, describing how the writes propagate through the write-only memory hierarchy and interact with S-FIFOs. Second, we document the lifetime of a basic read operation, particularly how this operation can be satisfied entirely by the separate read-optimized data path. Third, we describe how the system uses S-FIFOs to

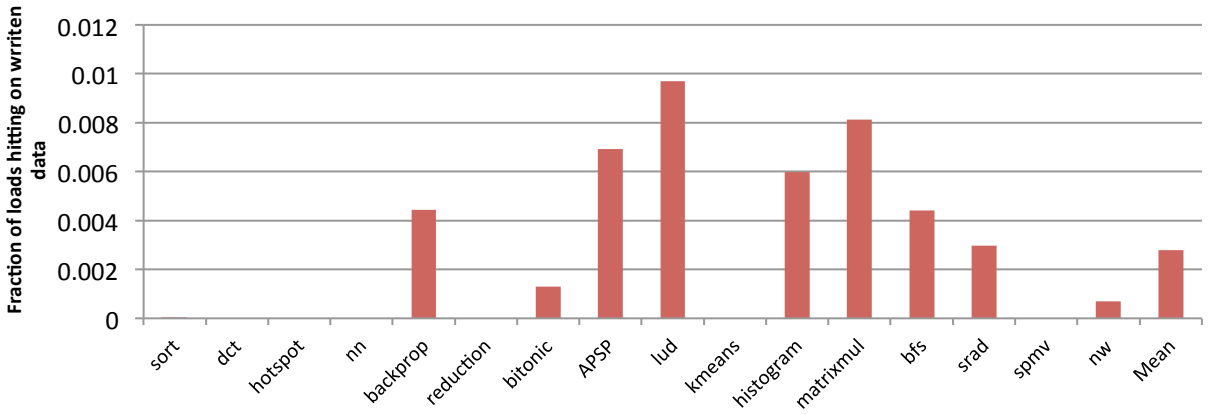

Figure 3: L1 read-after-write re-use (L1 read hits in M for RFO memory system).

synchronize between release and acquire events. Fourth, we discuss how reads and writes interact when the same address is found in both the read and write paths, and show how QR ensures correct single-threaded read-after-write semantics.

3.1. Detailed Operation

3.1.1. Normal Write Operation

To complete a normal store operation, a CU inserts the write into the wL1, enqueues the address at the tail of the L1 S-FIFO, and, if the block is found in the rL1, sets a written bit in the tag to mark that updated data is in the wL1. The updated data will stay in the wL1 until the block is selected for eviction by the wL1 replacement policy or the address reaches the head of the S-FIFO. In either case, when evicted, the controller also will invalidate the block in the rL1, if it is present. This invalidation step is necessary to ensure correct synchronization and read-after-write operations (more details in Section 3.1.3). Writes never receive an ack.

The operation of a wL2 is similar, though with the addition of an L1 invalidation step. When a wL2 evicts a block, it invalidates the local rL2 and broadcasts an invalidation message to all the rL1s. Broadcasting to eight or 16 CUs is not a huge burden and can be alleviated with coarse-grain sharer tracking because writing to temporally shared data is unlikely without synchronization. This ensures that when using the S-FIFOs to implement synchronization, the system does not inadvertently allow a core to perform a stale read. For similar reasons, when a line is evicted from the wL3, the controller sends invalidations to the CPU cluster, the group of CPUs connected to the directory, before the line is written to the L3 cache or main memory.

Completing an atomic operation also inserts a write marker into the S-FIFO, but instead of lazily writing through to memory, the atomic is forwarded immediately to the point of system coherence, which is the directory.

CPUs perform stores as normal with coherent write-back caches. The APU directory will invalidate the rL2, which in turn will invalidate the rL1 caches to ensure consistency with respect to CPU writes at each CU. Because read caches never contain dirty data, they never need to respond with data to invalidation messages even if there is a write outstanding in the wL1/wL2/wL3. This means that CPU invalidations can be applied lazily.

3.1.2. Normal Read Operation

To perform a load at any level of the QR hierarchy, the read-cache tags simply are checked to see if the address is present. If the load hits valid data and the written bit is clear, the load will complete without touching the write-cache tags. On a read-tag miss or when the written bit is set, the write cache is checked to see if the load can be satisfied fully by dirty bytes present in the write cache. If so, the load is completed with the data from the write cache; otherwise, if the read request at least partially misses in the write cache, the dirty bytes are written through from the write-only cache and the read request is sent to the next level of the hierarchy.

