Fast Fine-Grained Global Synchronization on GPUs

Kai Wang
University of Texas at Austin
kaiwang@cs.utexas.edu

Don Fussell
University of Texas at Austin
fussell@cs.utexas.edu

Calvin Lin
University of Texas at Austin
lin@cs.utexas.edu

Abstract

This paper extends the reach of General Purpose GPU programming by presenting a software architecture that supports efficient fine-grained synchronization over global memory. The key idea is to transform global synchronization into global communication so that conflicts are serialized at the thread block level. With this structure, the threads within each thread block can synchronize using low latency, high-bandwidth local scratchpad memory. To enable this architecture, we implement a scalable and efficient message passing library.

Using Nvidia GTX 1080 ti GPUs, we evaluate our new software architecture by using it to solve a set of five irregular problems on a variety of workloads. We find that on average, our solutions improve performance over carefully tuned state-of-the-art solutions by 3.6×.

CCS Concepts • **Computer systems organization** → **Single instruction, multiple data**; • **Software and its engineering** → *Mutual exclusion*; *Message passing*.

Keywords GPU; Synchronization; Irregular Workloads

ACM Reference Format:
Kai Wang, Don Fussell, and Calvin Lin. 2019. Fast Fine-Grained Global Synchronization on GPUs. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304055

1 Introduction

GPUs provide significant performance advantages over traditional CPUs, particularly for highly regular data-parallel workloads. Given the potential performance gains, there has been strong interest in applying GPUs to irregular computations. For example, there has been significant work in graph and tree-based algorithms [7, 8, 18, 22, 33], and Nvidia's recently released Volta [29] GPU supports a new SIMT execution model that makes it easier for programmers to use fine-grained synchronization without worrying about SIMT-induced deadlocks [12].

Unfortunately, despite hardware support for atomic instructions, more general forms of fine-grained synchronization—such as those that use fine-grained locks to provide mutual exclusion to global memory updates—are one of the most expensive operations that can be performed on a GPU. Two properties of GPUs contribute to this problem. First, GPUs typically run thousands of threads, which can lead to massive lock contention. Second, unlike CPUs, GPUs do not have coherent caches that could allow lock retries to be confined to local caches, so GPU retries must access the last level cache. With these two properties, lock contention can trigger a massive number of retries that need to touch the last level cache, creating a serious bottleneck. Moreover, fine-grained synchronization is often used for irregular workloads, so these retries often involve non-coalesced accesses that further degrade global memory throughput. Finally, lock retries also increase the latency of lock operations, which can significantly limit performance when many threads contend for a small number of locks. Thus, fine-grained synchronization can cause both throughput and latency problems. As a result, GPU programmers typically avoid fine-grained synchronization, often resorting to less efficient algorithms that admit less parallelism.

In this paper, we present a new software architecture that supports efficient fine-grained synchronization on GPUs by moving lock operations from slow global memory to the GPU's faster word-addressable local scratchpad memories (i.e. shared memory in CUDA or local memory in OpenCL). However, these scratchpad memories are not visible to all threads, so our solution views the GPU chip as a distributed system, where each thread block (*TB*) is a node equipped with a fast private memory (i.e., scratchpad memory), and the multiple nodes share a communication medium (i.e., global memory). Thus, our architecture decouples a baseline GPU kernel into two types of thread blocks that run concurrently on the GPU. *Server TBs* handle updates to *critical section data*[1]. *Client TBs* execute all other aspects of the baseline kernel, with added procedure calls to the

[1]We define critical section data to be the data accessed in a critical section.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04. . . $15.00
https://doi.org/10.1145/3297858.3304055

server TBs, which access the critical sections on behalf of the client TBs.

More specifically, our solution partitions critical section data among server TBs so that each server TB works on an exclusive set of data without conflicts. When a client TB needs to update some critical section data, it sends a message to the appropriate server TB. Threads within a server TB use locks implemented in the scratchpad memory to maintain mutual exclusion when processing the clients' requests. To support this architecture, we implement a software message passing library so that client TBs can efficiently communicate with server TBs via global memory. The benefit of our solution is that clients send these messages just once over global memory—subsequent lock requests and retries are performed by the server TBs without accessing global memory.

From the programmer's perspective, our software architecture is straightforward to implement—in most cases, the code in critical sections is moved to server threads, while the remaining code remains in Client threads.

This paper makes the following contributions:

- We implement a software client-server architecture that supports efficient global synchronization on GPUs by moving lock operations from global memory to fast scratchpad memories.
- We describe a highly optimized software inter-TB message passing system that supports this client-server architecture.
- Using Nvidia GTX 1080 ti GPUs, we evaluate our solution on a set of irregular problems, including graph-based problems and microbenchmarks, showing that our approach yields an average speedup of 3.6× over the best known GPU solutions for these same problems.
- We provide a performance analysis that explains why our solution is profitable.

The remainder of this paper is organized as follows. In Section 2, we place our work in the context of prior work. After describing our solution in Section 3, we present our evaluation methodology in Section 4 and our empirical evaluation in Section 5, before concluding.

2 Related Work

We now describe relevant prior work in the area of fine-grained synchronization for both CPUs and GPUs.

2.1 CPU-Based Solutions

We begin by comparing our solution with previous work that applies similar ideas to CPUs.

