Accelerating Atomic Operations on GPGPUs

Sean Franey and Mikko Lipasti
Electrical and Computer Engineering, University of Wisconsin - Madison
sfraney@wisc.edu, mikko@engr.wisc.edu

***Abstract*—General purpose computing on GPUs (GPGPU) has experienced rapid growth over the last several years as new application realms are explored and traditional highly parallel algorithms are adapted to this computational substrate. However, a large portion of the parallel workload space, both in emerging and traditional domains, remains ill-suited for GPGPU deployment due to high reliance on atomic operations, particularly as global synchronization mechanisms. Unlike the sophisticated synchronization primitives available on supercomputers, GPGPU applications must rely on slow atomic operations on shared data. Further, unlike general purpose processors which take advantage of coherent L1 caches to speed up atomic operations, the cost and complexity of coherency on the GPU, coupled with the fact that a GPU's primary revenue stream - graphics rendering - does not benefit, means that new approaches are needed to improve atomics on the GPU. In this paper, we present a mechanism for implementing low-cost coherence and speculative acquisition of atomic data on the GPU that allows applications that utilize atomics to greater extents than is generally accepted practice today, to perform much better than they do on current hardware. As our results show, these unconventional applications can realize non-trivial performance improvements approaching 20% with our proposed system. With this mechanism, the scope of applications that can be accelerated by these commodity, highly-parallel pieces of hardware can be greatly expanded.**

I. Introduction

Over the past several years, the previously rigid, fixed function units making up graphics processing units (GPUs) have evolved to meet the ever-more-general demands of advanced shaders for rendering graphics. Through this evolution, the GPU has changed to look much more like a highly parallel general purpose processor, not unlike expensive, low volume supercomputers. Correspondingly, early adopters motivated the evolution of the programming environment for these processors to make their capabilities available to a broader audience. This ecosystem of advanced programming environments for commodity, highly parallel processors has led to the emergence of general purpose computing on GPUs (GPGPU). Through this emergence, many applications that were previously limited to the costly realm of supercomputing, have now been ported to the GPU. While significant work in both the environment and hardware continues to expand the application scope for these parts, many highly parallel applications remain out of reach.

One large portion of the application space that remains unrealized on the GPU is that of parallel workloads that utilize atomic operations to update globally shared data, particularly those that utilize these updates to perform coarse, global synchronization. This is due to the fact that modern GPUs do not efficiently support synchronization outside of very local work units. In NVIDIA terminology for instance, there is no direct support for synchronizing threads across computational blocks (CTAs). While in the past, some applications have had access to specialized hardware [1] or message passing techniques [2] [3] for synchronization, GPGPU application have had to rely solely on atomic operations to global data. While this is a common technique utilized in general purpose CPUs, the lack of L1 cache coherence in GPUs make these operations significantly slower. Further, though coherence in fused CPU/GPU chips appears inevitable to support the desired tight coupling between the two units, it seems unlikely that the L1 coherence between processing units *within* the GPU, necessary to significantly improve these synchronization operations, will be seen on the GPU. This is due to the traditionally high design and verification cost of coherence, coupled with the fact that the primary source of GPU revenue - rendering of graphics - does not benefit from it. Therefore, it seems apparent that opening up this application class to the GPU will require new, inexpensive mechanisms.

In this paper, we will discuss two such mechanisms to provide high speed atomic operations to applications on the GPU. The first is a rather simple adaptation of Atomic Coherence [4] that allows the GPU to implement L1 cache coherence on atomic data in a complexity-effective way. Atomic Coherence contains complexity by mimicking a traditional blocking bus, thereby preventing races and the associated state explosion in the coherence controller, while retaining the performance of a non-blocking interconnect. It accomplishes this by mandating a node acquire a mutex corresponding to the data triggering the coherence action. Further, only the node holding the mutex can utilize the interconnect for the associated data, preventing races and leaving the interconnect available for non-conflicting activities. The serialization penalty of mutex acquisition is mitigated by a ultra-low-latency nanophotonic ring. Even without such an interconnect, Section II-A adapts this approach over a novel electrical interconnect that performs reasonably well by providing coherence only for atomic data. This approach reduces the cost of providing L1 coherence to a level low enough to merit adoption on a cost-sensitive GPGPU, while providing substantial improvements in the performance of synchronizing operations.

The second approach we present for improving atomic performance builds on the first with very little incremental cost (to be quantified in Section IV-G). The intuition behind it is that the state maintained by acquisition of mutexes for Atomic Coherence itself, is nearly enough to ensure safe access to atomic data. By removing the coherence controller

altogether and merely extending the information tracked at the nodes responsible for the mutexes and maintaining some information at the requester, we can determine if a node can safely access the associated data from its local cache without initiating an action beyond acquiring the mutex. Therefore, if enough locality exists in the access stream of atomic data (i.e., the same nodes are modifying the same data over a window of time), this mechanism can improve performance by eliminating coherence activity in the common case and fallback to a global access as a failsafe when mutex acquisition indicates multiple nodes are sharing the data.