While the write caches and their associated synchronization FIFOs ensure that data values are written to memory before release operations are completed, stale data values in the read caches also must be invalidated to achieve RC. QR invalidates these stale data copies by broadcasting invalidation messages to all rL1s when there is an eviction from the wL2. Though this may be a large amount of traffic, invalidations are much less frequent than individual stores because of significant coalescing in the wL1 and wL2. By avoiding cache flushes, valid data can persist in the rL1 across release operations, and the consequential reduction of data traffic between the rL2 and rL1 may compensate entirely for the invalidation bandwidth.

Furthermore, these invalidations are not critical to performance, unlike a traditional cache-coherence protocol in which stores depend on the acks to complete. In QR, the invalidations only delay synchronization completion. This delay is bounded based on the number of entries in the syn-

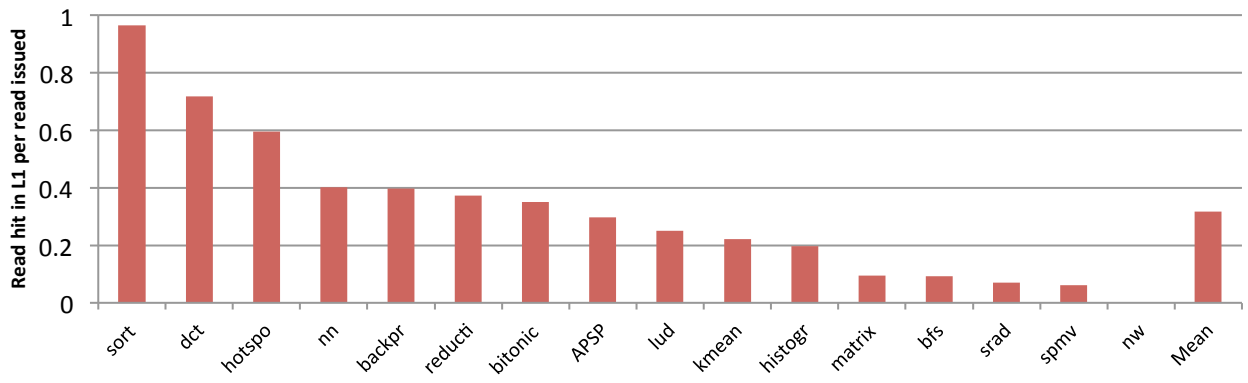

Figure 4: L1 cache read re-use (read hits per read access in RFO memory system).

chronization FIFO when a synchronization operation arrives. Meanwhile, write evictions and read requests do not stall waiting for invalidations because the system does not support strong consistency. As a result, QR incurs minimal performance overhead compared to a WT memory system when synchronization is rare.

QR's impact on CPU coherence is minimal and the CPUs perform loads as normal. For instance, a CPU read never will be forwarded to the GPU memory hierarchy because main memory already contains all globally visible data written by the GPU. A CPU write requires only invalidation messages to be issued to the GPU caches.

3.1.3. Synchronization

While loads and stores can proceed in write-combining caches without coherence actions, outstanding writes must complete to main memory and stale read-only data must be invalidated at synchronization events. QR caches implement these operations efficiently with the help of the S-FIFOs.

To start a release operation (e.g., a StRel or kernel end), a wavefront enqueues a special release marker onto the L1 S-FIFO. When inserted, the marker will cause the cache controller to begin dequeuing the S-FIFO (and performing the associated cache evictions) until the release marker reaches the head of the queue. The StRel does not require that the writes be flushed immediately; the StRel requires only that all stores in the S-FIFO hierarchy be ordered before the store of the StRel. The marker then will propagate through the cache hierarchy just like a normal write.

When the marker finally reaches the head of the wL3, the system can be sure that all prior writes from the wavefront have reached an ordering point (i.e., main memory). An acknowledgement is sent to the wavefront to signal that the release is complete.

When the release operation has an associated store operation (i.e., a StRel), the store can proceed as a normal store in the write path after the release completes. However, for performance, the store associated with the StRel should complete as soon as possible in case another thread is waiting for that synchronization to complete. Therefore, a store from a StRel will also trigger S-FIFO flushes, but it will not send an acknowledgement message back to the requesting wavefront.