The idea of designating one or more threads as servers (or delegates) to handle critical sections has been proposed for multi-core and many-core CPUs [9, 14, 20, 27, 32, 34, 35, 41, 43] The server threads can be chosen either statically [9, 27, 35, 41] or dynamically [14, 20, 32, 34]. While their designs differ, these solutions share the principle of transforming synchronization into communication, that is, to let clients offload (via message passing) the updates for the same data to the same server so that critical section updates can be serialized at the server. Our work is the first to apply similar principles for GPUs. Since GPU architectures differs significantly from that of CPUs, our solution differs from previous work in a number of key ways.

First, CPUs have fewer hardware threads, so previous work uses individual threads as servers, and since conflicts are serialized to a single thread, no further synchronization is needed for processing requests. Since GPUs have large thread or warp counts, our solution uses thread blocks (TBs) as servers so that when requests are processed, threads in the TB synchronize via the fast scratchpad memory.

Second, CPUs have implicit hardware inter-core communication mechanisms in the form of cache coherence and, in many cases, sophisticated on-chip interconnects. So previous solutions employ software message passing systems on top of these mechanisms for a relatively small number of threads. By contrast, our solution must scale to the much larger number of threads on GPUs, which lack such hardware support for inter-SM (inter-TB) communication.

2.2 GPU Solutions

Yilmazer, et al. [46] propose a hardware-accelerated fine-grained lock scheme for GPUs, which adds support for queuing locks in L1 and L2 caches and uses a customized communication protocol to enable faster lock transfer and to reduce lock retries for non-coherent caches. ElTantawy, et al. [13] propose a hardware warp scheduling policy that reduces lock retries by de-prioritizing warps whose threads are spin waiting. In addition, hardware accelerated locks have also been proposed for CPUs [4, 25, 42, 47].

By contrast, our solution does not require hardware modification. Moreover, a rough comparison with their published results (see Section 5.4) suggests that our solution performs as well if not better than previous hardware solutions, likely because our solution solves the problem at higher level by using scratchpad memories for global synchronization.

Li, et al [24] propose a lightweight scratchpad memory lock design in software for older Nvidia GPUs (Fermi and Kepler) that uses software atomics for scratchpad memories. Their solution improves local (i.e. intra-TB)

synchronization performance, whereas our solution solves global (i.e. inter-TB) synchronization problems.

Instead of focusing on performance, others have focused on the programmer productivity aspect of fine-grained synchronization. ElTantawy, et al [12] introduce a compiler and a hardware scheme that avoids SIMT-induced deadlocks. Xu, et al [44] present a lock design that reduces memory storage and uses lock stealing to avoid deadlock. Liu, et al [26] describe a compiler scheme that eliminates redundant locks in the program.

In addition, previous work attempts to improve the performance and programmability of GPUs by supporting transactional memory [10, 11, 15, 16, 37, 45] and by providing memory consistency and memory coherence on GPUs [5, 19, 36, 38–40].

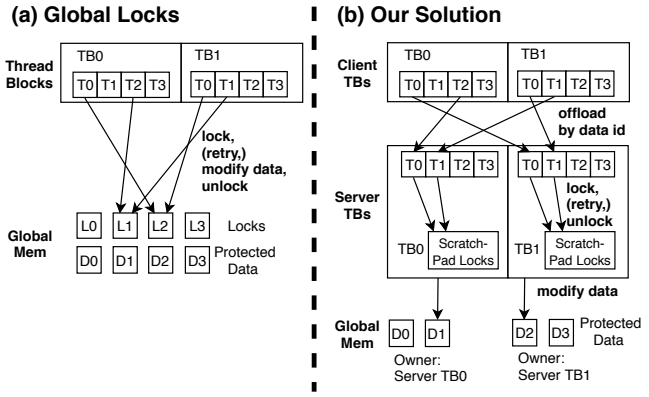

Figure 1. Fine-grained mutual exclusion with **(a)** global locks (baseline) and **(b)** our solution

3 Our Solution

Our solution is a software architecture that uses *synchronization servers* to enable efficient global fine-grained synchronization. The key idea is to replace global lock operations with (1) scratchpad memory lock operations and (2) message passing implemented in global memory. We describe our architecture in Section 3.1.

Our software message passing system, described in Section 3.2, makes efficient use of global memory by employing optimizations that reduce the overhead of message passing operations and that promote coalesced memory accesses.

The *synchronization server* design described in Section 3.1 handles cases where the original kernel code does not have nested lock acquisition. Additional components and optimizations are needed to handle nested locks, and these are described in Section 3.3.

3.1 Synchronization Server Architecture

Consider a baseline kernel that uses locks to protect some data for mutually exclusive updates. In the scheme shown in Figure 1(a), the protected data can be referenced by threads in any TB, so locks are implemented in global memory. The slow global memory becomes a throughput bottleneck for lock retries on conflicts, and it becomes a latency bottleneck for transferring lock ownership among threads.

Our solution separates the original kernel into two kernels—the *client kernel*, which handles the non-critical sections, and the *server kernel*, which executes the critical section on behalf of the client kernel. The two kernels run concurrently; each kernel launches a number of TBs, where the combined TB count of the two kernels does not exceed the maximum occupancy of the GPU. An overview of our scheme is shown in Figure 1(b).