The rest of this paper will elaborate on these mechanisms, systematically building them from a baseline Atomic Coherence implementation on a novel electrical interconnect topology.

II. Caching Atomic Data

As mentioned in the introduction, L1 cache coherence is a feature lacking in GPUs and one that is not likely to find support in the foreseeable future. Unfortunately, for atomic operations to approach the performance available on current general purpose CPUs, something akin to coherence for these operations is imperative. In order to understand the importance of this support, one must have an understanding of the current state-of-the-art regarding atomic operations on the GPU. Unfortunately, to the best of our knowledge, implementation of atomic operations on a GPU have not been publicly described. While it might be most intuitive to assume that atomic instructions are executed like non-atomic instructions in the shader core, some have suggested that these operations actually occur at the memory interface by the extension of alpha blending hardware to perform them [5]. This scenario is not only plausible, but corroborated by microbenchmarks [6]. Under this assumption, atomic requests are both ordered and performed remotely, after traversing the interconnect from the shader core to the corresponding memory bank. With the introduction of Fermi, an L2 cache is available which can greatly improve atomic performance [7] by performing them at the L2 bank, rather than at memory. This still requires a traversal of the interconnect to the L2 bank, however. Therefore, according to our understanding of current state-of-the-art systems, an atomic operation is generated in the shader core and traverses the interconnect to the appropriate L2 bank. Once at the L2 bank, the operation is ordered, data is acquired, and the operation is performed. The new data is written back and a response is sent back to the core containing the previous value of the data (to be consistent with the CUDA programming guide [8]).

In order to achieve any significant latency reduction over this configuration then, the atomic operations must be performed locally - at the shader core itself - with local data, in order to avoid the latency of traversing the interconnect. This, in turn, requires the ability to cache atomic data at the shader core, and to do so coherently, and in a manner which supports their atomic semantics. The following section describes our proposed lightweight schemes, which are utilized strictly for atomic data to provide this support.

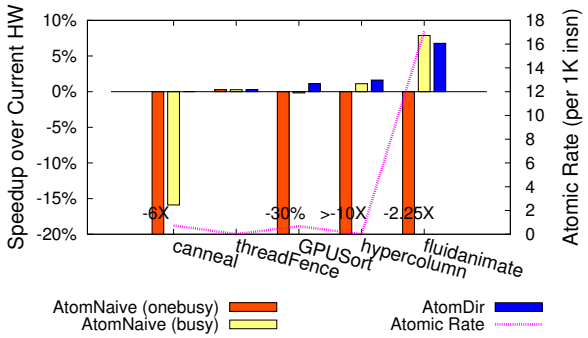

Fig. 1. *Atomic Coherence Results.*

A. Atomic Coherence

As previously described, Atomic Coherence is a mechanism that reduces design and verification complexity of coherence controllers by preventing race conditions, but it does so by leveraging a nanophotonic ring that is unique to the system it is evaluated on [4]. Therefore, a naïve first-pass implementation of Atomic Coherence on a GPU with an electrical interconnect would involve imposing a ring topology on the nodes in the system that would need to acquire mutexes (i.e., shader cores). We can limit cost by the realization that this ring need not be a physical set of wires, but can be mimicked more simply by a rotating token (what we will call a "talking stick" to avoid confusion between the similar terms 'token' and 'mutex'). This way, acquisition of a mutex involves waiting for the talking stick to reach the requesting node, acquiring the mutex (assuming it is available), and marking it unavailable for subsequent requesters. Release of the mutex would then again involve waiting for the talking stick to return and releasing the mutex (i.e., marking it available to the rest of the system).

In this system, the additional required structures are mutex status tables at each node to track the state of the mutexes in the system and buffers to hold requests until the talking stick arrives. The talking stick incurs no additional cost as it can be implied, essentially providing time-division multiplexed access to it. In other words, a node knows that it is the holder of the talking stick based on how many timesteps have occurred since it last had it (one could imagine a modulo operation on the cycle count). Cost can be further limited by utilizing the underlying interconnect to transmit updates.

Given that the underlying interconnect is likely to exhibit congestion or otherwise prevent the updates from travelling at the speed of the talking stick, we utilize a mechanism termed a "busy wire" to indicate to nodes when an update is in flight. This busy wire is a point-to-point wire in the interconnect, conceptually running along with the data links, that allows a node to indicate when an update is in flight. When it is in one state (nominally, '1') it indicates an inflight update and all mutex acquisitions are stalled, while the other state indicates no update is inflight and subsequently allows acquisitions to occur. While a busy wire is conceptually a single wire spanning all nodes in the system, since it is implemented point-to-point, it does not have the power and latency concerns such long global wires would have. Instead, each node is

responsible for keeping the busy wire asserted as indicated by its upstream node (as determined by talking stick rotation) until such time as that node indicates it no longer needs to be asserted. The state of the busy wire then propagates through the system at a rate at least equal to the talking stick to ensure no mutexes are acquired until the update completes. The same assertion/deassertion process occurs when a mutex is released.