Because QR broadcasts invalidations on dirty evictions, ensuring all stale data is invalidated before a release operation completes, acquire operations can be implemented as simple, light-weight loads; the acquire itself is a no-op. If a LdAcq receives the value from a previous StRel, the system can be sure that any value written by the releasing thread will have been written back to main memory and any corresponding value in a read-only cache has been invalidated.

3.2. Read/Write Partitioning Trade-offs

In the QR design, we chose to partition the cache resources for reads and writes. While this choice reduces implementation complexity, it adds some overhead to read-after-write sequences. For example, in QR a load that hits in the write cache requires two tag look-ups and a data look-up: first check the read-cache tags, then check the write-cache tags, then read from the write-cache data array. We can justify this overhead by observing that GPGPU applications rarely demonstrate read-after-write locality.

Figure 3 shows the percentage of read requests that hit an L1 cache block that has been written previously (i.e., is in a modified state under RFO). For several evaluated applications, written L1 cache blocks are never re-accessed. This occurs due to a common GPU application design pattern in which a kernel streams through data, reading one data set and writing another. Subsequently, another kernel will be launched to read the written data, but by this time all that data will have been evicted from the cache.

The partitioned design has several implementation benefits. First, it reduces the state overhead needed to support writes in a write-combining cache because the dirty bitmasks are required only in the write caches. Second, it is easier to build two separate caches than a single multi-ported

read/write cache with equivalent throughput. Third, the read cache can be integrated closely with the register file to improve L1 read hit latency. Meanwhile the write cache can be moved closer to the L2 bus interface and optimized exclusively as a bandwidth buffer.

Table 1: Memory System Parameters

Baseline				
Frequency		1 GHz		
Wavefronts		64 wide, 4 cycle		
Compute units		8, 40 wavefronts each		
Memory		DDR3, 4 Channels, 400 MHz		
	banks	*tag lat.*	*data lat.*	*size*
L1	16	1	4	16 kB
L2	16	4	16	256 kB
QR				
wL1	16	1	4	4 kB
wL2	16	4	16	16 kB
wL3	16	4	16	32 kB
S-FIFO1		64 entries		
S-FIFO2		128 entries		
S-FIFO3		256 entries		
total		80 kB		
RFO				
directory		256 kB		
MSHRs		1,024		
total		384 kB		

4. Simulation Methodology and Workloads

4.1. The APU Simulator

Our simulation methodology extends the gem5 simulator [24] with a microarchitectural timing model of a GPU that directly executes the HSA Intermediate Language (HSAIL) [1]. To run OpenCL applications, we first generate an x86 binary that links an OpenCL library compatible with gem5's syscall emulation environment. Meanwhile, the OpenCL kernels are compiled directly into HSAIL using a proprietary industrial compiler.

Because the simulation of our OpenCL environment is HSA-compliant, the CPU and GPU share virtual memory and all memory accesses from both the CPU and GPU are assumed to be coherent. As a result, data copies between the CPU and GPU are unnecessary.

In this work, we simulate an APU-like system [25] in which the CPU and the GPU share a single directory and DRAM controller. The GPU consists of CUs. Each CU has a private L1 data cache and all the CUs share an L2 cache. The L2 further is connected to a stateless (a.k.a. null) directory [26] with a memory-side 4-MB L3 cache, which is writeable only in the RFO system. The configurations of WT, RFO, and QR are listed in Table 1.

As previously noted, the storage overhead of QR compared to WT is similar to dirty bits for all WT caches. Figure 2 summarizes this design with a block diagram. Overall, QR uses 80 kB of additional storage that is not present in the WT baseline. To ensure that the comparison with WT is fair, we tested whether doubling the L1 capacity could benefit the WT design. Further, the RFO design requires nearly double the storage of the baseline WT memory system. We found that the extra capacity provided little benefit because of the lack of temporal locality in the evaluated benchmarks. The benefit is reduced further because WT's caches must be flushed on kernel launches.