On the client side, client threads can still update arbitrary protected data items, but they do so indirectly by sending messages to server TBs; this is handled by our software message passing system.

On the server side, the ownership of protected data is partitioned among server TBs so that each data item is accessible exclusively through a unique server TB. Client threads choose the appropriate server TBs as destinations of messages based on data IDs. Threads within each server TB service clients' requests in parallel and use scratchpad memory locks to maintain mutual exclusion.

In essence, our solution replaces global lock operations with a two-level mutual exclusion scheme. At the global level, instead of synchronizing among themselves with locks, client threads send messages to server TBs when they need to update protected data items. Since data item IDs have a one-to-one mapping to server TBs, all update requests for the same data are guaranteed to be sent to the same server TB, so mutual exclusion is maintained at the global level, which allows threads within each server TB to synchronize using scratchpad memory locks at the local level. Scratchpad memories are high-bandwidth, low-latency on-chip SRAM that support word-granularity accesses, where accesses waste no bandwidth due to unused cache-line data. Thus, scratchpad memories are ideal for lock accesses and retries with irregular memory accesses.

Compared to a baseline kernel with global lock operations, our solution makes better use of global memory bandwidth. Instead of contending for locks through global memory, contentions are handled in scratchpad memories. In addition, since fine-grained locks are usually used for irregular workloads, lock accesses tend to be non-coalesced memory accesses; by contrast, our message passing system has various optimizations that allows clients and servers to access global memory in a more coalesced manner (as we will show in Section 3.2). Aside from bandwidth benefits, our solution also avoids

transferring lock ownership in the high latency global memory, since locks are now implemented in the low latency scratchpad memories. Because the latency of transferring lock ownership is on the critical path, the reduced latency is beneficial when protected data updates are distributed in a manner that concentrates a large number of updates to a small number of data items.

3.1.1 Implementation

Our solution is implemented at the source code level without modifications to the compiler or the hardware. This section shows how a baseline kernel with global locks (Listing 1) would be modified to use our synchronization server architecture (Listing 2).

Listing 1. Original Kernel with Global Locks

```
1  void kernel(...){
2    // begin critical section
3    bool success = false;
4    do{
5      if(try_lock(data_id)){
6        critical_sec(data_id,arg0,arg1);
7        __threadfence();
8        unlock(data_id);
9        success = true;
10     }
11   }while(!success);
12   // end critical section
13 }
```

Listing 1 shows how a critical section is encapsulated into a function (line 6) and is protected by a try-lock loop. *Data_id*, which can be a single word or a data structure, refers to the data item to be updated, and *arg#* are additional arguments—generated by computations in the non-critical sections—passed to the critical section.

Listing 2 shows how our software architecture uses two new procedures, *send_msg* and *recv_msg* (lines 2-3), to pass messages from a client TB to a server TB, where *dst* in *send_msg* denotes the server TB. The message size (in terms of words) corresponds to the number of arguments of the critical section function.

The *client_kernel* (lines 5-16) corresponds to the original baseline kernel in Listing 1, where the critical section in the try-lock loop has been replaced with message sending to server TBs. The code at line 8 maps offloaded work to server TBs based on data IDs. The mapping interleaves the ownership of data items to server TBs. This fine-grained partitioning provides better load balance among server TBs than a coarse-grained partitioning. However, data_IDs are dynamically generating by client TBs depending on data inputs, so load imbalance can still occur when a large number of threads serialize on relative a small number of data items. Even in these scenarios, the original kernel with global locks suffers much more due to high latency global memory and the interference caused by lock retries.

Listing 2. Pseudocode Code For Our Solution

```
1  //procedure calls for message passing
2  void send_msg(int dst,int data_id,any arg0
       ,...);
3  bool recv_msg(int& data_id,any& arg0,...);
4
5  void client_kernel(...){
6    // execute non-critical section
7    ...
8
9    //map data to server
10   int server_id = data_id % num_server_TB;
11
12   //offload critical section execution
13   send_msg(server_id,data_id,arg0,arg1);
14   // execute non-critical section
15   ...
16 }
17
18 void server_kernel(...){
19   //scratchpad memory locks
20   __shared__ int locks[4096];
21
22   //loop to handle client requests
23   bool terminate = false;
24   while(!terminate){
25     int data_id,arg0,arg1;
26     if(recv_msg(data_id,arg0,arg1)){
27     //received msg, do critical section
28       bool success = false;
29       do{
30         if(try_lock_local(data_id)){
31           critical_sec(data_id,arg0,arg1);
32           __threadfence_block();
33           unlock_local(data_id);
34           success = true;
35         }
36       }while(!success);
37     }
38     terminate=check_termination();
39   }
40 }
```

The *server_kernel* (lines 18-40) executes the critical section on behalf of the clients, so any try-lock loop in the original kernel is now in the server kernel (lines 29-36), which uses locks implemented in scratchpad memories rather than global memory. Since scratchpad memories have limited size, there can be a limited number of locks; multiple data_IDs can be mapped (aliased) to the same lock. On modern Nvidia GPUs, for example, the maximum TB is 1K threads; we use 4K locks per server TB to reduce the chance of unnecessary serializations caused by aliasing.