Not surprisingly, this simple mechanism leads to a significant false conflict problem, where non-conflicting requests are stalled waiting for an update to traverse all nodes in the system. Figure 1 shows the results of this with the "AtomNaive (onebusy)" bars, showing that many applications are severely impacted. Therefore, for the small cost of additional point-to-point wires, analogous to adding a few control signals between nodes, we can reduce this false conflict problem by incorporating more busy wires. The number of busy wires then becomes a design time decision based on expected mutex acquisition interleavings, but as the number increases beyond one, has the effect of segmenting the mutex space that they correspond to. For instance, with two busy wires, one could correspond to "odd" mutexes while the other was associated with "even." Then when a node acquires a mutex, it asserts the associated busy wire to indicate that a node may not acquire a mutex associated with it. In this way, we are able to significantly reduce the false conflict rate in the system to improve performance.

Figure 1 shows the results for both naïve implementations of Atomic Coherence (with AtomNaive (busy) corresponding to the configuration with busy wires) with 8,192 mutexes. For the AtomNaive (busy) results, 16 busy wires are used as each benchmark was rather sensitive to the number, with performance being strongly penalized for fewer. We will show in Section IV how we can design a more robust system that is less sensitive to the number of busy wires.

While much better than a single busy wire, performance is still not significantly better in most cases than the current state-of-the-art approach, and indeed much worse for certain applications. This is due to the fact that this approach ignores the significant latency characteristics of an electrical interconnect compared to the fast communication of nanophotonics. As such, performance of applications that are highly sensitive to the latency of atomic operations may experience the significant slowdown that canneal exhibits. The applications evaluated in this graph will be explained in greater detail in section IV-B, but represent GPGPU applications that utilize atomic operations to varying degrees, with their rate of atomic instructions shown on the secondary y-axis as atomic operations per 1,000 instructions.

We can improve on this baseline, by avoiding the sort of steep penalties experienced by some applications, by adapting techniques used in directory-based cache coherence and replacing the ordering achieved through the talking stick with "owner" directories. Whenever a node wishes to acquire a mutex in this system, instead of waiting for a circulating talking stick, it simply requests the mutex from the owner. The owner then responds back to the requester with the mutex if it is available or queues or NACKs the request if it is not. Utilizing this mechanism, we free ourselves from waiting for the talking stick by replacing that wait with a round-trip communication with the owner. This results in an average, uncontended, no congestion mutex acquisition latency reduction from 7.5 cycles to 5 cycles in a 4x4 mesh, assuming each "hop" (traversal through one node to the next) is 1 cycle. The results for this approach are presented in Figure 1 (labeled AtomDir).

III. Fast Exclusive Access to Resources

As can be seen, a simple adaptation of Atomic Coherence that takes into account the limitations of an electrical interconnect allows it to perform reasonably well, and without the potential for significant penalty exhibited by the naïve case. We note in this system, however that the owner of the mutex has a significant amount of information on the acquisition patterns in the network that is nearly sufficient for it to conclude if a requesting node has exclusive access to the data. If the owner were merely capable of retaining the identity of the node that most recently acquired a mutex, it could also unambiguously indicate to the requester whether or not it had potentially stale cached data. Starting with this insight, we develop a more robust system that combines coherence information in the same structure as mutex management and improves latency to acquire a mutex through management distribution and speculative data acquisition. The rest of this section will develop this more robust system through a series of evolutions from Atomic Coherence's directory-inspired baseline.

A. Requester-Side Lookup Tables

To start, we extend the Atomic Coherence, directory-based system to maintain simple Modified/Invalid (MI) coherence on atomic data in the mutex directory itself, doing away with a separate coherence controller altogether. We accomplish this by first extending the mutex status tables present at each directory to also include the identity of the node that most recently acquired the mutex. Therefore, the table increases from a simple boolean per entry to identify the state (available/unavailable) to the boolean plus $\log_2$(# of nodes), resulting in 5 bits per entry in a 16-node system. Further, to prevent aliasing at the requester, we introduce a table for the requester to associate an address with each mutex it acquires. This support is necessary because we assume a similar mapping of addresses to mutexes that Atomic Coherence does, which allows multiple addresses to map to the same mutex. Therefore, acquisition of the same mutex for two separate accesses could erroneously result in the requester trusting its local data when it should not. Structurally, for our simulations, we allow this table to have a power-of-two number of entries equal to or less than the number of mutexes in the system, allowing it to be simply indexed by the appropriate number of address bits. While performance could certainly be improved by utilizing a more sophisticated hash or making the tables associative, we did not find performance to be greatly impacted by the size of these tables (to be discussed in further detail in