4.2. Benchmarks

We evaluate QR against a conventional GPU design that uses WT caches and an idealized GPU memory system that uses RFO coherence. We run our evaluation on a set of benchmarks with diverse compute and sharing characteristics. The benchmarks represent the current state-of-the-art for GPU benchmarks. The applications and compute kernels come from the AMD APP SDK [27], OpenDwarfs [28], Rodinia [3], and two microbenchmarks that were designed to have increased data re-use and synchronization. Our microbenchmarks attempt to approximate the behavior of future workloads, which we expect will have more frequent synchronization and data re-use. Here is a brief description of the microbenchmarks:

- **APSP:** Performs a single-source shortest path until converging on an all-pairs shortest path. This application uses LdAcq and StRel to view updates as soon as they are available, to speed convergence, and uses multiple kernel launches to perform frequent communication with the host.
- **sort:** Performs a 4-byte radix sort byte by byte. For each byte, the first step counts the number of elements of each byte; the second step traverses the list to find the value at the thread ID position; and, the final step moves the correct value to the correct location and swaps the input and output arrays.

4.3. Re-use of the L1 Data Cache

Figure 4 shows the measured L1 read hits as a fraction of read requests (i.e., re-use rate) in the RFO memory system. RFO allows for a longer re-use window than either the QR or WT memory systems because cache blocks are written only locally and synchronization does not force dirty data to a common coherency point. In contrast, the WT and QR memory systems must ensure all writes are performed to memory before synchronization completes. In addition, WT will invalidate its L1 cache on each kernel launch.

The workloads from Section 4.2 exhibit a huge range of re-use rates, capturing the diverse range of traffic patterns exhibited by GPGPU applications. In either of the extremes of re-use, we expect that all of the memory systems should

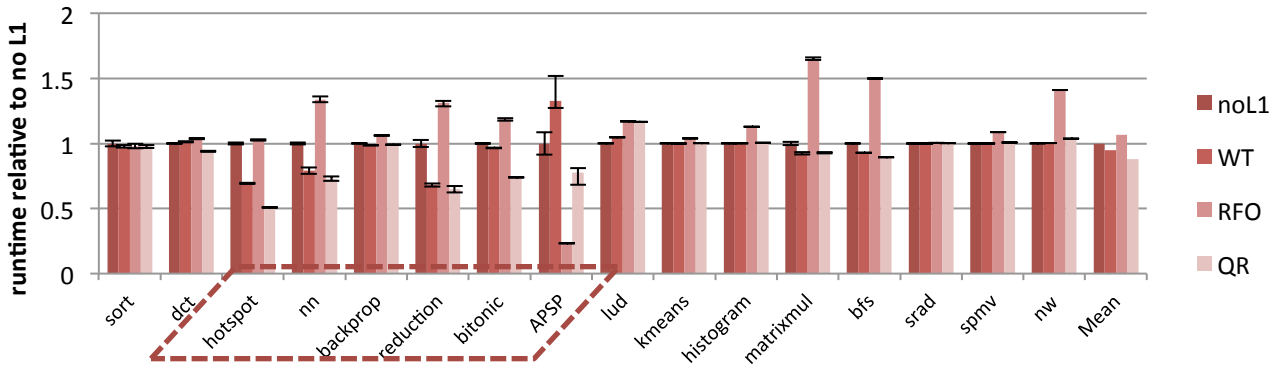

Figure 5: Relative run-times of WT, RFO, and QR memory systems compared to not using an L1 cache.

perform equivalently. In applications with a high re-use rate, L1 cache hits will dominate the run-time. In applications with a low re-use rate, the performance will be bound by the memory bandwidth and latency. Because L1 cache and memory controller designs are effectively equivalent in QR, RFO, and WT, the expected performance is also equivalent.

5. Results

5.1. Performance

Figure 5 plots the relative run-times of WT, RFO, and QR relative to a system that disables the L1 cache for coherent traffic, similar to NVIDIA's Kepler architecture. The applications are ordered across the x-axis by their L1 re-use rate (Figure 4). The final set of bars shows the geometric mean of the normalized run-times. Overall, QR gains 7% performance compared to WT, which gains only 5% performance compared to not using an L1 cache. On the other hand, the RFO memory system loses 6% performance relative to a memory system with no L1 cache. The RFO performance drop comes from the additional latency imposed to write operations because they first must acquire exclusive coherence permissions.

Figure 5 supports the insight that a QR memory system would outperform a WT memory system significantly when there is an intermediate amount of L1 re-use. In particular, QR outperforms WT by 6-42% across six of the seven workloads (dotted-line box in Figure 5) because there is significant L1 re-use across kernel boundaries and LdAcqs. In these applications, the WT memory system cannot re-use any data due to the frequency of full cache invalidations. The lone exception is backprop, which is dominated by pulling data from the CPU caches; thus, QR and WT see similar performance.