Server threads execute a loop (lines 19-34) that listens to clients' messages and terminates when all clients are finished. The termination condition is a flag (set by clients) in global memory, which is only checked

periodically by servers; the overhead of checking for termination is negligible because only one thread per TB checks the flag and then informs the other threads of the condition.

Our solution is mostly straightforward for programmers. For most cases, our code in Listing 2 can be used as a template; the programmer needs to insert code for both non-critical sections and critical sections into the indicated places. The server architecture for nested locks (discussed in Section 3.3) is more complex, but a template is also provided for that case.

The maximum occupancy of the GPU (i.e. the number of TBs that can be executed concurrently) can be determined by API calls. Some of those TBs are used by the server kernel, and remaining TBs are used by the client kernel. The ratio of server to client TBs is based on the relative amount of work performed in the critical sections versus the non-critical sections. The programmer is responsible for setting this parameter based on the characteristics of the specific application. This may also require some tuning by the programmer.

3.2 Our Message Passing System

This section describes our software message passing system, where client threads send messages and server threads receive messages. Message passing is achieved by reading and writing buffers that reside in global memory. There is one message buffer per receiver TB, which is shared by all threads of that TB and not accessed by other receiver TBs. To send a message, a thread writes to the specific buffer associated with the receiving TB.

We first describe our basic algorithm for sending and receiving messages, before describing optimizations for efficiently using global memory.

3.2.1 Our Basic Algorithm

Each message buffer is a large array accessed as a circular buffer (we use 4K message entries as a default); our data structure is lock-free and does not use global barriers. The message buffer has the following metadata as shown in Figure 2.

The *write_idx* is atomically incremented by the sender to reserve a buffer index for writes. To determine whether the reserved index is free to write, the sender checks the *read_idx*, which is atomically incremented by the receiver after reads. The *bit-mask* has one bit corresponding to each buffer location; it is set by the sender after the message data has been successfully written to L2 (i.e. after the memory barrier[2]), and it is checked by the receiver to find available messages for reads. The

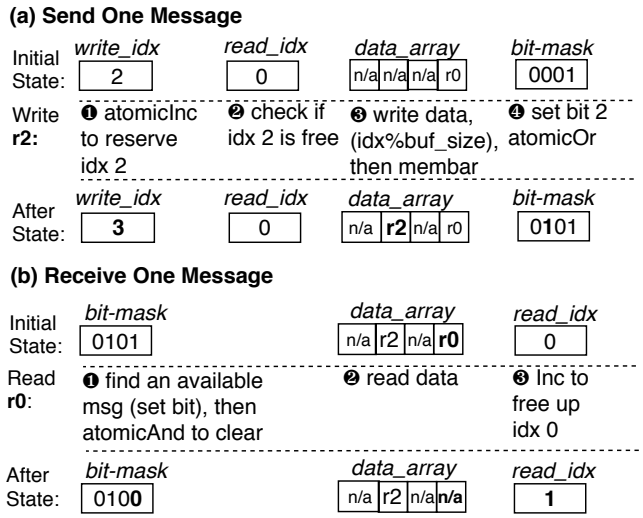

Figure 2. The data structure of a single message buffer and the basic algorithm for reading and writing

.

bitmask is needed because concurrent sender threads may finish writing out-of-order, e.g. in the figure, buffer location 1 is reserved before location 2, but location 2 is written before location 1, so the receiver must be able to determine whether a specific location is valid.

3.2.2 Our Optimized Algorithm

Each message buffer is concurrently accessed by tens of thousands of threads, so our system's buffer accesses need to be highly efficient. Unfortunately, our basic algorithm is inefficient, because threads access the message buffer individually, so each message send or receive incurs the overhead of accessing metadata in global memory (e.g. bit-mask, etc.); moreover, the lanes of a sender warp may have different destination buffers (receiver TBs), and the lanes of the receiver warp might not read consecutive buffer locations, so memory accesses can be non-coalesced.

Our optimized solution amortizes the cost of global memory accesses by aggregating both messages and their metadata. Senders aggregate messages by collecting them in local buffers residing in scratchpad memory before being written to global message buffers in bulk. Receivers aggregate message passing metadata access by having a single warp, known as a *leader warp*, manipulate the metadata on behalf of the other warps, which we refer to as the *follower warps*.

For senders, each TB has a set of small message buffers in the scratchpad memory, with each local buffer corresponding to one receiver TB. Messages from multiple warps are aggregated in local message buffers before being written to global message buffers, so metadata overhead is amortized and global memory accesses are

[2]Memory barriers are supported in CUDA by calls to __threadfence(), and they are supported in OpenCL by calls to mem_fence(CLK_GLOBAL_MEM_FENCE).

typically coalesced. In addition, the metadata overhead of accessing *read_idx* (Figure 2(a)) can be further reduced by keeping a local scratchpad copy and updating it lazily. The design is described in Section 3.2.4.

For receivers, with one warp-granularity global memory access, the leader warp reads multiple bit-masks to find available messages and then uses a set of *assignment buffers* in local scratchpad memory to assign these messages to follower warps. The leader warp only reads the bit-mask, while the actual message data is read by follower warps. The messages assigned to each follower warp are stored in consecutive buffer locations, so accesses to message data are coalesced memory accesses. Since the warp can read 32 bit-masks which each represent 32 buffer locations, the leader warp can with one memory access read the bit-mask of up to 1024 messages. The design is described in Section 3.2.3.