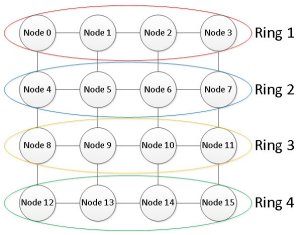

Fig. 2. *Ring Layout.*

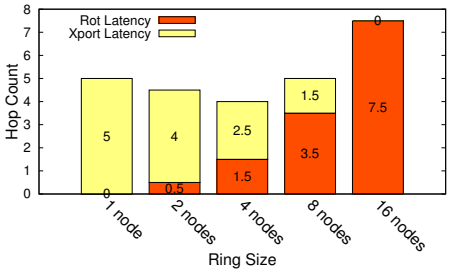

Fig. 3. *4x4 Mesh Average Hop Count.*

Section IV) and leave evaluation of other implementations to future work.

B. Multiple Rings

While replacing the Atomic Coherence controller with simple lookup tables and logic benefits both design and verification complexity and likely improves area and power, the performance results are unaffected. Given the performance of Atomic Coherence with directories, this adaptation alone is likely insufficient to provide a compelling solution to speeding up atomic operations on the GPU. Therefore, we undertake the task of developing mechanisms for improving latency that are motivated by the observation that mutex acquisition latency is optimal when a node is able to satisfy its own mutex requests in the same cycle the requests are generated (i.e., an immediate self-satisfied request). In the AtomDir case, this occurs only when the requester happens to also be the owner of the mutex. In AtomNaive, this occurs whenever the talking stick is present at the requester on the same cycle of the request. For both configurations in a 16-node network, this is a 1 in 16 chance, given a uniform distribution of mutex requests. Therefore, we attempt to increase the number of self-satisfied mutex requests - that is maximized in AtomNaive - without imposing the severe latency overhead of the talking stick. As Figure 3 shows, this challenge is addressed by effectively finding a middle point between the two configurations. These results are obtained by evaluating a system that is a hybrid of the single owner directory of AtomDir and the fully-distributed directory in AtomNaive. In this system, shown as a 4x4 mesh in Figure 2, mutex state is distributed across some number of logical *rings* in the system, where each member of a ring maintains and responds to requests for the same mutexes and nodes in different rings maintain different mutexes. The "Xport Latency" (Transport Latency) numbers for Figure 3 corresponds to the latency of communicating a request to a node that maintains a mutex's state and are generated by iterating over all requester-responder combinations in a system, accumulating the round-trip hop counts between them, and averaging this sum over the possible combinations. "Rot Latency" (Rotational Latency) is simply:

$$\frac{\sum_{i=0}^{N-1} i}{N}$$

where N is the number of nodes in the ring and the result is the average time a node has to wait for the talking stick. Since the growth rate for a talking stick to circulate is slower initially than the reduction in latency to communicate a mutex to a directory as ring sizes increase, these figures show that the selection of a proper ring size can improve latency over either Atomic Coherence configuration.

While we are able to gain some of the benefits of both Atomic Coherence configurations with this mechanism, we also gain some of their shortcomings. For one, we have the same coherency issues between replicated mutex status tables that AtomNaive had. To address this, we once again utilize updates on the existing interconnect along with busy wires. Further, just as in AtomNaive, the talking stick is implied, however since more than one ring can exist in this system, more than one talking stick exists, one for each ring.

Before continuing, it is important to take a moment to discuss the potentially non-intuitive nature related to how waiting for the talking stick to circulate to a responding node is an improvement over merely forwarding the request on to a single owner. This is because FEAR has essentially replaced the Δx plus Δy latency to transmit a request to an owner by replacing one dimension (Δx, nominally) with the stick latency. This is beneficial because average stick latency is half Δx since it is always a one-way trip while Δx is round-trip.

C. Speculative Fetches

In current GPU systems, uncontended atomic operations simply travel to the L2 and are immediately satisfied, while in FEAR they incur the additional mutex acquisition penalty. If frequent enough, even this small penalty can result in overall performance degradation. Therefore, we introduce one final optimization to FEAR to support speculative fetches. With this optimization, memory fetches are issued in parallel with a mutex request to proactively acquire data in the case that mutex acquisition indicates it needs new data. This naturally creates issues if the mutex is contended as a node needs to know if it fetched stale data due to a racing request. To address this issue, we employ a virtual timekeeping system based on the concept of epochs. The epoch concept is similar to a Lamport clock, and provides a low cost way for differentiating mature and immature events in the system [9]. An epoch consists of a fixed number of cycles. At the boundary of each epoch, all responders indicate that their mutex *releases* (i.e., available mutexes in the mutex status table) are mature and all requesters indicate that their outstanding mutex *requests* are stale. Therefore when a responder sends a mutex to a requester, it indicates whether the last release was mature or not. When the requester receives the mutex, if it is mature, it checks whether or not its request is stale. If both the release is

mature and the request is not stale, the requesting node knows that no update could have occurred to the data associated with a mutex between its speculative fetch and mutex acquisition (i.e., both conditions of the release being done in a previous epoch and the request being made in the current epoch are satisfied). Note that in cases where the requesting node can use locally cached data, the speculative fetch data is ignored.