Across the seven highlighted workloads, APSP is particularly noticeable because of the impressive performance improvement achieved by QR and the even more impressive performance improvement achieved by RFO. APSP is the only benchmark that frequently uses LdAcq and StRel instructions within its kernels. While the QR memory system efficiently performs the LdAcq and StRel operations in a write-combining memory system, the RFO memory system performs the operations much faster at its local L1 cache. The resulting memory access timings for the RFO memory system lead to far less branch divergence and fewer kernel launches compared to the other memory systems because the algorithm launches kernels until there is convergence.

The applications bfs, matrixmul, and dct are on the border between intermediate and high or low re-use. As a result, the performance advantage of QR relative to WT is muted.

Similar to backprop, kmeans and histogram invoke many kernel launches and frequently share data between the CPU and GPU. Their performance also is dominated by pulling data in from the CPU, resulting in QR and WT achieving similar performance.

The one application on which QR encounters noticeable performance degradation is lud. As shown in Figure 3, lud exhibits the highest rate of temporal read-after-writes; thus, the extra latency of moving data between QR's separate read and write caches is exposed. Furthermore, lud has a high degree of false sharing between CUs, which lowers the effectiveness of QR's L1 cache compared to WT due to its cache block granular invalidations. Overall, due to its unique behavior, lud is the only benchmark on which simply disabling the L1 cache achieves a noticeable performance improvement relative to the other designs.

The rest of the applications (sort, srad, spmv, and nw) exhibit either very high or very low L1 re-use, which means we would expect a small performance difference due to the on-chip memory system. The results confirm this intuition because all non-RFO memory systems perform similarly.

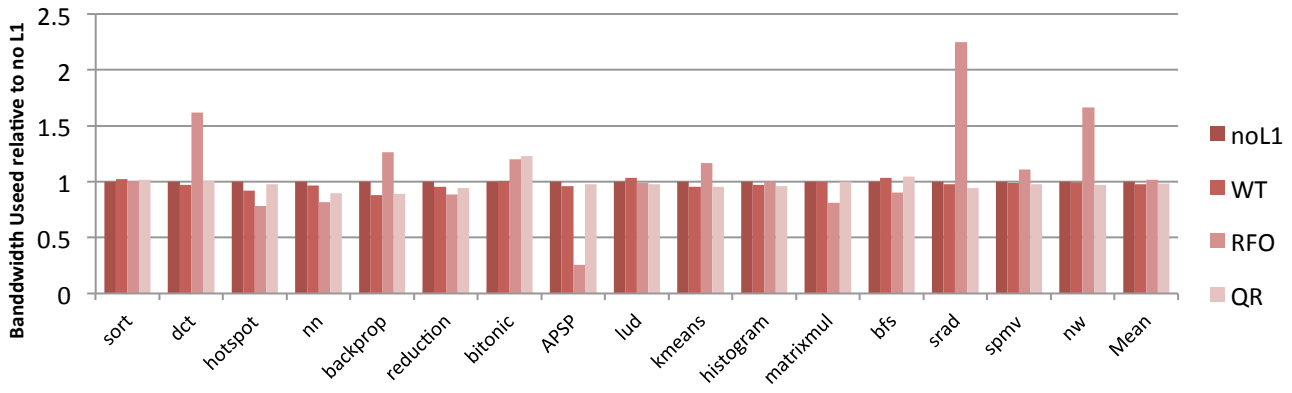

Figure 6: L2 to directory bandwidth relative to no L1.

5.2. Directory Traffic

Figure 6 shows the bandwidth between the GPU cache hierarchy and the APU directory for WT, RFO, and QR relative to the system without an L1 cache. Due to aggressive write-combining, QR generates less total write traffic than WT for the same or better performance.

To explore the directory write traffic, Figure 7 shows the effectiveness of the write-combining performed by a QR memory system. The RFO memory system includes a memory-side L3 cache, which filters many DRAM writes, so only the no-L1-memory, WT, and QR designs are shown in Figure 7. Most applications see significantly fewer write requests at the DRAM in QR compared to a WT or no-L1-memory system due to the write-combining performed at the wL1, wL2, and wL3. As Figure 7 shows, applications with the greatest reduction generally achieve the greatest performance gains, indicating that good write-combining is critical to performance. In nn and nw, WT and QR have similar DRAM traffic. In these applications, there is no opportunity to perform additional write-combining in QR because all of the writes are full-cache-line operations and each address is written only once.