Benefits Over Global Locking. Our message passing system makes much more efficient use of global memory than codes that access global locks.[3] The key insight is that global lock operations must directly access specific lock variables that are spread throughout the address space, so memory accesses are inherently non-coalesced. By contrast, because our solution handles locking indirectly and locally in the server TBs' scratchpad memories, a client's send messages are not bound to specific global memory addresses; therefore, these messages can be placed consecutively in circular buffers, allowing our optimized solution to perform coalesced reads and writes of global memory.

3.2.3 Receiver Design

For receiving messages, the heavy lifting is performed by the leader warp. Figure 3(a) shows the operations performed by the leader warp in one iteration. In *step a1*, 4 lanes of the leader warp read 4 bit-masks (words) from global memory. Here, *head_idx* denotes the current progress of bit-mask read. The leader warp only handles consecutive available messages (i.e. consecutive 1s) starting from head_idx; in the figure, the last message to handle is at location 12. This requirement simplifies buffer management for two reason. First, the leader warp does not have to backtrack to check previously unavailable messages. Second, when assigning messages to follower warps, the set of valid messages can be simply represented as a range starting from the head_idx.

To derive the total number of consecutive available messages, the leader warp performs a reduction on bit-masks read by individual lanes, which is achieved

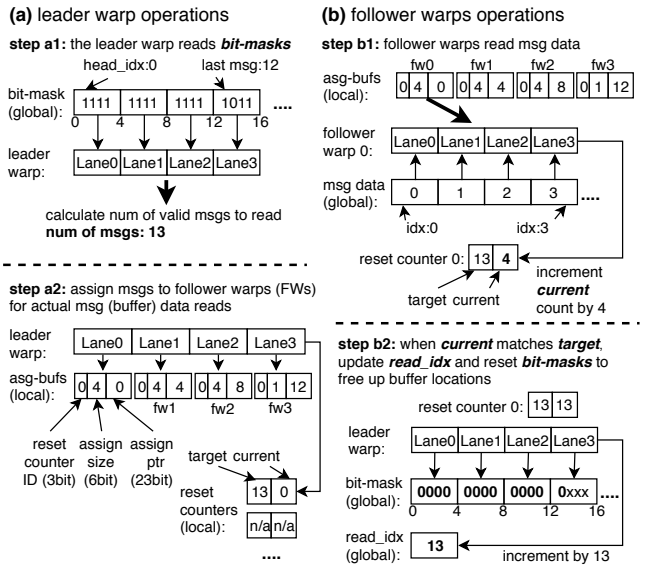

Figure 3. Operations performed by **(a)** the leader and **(b)** the follower warps when reading messages—assuming 4 bits per bit-mask words, 4 lanes per warp, and 4 follower warps per TB

.

by two intra-warp communication functions—*vote_ballot* and *shuffle_idx*, which are register-like operations supported by hardware. Vote_ballot accepts a 0 or 1 bit (ballot) from each lane as its argument and returns the entire warp's vote as a bit-vector (all lanes get the same bit-vector), so each lane knows how other lanes have voted. For our purpose, any lane that has a 0 in its bit-mask (i.e. unavailable messages) votes **1**; otherwise it votes **0**. In the figure, lane 3 votes 1, and other lanes vote 0, so we know that the sequence of consecutive 1s ends somewhere in lane 3's bit-mask. Next, we just need to know exact bit position of the first 0 in lane 3's bit-mask, and this is done by using shuffle_idx, which allows all lanes to read a register value from a specific lane. With this information, the warp determines that there are 13 consecutive valid messages starting from the head_idx (at 0).

In *step a2*, these valid messages are assigned to follower warps via *assignment buffers*, which are packed 32-bit words in scratchpad memory. Each assignment buffer corresponds to one follower warp. The *assign_idx* and the *assign_size* fields indicate the starting location and the number of messages to read. As shown in *step b1* (Figure 3(b)), each follower warp reads message data according to its assignment buffer, where assign_size, the number of assigned messages for each follower warp, has a maximum value of the GPU's warp size (e.g. 32). To wait for work from the leader warp, the follower warps poll the assignment buffers. Similarly, when follower warps are busy, the leader warp polls assignment

[3]Of course, our overall software architecture has the added advantage that it does not perform lock retries in global memory.

buffers to wait for available follower warps. Both types of polling only generate accesses to scratchpad memory.

Once messages are read, their buffer locations are freed by resetting the bit-masks and updating the read_idx (see Figure 2(b)). *Reset_counters* in scratchpad memory are used to aggregate these global memory metadata operations, so the overhead can be paid once for a large number of messages. *Step a2* shows that when assigning messages, the leader warp initializes the *target* field of a reset_counter to the number of message assigned and the *current* field to 0. In *step b1*, the follower warps increment the current field atomically after reading messages; once the target field matches the current field, the follower warps have finished reading a range of consecutive messages, so the leader warp resets multiple bit-mask words with one warp-granularity memory access and then updates the read_idx.

After message assignment, the leader warp proceeds to the next iteration without waiting for the follower warps to finish reading, thereby allowing the leader warp and follower warps to execute concurrently. In the next iteration, the head_idx (*step 1a*) is set to 13. *Step 2b*, shows that each of the multiple reset counters can be used for different iterations, and the *reset_counter_ID* field of the assignment buffers indicates the reset counter that the follower warps should use.