Figure 4 highlights just the sort of problematic situation a speculative fetch can encounter and how FEAR's epoch method ensures correctness. In this example, a requester (Node A) requests a mutex from the nearest eligible responder (Node B) and sends a speculative fetch to the L2 in parallel. Meanwhile, the current holder of the mutex (Node C) completes and sends its writeback and mutex release to the appropriate L2 slice. Piggy-backing the release to the writeback in this way is necessary to prevent race conditions that could allow a subsequent mutex acquisition to get data from the L2 before the previous writeback has been received at the L2. While Node A's mutex request is queued up at Node B, the L2 receives the speculative fetch request and creates a response to Node A with data (that is stale with respect to Node C's update). Once the L2 receives Node C's update, it then sends the mutex release on to its nearest responsible node (Node B in this example, but it could be any other node in the ring). Upon receipt of the mutex, Node B is now free to give it to Node A. To exacerbate the problem, this mutex response actually arrives at Node A prior to the speculative fetch response. The system is now in a situation where Node A has speculative data that is stale, and a mechanism is needed to ensure it gets a fresh copy from the L2 before proceeding. The epoch method fills this role, because no matter where the epoch boundary is placed on the diagram, either the mutex release will be considered immature (Case 1 & 2) or the request will be stale (Case 2), indicating to Node A that it cannot use its speculatively fetched data. Two subtle cases for epoch boundaries are shown, but careful examination reveals that placing a boundary anywhere on the timeline still maintains correct behavior. While ensuring correctness, this method can certainly have a negative impact on performance when an otherwise safe speculative fetch is deemed unsafe. This can be due to excessively long delays in a responder giving a requester a mutex or due to excessively long epochs that indicate a sufficiently old release is immature. As we will show in Section IV, however, proper selection of epoch length can almost completely eliminate any penalty and indeed greatly improve performance over a system that does not allow speculative fetches.

D. Summary

To summarize, FEAR is composed of multiple non-overlapping logical rings imposed on the nodes in a system. All nodes within a ring are responsible for the same subset of the mutex space. When a node wishes to acquire a mutex, it makes a request of the node nearest it that is a member of the set (i.e., ring) responsible for the space the mutex resides. The node that receives this request then responds when:

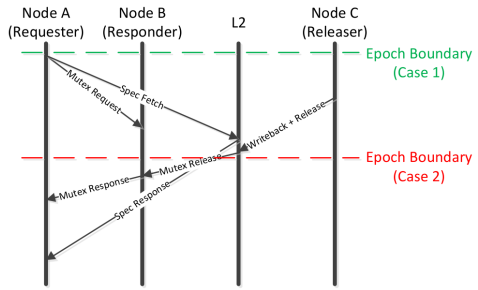

Fig. 4. *Epoch Example.*

1) it is their turn (i.e., they have the "talking stick")
2) they have up-to-date mutex information (as indicated by the appropriate busy wire being clear)
3) their mutex status table indicates that the mutex is available.

Once they grant the mutex to a requester, the responder then generates an update message on the underlying interconnect to inform the other nodes in the ring that the mutex is unavailable and they assert the busy wire. Finally, the responder clears the busy wire when their update message gets back to them (indicating all other nodes have seen the update).

Release of a mutex follows a similar path in that the requester sends a release message to the nearest node in the ordering ring. This nearest node then indicates in their mutex status table that the mutex is now available, generates another update message to pass this information along to the other nodes in the ring, and asserts the busy wire. Once again, the busy wire remains asserted until the generator of the update message receives its update.

IV. Evaluation

We perform two phases of evaluation for FEAR. In the first, we perform sensitivity analysis on the FEAR system's parameters such as the number of mutexes and busy wires, to see how it responds to varying the number of resources available to the system. In the second, we evaluate the relative performance improvement of FEAR's different mechanisms such as speculative fetches, to determine their impact on overall performance.

A. Simulation

We evaluate FEAR on the GPU using GPGPU-Sim [10] to simulate execution of a set of CUDA benchmarks that utilize atomics to varying degrees. GPGPU-Sim is a cycle-accurate GPU simulator that incorporates a network model - which is critical to the evaluation of FEAR - that is itself a cycle-accurate simulator from Dally and Towles [11]. Since GPGPU-Sim does not properly model the serialization and read-modify-write semantics of atomic operations currently, we had to add support for them in the model along with support for features of FEAR. As discussed in Section II, we implement memory-side computation of atomics for our baseline to be consistent with state-of-the-art hardware, while FEAR configurations use shader-side computation to be able to take advantage of cached data.