5.3. L1 Invalidation Overhead

Figure 8 shows both the cost and benefit of broadcasting precise invalidations in QR. Bars represent the normalized number of bytes that arrive at the L1 cache in QR compared to WT. Within each bar, segments correspond to the number of bytes that arrived due to an invalidation probe request or a data response, respectively.

Almost all benchmarks receive equal or fewer L1 data messages in a QR memory system compared to a WT memory system. The only exception is backprop, in which false sharing created additional cache misses for QR due to invalidations after wL2 evictions.

When invalidation traffic is added, the total bytes arriving at the L1 in a QR memory system can be up to three times the number of bytes arriving in a WT system, though on average the number is comparable (103%). Some workloads even experience a reduction in L1 traffic. APSP saw a significant reduction in overall traffic because frequent LdAcqs and the subsequent cache invalidations result in a 0% hit rate at the WT L1. In most workloads, QR and WT have comparable traffic at the L1. QR achieves this comparable traffic despite extra invalidations because it is able to re-use data across kernel boundaries, whereas WT's full L1 cache invalidation cause data to be refetched.

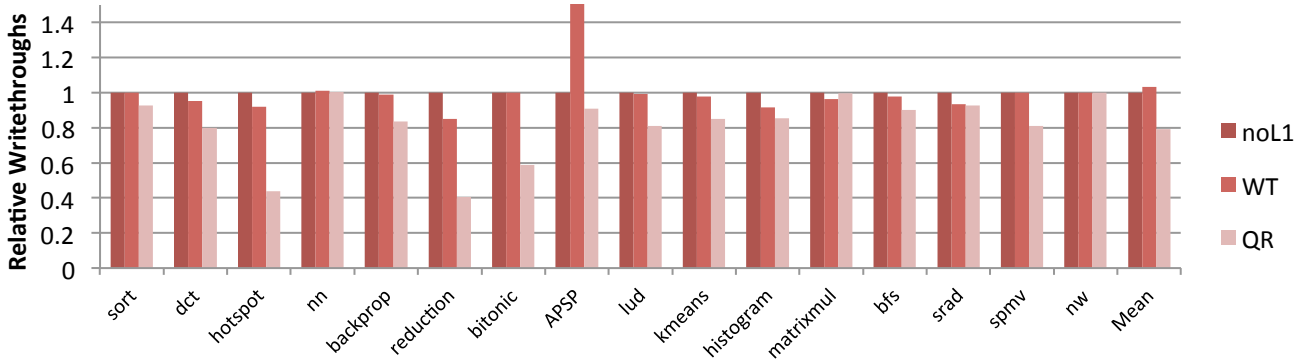

Figure 7: Write-through requests seen at DRAM relative to a system with no L1.

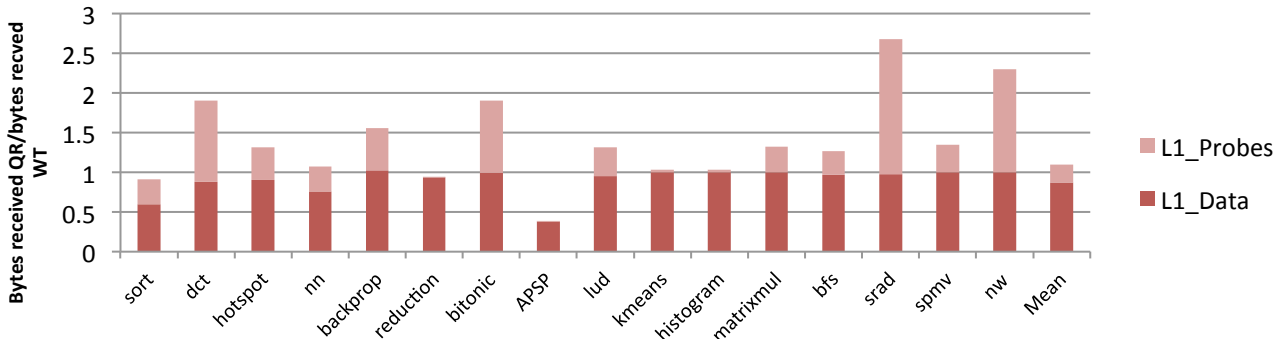

Figure 8: Invalidation and data messages received at the QR L1 compared to WT data messages.