Design Benefits. Our receiver design has several benefits. First, since consecutive messages are assigned to each follower warp, memory accesses for message data reads are coalesced. Second, on a GPU with 32 lanes per warp, the metadata overhead of bit-mask read, bit-mask reset, and read_idx update are amortized across as many as 1K messages (32 lanes of the leader warp can each process 32 messages per iteration). The actual number processed per iteration depends on message availability. Since we use relatively large message buffers (4K entries), and since the buffer free is not on the critical path, we can accumulate each reset counter for multiple iterations (not shown) so that the overhead of bit-mask reset and read_idx update are guaranteed to be amortized across a large number of messages. Third, when senders send messages to a receiver TB infrequently, the use of a leader warp reduces the amount of useless polling of the bit-mask, since the bit-mask is polled only by the leader warp rather than all of threads in the TB.

3.2.4 Sender Design

To reduce global memory metadata overhead and to promote coalesced memory accesses, each sender TB is equipped with a set of small message buffers in the scratchpad memory

Figure 4 shows how the message write operations use local message buffers; the 4 local buffers correspond to 4 destination receiver TBs. Local message buffers have data structures similar to those in global message buffers—the default size is 64 entries.

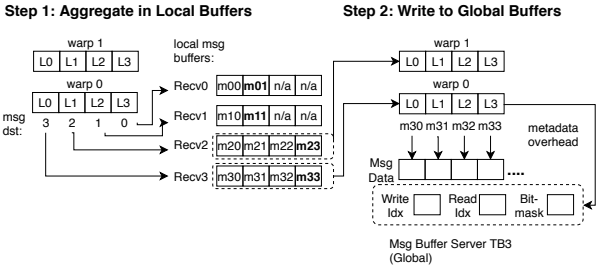

Figure 4. Local Message Buffers on a Sender TB—assuming 4 receiver TBs and 4 lanes per warp

.

A message send is a two-step process. In step 1, sender warps aggregate messages in the local buffers; each lane of a warp writes its message to the appropriate local buffer based on the destination. In step 2, after writing its own messages to local buffers, a sender warp chooses one local buffer from which to copy its messages to the global buffer. Sender warps select local buffers in a round-robin fashion by atomically incrementing a pointer in scratchpad memory. The local buffers coalesce messages from multiple warps according to their destinations. Thus, buffer writes to global memory are coalesced memory accesses, and metadata overheads are now amortized over multiple messages.

To further optimize the overhead of checking read_idx, recall that read_idx is incremented by the receiver after reads, so senders check read_idx to determine whether buffer locations can be reused. Since senders do not always need up-to-date value of read_idx, each sender TB keeps a local copy of read_idx for each receiver TB, and each sender TB updates the local copy with the global value only when necessary (see Figure 5).

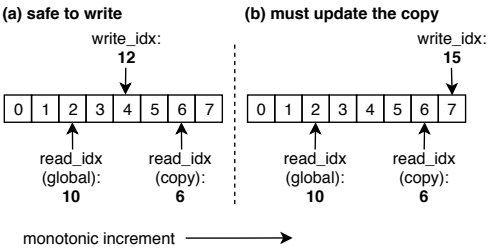

Figure 5. Read_idx checking using local copy

.

In Figure 5(a), the sender has reserved index 12 to write one message. The local copy of read_idx is not up-to-date, but it still guarantees that the receiver has finished reading indices 0 to 5, so those buffer locations

can be reused by senders for indices 8 to 13; index 12 is safe to write, and there is no need to access the global value. In Figure 5(b), the sender has reserved index 15, and it can no longer determine whether the buffer location can be reused by the local copy, so the local copy must be updated with the global value.

Since that we use relatively large message buffers, this lazy update shields global memory from most of the read_idx checking. Furthermore the local copies are shared by all threads of the TB, so if one thread updates a copy, all threads will see the updated value.

3.3 Handling Nested Locks

Listing 3 shows the original kernel code with two nested locks, where the critical section manipulates two data items, so a thread must acquire the locks for both data items before entering the critical section.

Listing 3. Original Kernel With Two Nested Lock

```
1  // begin critical section
2
3  //data_id1 < data_id2
4  bool success1 = false;
5  bool success2 = false;
6  do{
7    if(!success1){
8      if(try_lock(data_id1))
9        success1 = true; //acquired 1st lock
10   }
11
12   if(success1){ // acquire 2nd lock
13     if(try_lock(data_id2)){
14       critical_sec(data_id1,data_id2,...);
15       __threadfence();
16       unlock(data_id1);
17       unlock(data_id2);
18       success2=true;
19     }
20   }
21 }while(!(success1 && success2));
22 // end critical section
```

Just as with non-nested locks, our solution partitions data items among server TBs so that lock operations can be handled in scratchpad memories. As shown in Figure 6, server TB0 has ownership of data D0 and D1, so TB0 has exclusive access to the associated locks (L0 and L1) in the scratchpad memory. TB1 similarly has exclusive access to L2 and L3. The client's offload request now contains two data items (D0 and D2) belonging to two different server TBs.