Core & Memory	
No. Shader Cores	28
No. Mem Interfaces	8
Topology	6x6 Mesh
No. Registers / Core	16K
Shared Mem. / Core	16KB
L1 Cache	32KB 4-way set assoc., 64B lines, LRU
L2 Cache	512KB 8-way set assoc., 64B lines, LRU
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
Interconnect	
Routing Function	Dimension-ordered
Priority	Class-based
No. VCs	2
Buffer size / VC	4

TABLE I
GPGPU-Sim Configuration.

Benchmark	*Best Speedup*	*IPC*	*Atom Rate*
fluidanimate	19.5%	22.4	17.1
hypercolumn	7.19%	163.7	0.008
canneal	3.73%	23.5	0.74
GPUSort	1.07%	60.9	0.67
threadFenceReduction	0.30%	12.1	0.01

TABLE II
Result Summary. (Atomic Rate is per 1K instructions)

B. Benchmarks

For exercising FEAR, we encountered a chicken-and-egg problem of a dearth of applications that utilize atomic operations, due to the fact that GPUs do not perform them well. Therefore, we accumulated a small suite of benchmarks from various sources, with the goal of compiling a set that has a range of atomic instruction reliance. The suite consists of threadFenceReduction from the CUDA SDK [12] and GPUSort from Rodinia [13], along with less traditional GPU benchmarks such as a port of fluidanimate to CUDA [14], a brain learning model (hypercolumn) [15], and our own port of canneal. The ports of fluidanimate and canneal are based upon the implementations in the PARSEC benchmark suite [16]. Together, this suite represents a range of applications that utilize atomics on the GPU only rarely (threadFenceReduction) to quite frequently (fluidanimate). In Table II's final column the rate at which each benchmark executes atomics per 1,000 instructions is shown. The range is quite broad, and generally correlates with reported performance gain (to be discussed in Section IV-D). The notable exception to this trend is hypercolumn. Despite its low atomic rate, hypercolumn actually utilizes atomic operations to perform global synchronization of work units by utilizing them to index into a queue to coordinate work among the blocks. As opposed to the other workloads that largely utilize atomics to protect *potential* racing updates, hypercolumn's explicit serialization and synchronization through these operations make it much more sensitive to atomic performance as reflected in the performance results.

For the rest of this section, we report our results as speedup relative to the baseline - with memory-side computation of atomic operations - that is meant to model a state-of-the-art GPU system. For each benchmark, we report these speedups for complete execution of the kernel that uses them.

C. Configuration

Table I shows the configuration used in GPGPU-Sim to approximate that of a Fermi architecture GPU that performs atomic operations at the L2 interface. As previously mentioned, functionality was added to that of publicly-available GPGPU-Sim in the form of support for atomic operations consistent with assumptions described in Section II along with support for the features of FEAR. Request-response deadlock that could be introduced by FEAR's requests and releases is avoided by prioritizing releases over requests in the class-based, 2 virtual channel interconnect.

D. Overall Performance

Table II summarizes the results obtained for different configurations for each benchmark as speedup over the baseline. For each, the best speedup is first presented followed by characteristics of the application. The characteristics presented are the absolute performance of each benchmark in terms of instructions executed per cycle, followed by the rate at which atomic operations are performed by the application per 1,000 instructions.

As these results show, performance with FEAR is very much related to the function and rate at which the application utilizes atomics, lending credence to the claim that a system utilizing FEAR can improve the performance of applications that utilize atomics in ways and at rates that are poor fits to today's GPUs. Further, we see that FEAR does not, as it should not, adversely affect applications that utilize atomics sparingly, threadFenceReduction being a prime example. While its restrained use of atomic operations makes it a good fit for current hardware, a FEAR-enabled system would allow it to benefit from more liberal use of them.

E. Sensitivity Analysis

Figures 5 through 8 show how sensitive each application is to varying design-time configurable parameters of the system. One feature of note on these graphs is the lack of sensitivity for any application to the size of the lookup tables. Despite the simple, direct mapping of mutexes to it, once it gets above 1K entries (and even less than that for all but canneal) there is only negligible performance impact. Another interesting feature is the fact that, with relatively few busy wires, each application quickly achieves very close to the performance they could achieve if updates were ideal (i.e., all nodes in a ring were instantaneously updated with each acquisition and release). Finally, a feature present in these graphs is one that will become a recurring theme; the influence system parameters