Finally, other workloads see a doubling or more of L1 traffic in QR. This is because they have a significant number of independent writes without re-use between kernels to amortize the cost of invalidations. In the future, we predict that reducing the data required from off-chip likely will trump the cost of additional on-chip invalidation messages, making QR a reasonable design despite this increased L1 traffic.

5.4. Total Memory Bandwidth

Figure 9 shows the combined number of read and write memory accesses for each benchmark relative to the memory accesses performed by the memory system with no L1. The RFO has fewer memory reads because dirty data is cached across kernel bounds, which is not possible in the QR or WT memory systems because data responses to CPU probes are not supported. This is especially effective because kernels often switch the input and output pointers such that previously written data in the last kernel is re-used in the next kernel invocation.

5.5. Power

Combining the results from Figure 8 and Figure 9, we can estimate the network and memory power of QR and WT. Because GPUWattch showed that memory consumed 30% of power on modern GPUs and network consumed 10% of power [29], we can infer that QR should save 5% of memory power and increase network power by 3%. As a result, it follows that QR should save a marginal amount of power that may be used by the additional write caches. Further, the improved performance of QR relative to WT implies less total energy consumption.

5.6. Scalability of RFO

To support the claim of increased bandwidth scalability compared to an RFO memory system, nn and reduction are evaluated with smaller inputs to see how well a latency-oriented RFO memory system could perform compared to a throughput-oriented WT or QR memory system. Figure 10 shows the performance of nn and reduction for various problem sizes. For small input sets, all memory systems have similar performance. As the input size increases, the demand on the memory system increases and QR's reduced write overhead improves the performance relative to RFO and WT.

6. Conclusion

This paper demonstrates that QuickRelease can expand the applicability of GPUs by efficiently executing the fine-grain synchronization required by many irregular parallel workloads while maintaining good performance on traditional, regular general-purpose GPU workloads. The QR design

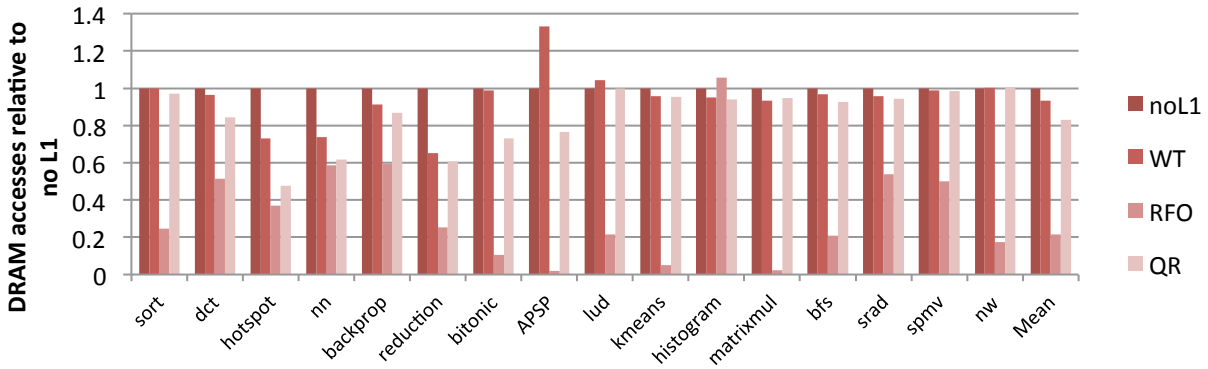

Figure 9: Total DRAM accesses by WT, RFO and QR relative to no L1.