Our solution lets a client send an offload message to the server TB that owns the first lock (TB0), which then acquires the remote lock (L2) from the other server TB (TB1) by sending to TB1 a *request message* for L2, which will try to lock L2 locally. Once successful, TB1 sends a *reply message* back to TB0, temporarily granting the ownership of lock L2 to TB0 and preventing other

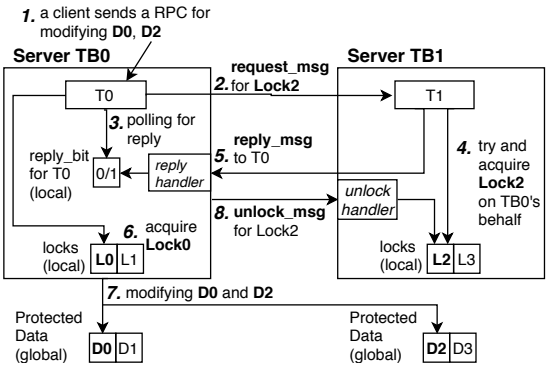

Figure 6. Synchronization Server for Two Nested Lock—operations for handling an offloaded request from client that involves two server TBs

requests from modifying D2. Upon receiving the reply message, TB0 acquires L0 locally and executes the critical section. Once finished, TB0 sends an *unlock message* back to TB1 to unlock L2. As with non-nested locks, our solution handles lock retries in scratchpad memories, so server TBs send messages only once over global memory.

To avoid deadlocks, we use a lock hierarchy prevent circular dependences. In particular, the lock hierarchy is defined by the *global ID* of the locks to which they are mapped. Global_ID uniquely identifies a local lock, where $global\_ID = server\_ID \times locks\_per\_server + local\_ID$. Furthermore, we avoid deadlock caused by insufficient buffer space availability by using different message channels for different message types, similar to the idea of using virtual channels to prevent protocol-level deadlocks.

The reply and unlock messages are on the critical path of lock acquisition and release, so to reduce their latency, we replace the receiver's leader warp (see Section 3.2.3) with a *reply handler warp* and an *unlock handler warp*. These two warps handle metadata in the same manner as the leader warp, but instead of assigning messages to follower warps, they read message data directly and then perform their associated action—setting reply_-bits or resetting local locks—directly, thereby reducing latency. This modification is possible because reply and unlock are simple tasks that are guaranteed to succeed without retry, so follower warps are not needed. At the sender side, we do not aggregate reply and unlock messages in local buffers (see Section 3.2.4), since local buffering increases latency. Instead, the two types of messages are sent directly to global buffers.

[4] *GPU Max Clock rate* reported by devicequery on our GPU

Compute Capability	sm_61	Scratchpad Mem per SM	96KB
Shader Clock Rate[4]	1.68 GHz	Max Scratchpad Mem per TB	48KB
SM Count	28	L2 Size	2.75MB
Max Threads Per SM	2048	L2 Cache Line Size	128 Byte

Table 1. GTX 1080 ti Specifications

4 Evaluation Methodology

Before describing the experimental evaluation of our solution in the next section, we first describe our empirical methodology.

We evaluate our solution on an Nvidia GTX 1080 ti GPU (Pascal, GP102) [28, 31] using CUDA toolkit version 9.2 with driver 410.57. Table 1 summarizes this hardware platform.

To gather kernel execution statistics, such L2 misses and DRAM traffic, we use the Nvidia Profiler (nvprof) [30] that comes with CUDA 9.2. This profiler records statistics by replaying kernel executions and periodically accessing hardware performance counters on the GPU.

4.1 Benchmarks and Inputs

To evaluate our solution, we use three state-of-art GPU implementations of irregular algorithms, which have been shown to compare favorably against CPU implementations [18, 22, 33], and we use two microbenchmarks. which have been used in previous work on fine-grained locking [12, 13, 46] and transactional memory [10, 15, 16, 37, 45] on GPUs. The two microbenchmarks represent commonly used lock patterns for workloads that manipulate irregular data structures, such as graphs and trees.

We now describe each of our five baseline benchmark programs.

Hash Table (HT). HT is a microbenchmark in which threads randomly select an element from a pool of elements and inserts the element into a hash table, where each hash table entry is a linked list. Locks are used to provide mutual exclusion on entry updates. We use a large hash table with collision factors of 256, 1K, 32K, and 128K, where the collision factor is the pool size. Thus, smaller collision factors lead to a larger number of lock conflicts.

Bank Account (ATM). ATM is a microbenchmark with two nested locks. Each thread performs a transaction that withdraw funds from one account and deposits them into a second account. A lock is associated with each account, so each thread acquires two locks to perform a transaction. Similar to HT, threads randomly choose the source and destination accounts with collision factors 256, 1K, 32K, and 128K.

Minimum Spanning Tree (MST). MST finds a spanning tree that connects all vertices of a graph with minimum weight. Our baseline is a GPU implementation of Boruvka's algorithm from the newly released LonestarGPU 3.0 benchmark suite [7, 33]. Each thread works on a vertex of the graph and updates a data structure called a component. Since multiple vertices can be mapped to the same component, fine-grained locks are used to provide mutual exclusion for component updates. We use as inputs the three largest graphs from the benchmark suite—rmat22 (power law), USA-road-d.USA (high diameter), and r4-2e23 (random).