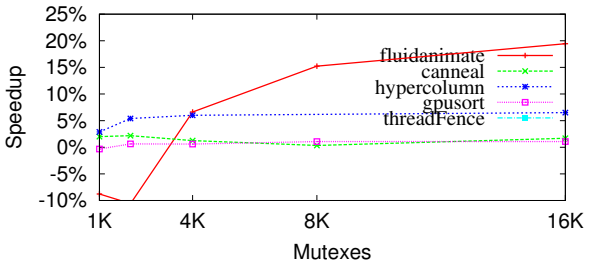

Fig. 5. *Number of Mutexes Sensitivity.*

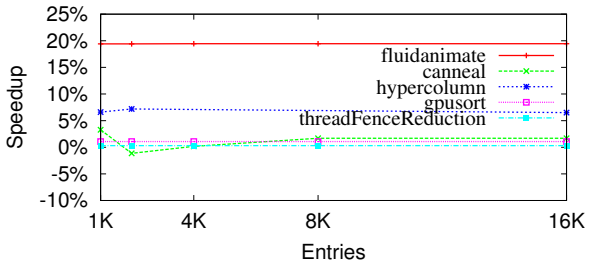

Fig. 6. *Lookup Table Size Sensitivity.*

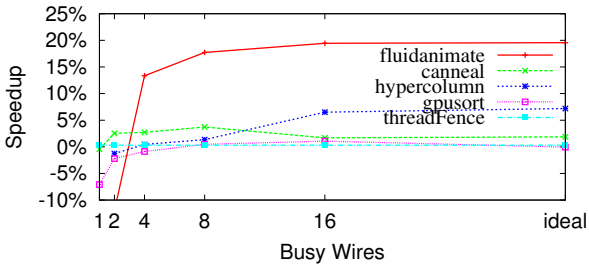

Fig. 7. *Busy Wire Sensitivity.*

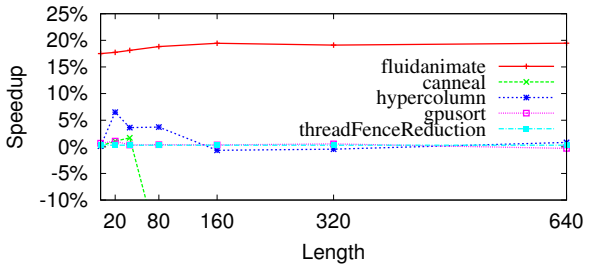

Fig. 8. *Epoch Length Sensitivity.*

have on fluidanimate, with the exception of lookup table size, that highlights the affect FEAR has on non-traditional GPGPU applications that frequently use atomic operations to update global data.

F. Deconstructing FEAR

In order to evaluate the contribution of different components of FEAR, we systematically simulate the system, building features on top of one another. Through this evaluation, we seek to measure the performance contributions of 1) allowing atomic data to be cached, 2) distributing mutex ownership (i.e., creating multiple rings of mutex maintainers), and 3) issuing speculative fetches along with mutex requests. Figure 9 breaks down these elements for each application and are normalized to the standard memory-side atomic computation model that is assumed as the current state-of-the-art. "AtomDir" shows the inherent benefit of being able to cache atomic data, "Topology" shows the benefit of distributing ownership, and "SpecFetch" shows the advantage of issuing speculative memory fetches along with mutex acquisition.

From this figure we not only see that each element of FEAR is important to achievable performance for one or more benchmarks, but also that application behavior can have a significant effect on how important each is. GPUSort for instance, values caching of atomic data over any other as evidenced by its performance in the very simple caching scheme, AtomDir. On the other hand, the increased traffic FEAR introduces with its update messages has a somewhat negative impact on the system that is largely recovered with speculative fetching. Canneal on the other hand requires speculative fetches to achieve significant improvement. Finally, fluidanimate and hypercolumn exhibit monotonically increasing performance for each feature. Given the non-traditional reliance on atomics each of these benchmarks exhibit, these results again highlight FEAR's potential to broaden the application scope of GPGPU.

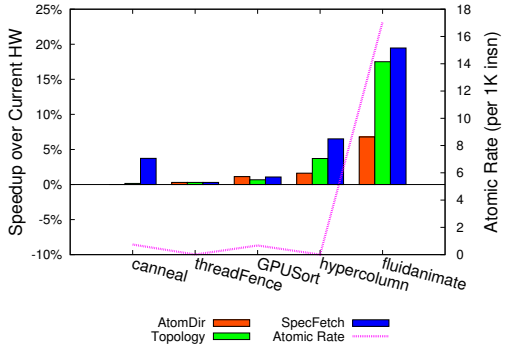

Fig. 9. *FEAR Contributions to Performance.*

G. Power and Area Implications

In order to quantify the power and area impact of FEAR, we used fabmem to evaluate the additional structures needed to support it [17]. These structures consist of the mutex status tables and the requester-side lookup tables. Note that these two structures along with the busy wires, are the only additional costs of FEAR as it does not require tagging memory locations as some other synchronizing systems do. The mutex availability table is replicated across all nodes that participate in mutex acquisition in the system (cores and memory interfaces), while the lookup tables must be present at all nodes that acquire mutexes (cores). Busy wires exist along with the data links between nodes and due to the small number required to closely approximate ideal updates - with respect to the link widths themselves - we did not quantify their power or area effects.