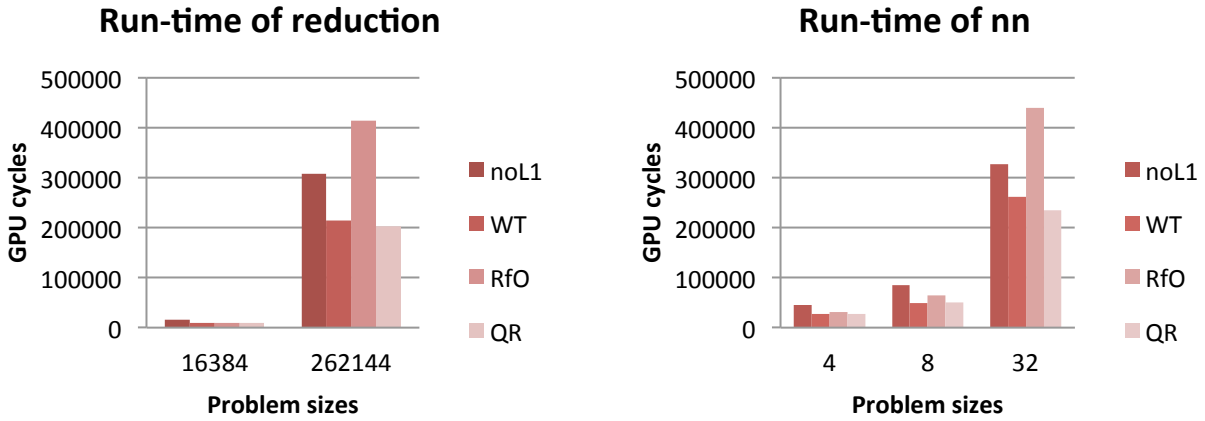

Figure 10: Scalability comparison for increasing problem sizes.

improves on conventional write-combining caches in ways that improve synchronization performance and reduce the cost of supporting writes. First, QR improves performance by using efficient synchronization FIFOs to track outstanding writes, obviating the need for high-overhead cache walks. Second, QR reduces the cost of write support by partitioning the read- and write-cache resources, exploiting the observation that writes are more costly than reads.

The evaluation compares QR to a GPU memory system that simply disables private L1 caches for coherent data and a traditional throughput-oriented write-through memory system. To illustrate the intuitive analysis of QR, it also is compared to an idealized RFO memory system. The results demonstrate that QR achieves the best qualities of each baseline design.

References

[1] HSA Foundation, "Deeper Look Into HSAIL And It's Runtime," 25-Jul-2012.

[2] I. Singh, A. Shriraram, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *Proceedings of HPCA* 2013, pp. 578–590.

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

[4] B. A. Hechtman and D. J. Sorin, "Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors," in *Proceedings of ISCA*, 2013.

[5] A. Munshi, "OpenCL," *Parallel Computing on the GPU and CPU, SIGGRAPH*, 2008.

[6] M. Harris, "Many-core GPU computing with NVIDIA CUDA," in *Proceedings of SC* 2008, pp. 1–1.

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

[8] AMD, *Southern Islands Series Instruction Set Architecture*. 2012.

[9] HSA Foundation, "AFDS 2011 Phil Rogers Keynote: "The Programmer's Guide to the APU ...," 10-Jun-2012.

[10] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[11] J. Goodacre and A. N. Sloss, "Parallelism and the ARM Instruction Set Architecture," *IEEE Computer*, vol. 38, no. 7, pp. 42–50, Jul. 2005.

[12] Compaq, *Alpha 21264 Microprocessor Hardware Reference Manual*. 1999.

[13] *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. 2002.

[14] M. Raynal and A. Schiper, "From causal consistency to sequential consistency in shared memory systems," 1995, pp. 180–194.

[15] "Standards," *HSA Foundation*.

[16] D. J. Sorin, M. D. Hill, and D. A. Wood, "A Primer on Memory Consistency and Cache Coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, May 2011.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH Multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[18] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Apr. 2010.

[19] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proceedings of PACT* 2012, pp. 241–252.

[20] T. Sha, M. M. K. Martin, and A. Roth, "Scalable Store-Load Forwarding via Store Queue Index Prediction," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005, pp. 159–170.

[21] A. A. Thomas F. Wenisch and A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors," in *Proceedings ISCA* 2007.

[22] A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on chip-multiprocessors," in *Proceedings of PACT 2005*, pp. 63–74.

[23] D. R. Hower, Hechtman, Blake A., Beckmann, Bradford M., Gaster, Benedict R., Hill, Mark D., Reinhardt, Steven K., and Wood, David A., "Heterogeneous-Race-Free Memory Models," ASPLOS 2014.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[25] A. Branover, D. Foley, and M. Steinman, "AMD's Llano Fusion APU," *IEEE Micro*, vol. 99, no. 1, 5555.

[26] P. Conway and B. Hughes, "The AMD Opteron Northbridge Architecture," *IEEE Micro*, vol. 27, no. 2, pp. 10–21, Apr. 2007.

[27] AMD, *Accelerated Parallel Processing (APP) SDK*. 2013.

[28] W. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 dwarfs: a work in progress," in *Proceedings of WOSP/SIPEW* 2012, pp. 291–294.

[29] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *proc. of ISCA*, 2013, vol. 40.