Tables III and IV show both the per-node and chip-wide impact of various sizes of these structures. With our 6x6 mesh system, we estimate chip-wide impact by multiplying the area and power results for the structures by 36 for the mutex availability table and 28 for the lookup tables. This is due to the fact that 8 nodes in the system are memory interfaces and only require a mutex availability table since they

	Per-Node		Chip-Wide		
Mutexes	*Entries*	*Area* (μm^2)	*Power* (mW)	*Area* (μm^2)	*% Area*
2K	512	8,496	3.13	305,857	0.06%
4K	1K	14,270	5.48	566,880	0.11%
8K	2K	28,077	13.86	1,064,264	0.21%
16K	4K	55,868	29.88	2,065,122	0.41%

TABLE III
Mutex Availability Table Impact.

Per-Node		Chip-Wide		
Entries	*Area* (μm^2)	*Power* (mW)	*Area* (μm^2)	*% Area*
2K	121,417	5.54	3,400,000	0.68%
4K	251,465	10.76	7,041,007	1.41%
8K	473,525	20.33	13,258,705	2.65%
16K	988,566	39.36	27,679,839	5.54%

TABLE IV
Requester-Side Lookup Table Impact.

will never make a mutex request and act only as responders. We also assume a 500 mm^2 die based on GPU die reports, though values can be easily scaled for larger or smaller dies. For instance, assuming a more typical CPU die size of 200 mm^2, one would multiply each percentage and power value by 2.5. From these tables, it is clear to see that FEAR imposes trivial overhead, particularly in a state-of-the-art GPU context.

V. Related Work

In the realm of distributed mechanisms for enforcing global synchronization, recent years have provided MP-LOCKS [2], GLocks [3], and TLSync [18], which bear significant similarities to FEAR. These approaches streamline the acquisition of locks in systems that can afford the integration of substantial additional hardware for this purpose. In contrast, FEAR emphasizes low cost, in both complexity and hardware, to minimize the impact on the cost-efficiency of 3D rendering, the primary driver of revenue for the product in the first place.

Another recent proposal is Synchronization State Buffers (SSB) by Zhu, et. al. [19]. This proposal also succeeds at maintaining the lock status of a large memory in a space efficient way, but it does not have the unique topology to improve time to acquisition and contention reduction that FEAR has, nor is it binary compatible as FEAR is.

VI. Conclusion

We have presented FEAR; a low-latency mechanism for acquiring and releasing mutexes in a system of multiple nodes, applied to improve the performance of atomic operations on a GPU. With this mechanism, we find that traditional GPGPU workloads can achieve modest improvement, while certain workloads that have not traditionally been applied to the GPU can achieve substantially better improvement. Further, FEAR contains no GPU-centric elements, lending itself to many application domains outside of GPU atomics. It is general enough that it is easily applied to enforcing mutually exclusive access to any shared resource and could be easily applied to more traditional CPU problems. In fact, our motivating example of Atomic Coherence could leverage FEAR on a traditional CPU in order to realize its benefits without relying on an optical interconnect.

Acknowledgment

We would particularly like to thank Dana Vantrease for her input in developing the mechanisms presented in this paper. This work was supported in part by NSF grant CCF-1116450. In addition, we would like to thank the anonymous reviewers for their insight and feedback.

References

[1] R. Kessler and J. Schwarzmeier, "Cray t3d: A new dimension for cray research," in *Compcon Spring'93, Digest of Papers.* IEEE, 1993, pp. 176–182.

[2] C. Kuo, J. Carter, and R. Kuramkote, "Mp-locks: Replacing h/w synchronization primitives with message passing," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. IEEE, 1999, pp. 284–288.

[3] J. Abellán, J. Fernández, and M. Acacio, "Glocks: Efficient support for highly-contended locks in many-core cmps," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 893–905.

[4] D. Vantrease, M. Lipasti, and N. Binkert, "Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, p. 132143.

[5] W. W. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for GPU architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, p. 296307.

[6] S. Robertson, "CUDA atomics: a practical analysis," http://strobe.cc/cuda_atomics/, Mar. 2010.

[7] D. Patterson, "The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges," *NVIDIA Whitepaper*, 2009.

[8] C. Nvidia, "Programming guide," 2008.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, p. 558565, 1978.

[10] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, p. 163174.

[11] W. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

[12] "CUDA SDK code samples," http://developer.nvidia.com/cuda-cc-sdk-code-samples.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009, p. 4454.

[14] M. Sinclair, H. Duwe, and K. Sankaralingam, "Porting CMP Benchmarks to GPUs," Department of Computer Sciences, The University of Wisconsin-Madison, Tech. Rep., 2011.

[15] A. Nere, A. Hashmi, and M. Lipasti, "Profiling heterogeneous Multi-GPU systems to accelerate cortically inspired learning algorithms," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, p. 906920.

[16] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, 2009.

[17] T. Shah, "FabMem: a multiported RAM and CAM compiler for superscalar design space exploration." Tech. Rep., 2010.

[18] J. Oh, M. Prvulovic, and A. Zajic, "Tlsync: support for multiple fast barriers using on-chip transmission lines," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 105–115.

[19] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 35–45.