Stochastic Gradient Descent (SGD). SGD works on bipartite graphs, e.g., a movie rating graph with some vertices representing movies, with other vertices representing users, and with weighted edges between a user and a movie representing a rating. SGD predicts missing edges (ratings) based on existing edges. We use Kaleem, et al's [22] edge-lock implementation, where edges are assigned to threads and two nested locks are used to guard movies and users. We use three commercial inputs—Netflix (NF) [2, 6], reuters (RT) [3, 23], and movie-lens 10M (ML) [1, 17].

Maxflow (MF). MF is a push-relabel algorithm that finds the maximum flow of a weighted graph, where edge weights represent network capacity. The algorithm iteratively pushes excess capacity at a node to an eligible neighboring node; if a node's excess capacity cannot be moved, then the vertex is relabeled.

He, et al [18] present a GPU implementation that works on general graphs. The implementation is topological, which means that all nodes are checked at each iteration, regardless of whether they have excess. Nodes are parallelized and fixed to threads, so the algorithm does not need locks or worklists, but it performs a considerable amount of useless work.

Our baseline implementation is based on He's implementation but uses a worklist, where threads only push neighbor nodes with useful work to the worklist. However, our algorithm needs locks to guard each node, since multiple threads can push the same neighbor node onto the worklist, so multiple threads may work on the same node.

Because the behavior of these benchmark programs can vary greatly depending on their inputs, we use a variety of inputs for each benchmark. Our inputs are mesh graphs (2k×2k, 4k×4k, and 8k×4k) generated by a Washington generator [21]. Compared to He, et al's algorithm, our baseline implementation performs worse on smaller inputs and better on larger inputs. The speedups over the original are 0.78× (2k×2k), 0.98× (4k×4k), and 2.05× (8k×4k), with a 1.27× mean.

Besides the use of locks, another major inefficiency of our baseline implementation is the worklist. On GPUs, current lock-free worklist implementations use double buffering, where threads read from one buffer and write to another. When the read buffer is empty, the two buffers are swapped at a global barrier (implemented as a GPU-side kernel launch), and the process repeats. Besides its overhead, the global barrier limits concurrency. For some algorithms, this may not be a problem provided that a large number of work-items are processed each round. But with this algorithm, especially for later more sparse phases, each round does not have enough work-items for double buffering to be efficient.

For this particular benchmark, our server implementation both replaces the worklist and handles mutual exclusion. All TBs are both servers and clients; a TB accepts work (node update requests) and then sends work to others. The message passing system connecting the TBs essentially performs global work redistribution and buffering, which is similar to the functionality of a worklist. Since mutual exclusion already requires message passing, the worklist overhead can be regarded as free. More importantly, our message passing system is both lock-free and asynchronous, so it has more concurrency and no barrier overhead.

5 Evaluation

This section evaluates our solution by first presenting speedups over the current state-of-the-art and then examining in detail the causes of the performance gap.

5.1 Performance

Figure 7 shows the speedup of our solution over state-of-the-art baseline implementations of each of our benchmarks. Our solution achieves a mean speedup of 3.6× by addressing the *throughput* and *latency* bottlenecks of lock operations.

When compared with the baselines, our solution improves throughput by shielding global memory from non-coalesced locks operations, which are instead performed in scratchpad memories and supported by an efficient message passing system. The end result is a reduction in L2 and DRAM traffic, which be will examined in details in Section 5.2.

In addition to reduced bandwidth waste, our solution also improves *lock transfer latency*, which is the difference between the time at which one thread releases a lock and another thread acquires that lock. When threads contend on a relatively small number of locks, lock transfer latency significantly affects performance, which we find to be the case for ATM-256, ATM-1k, HT-256, HT-1k, MST-rmat and MST-2e23. The baseline performs lock transfers via global memory, so its transfer latency

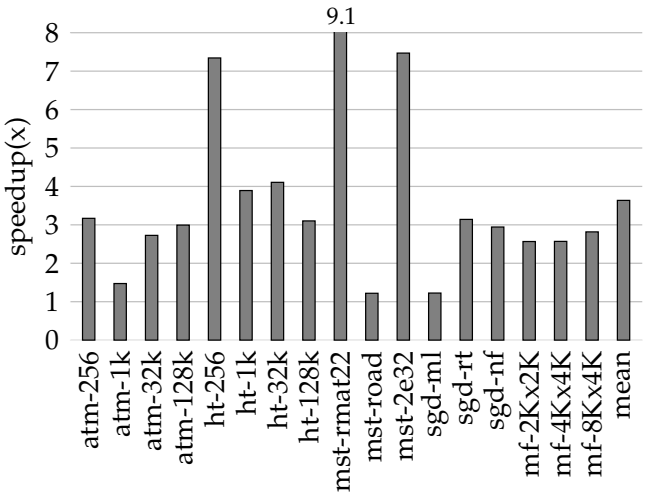

Figure 7. Speedup of our solution over the state-of-the-art

is affected both by the global memory latency and by the interference caused by lock retries. For HT and MST, our solution performs the acquire, retry, and release operations entirely in scratchpad memories, so we see significant performance improvements for these inputs. ATM uses two nested locks, where an offloaded request requires a local lock and a remote lock. For our ATM solution, global memory latency is on the critical path of handling the remote lock, but lock retries are still handled entirely in scratchpad memories, so global memory latency is not affected by lock retries. Therefore, our solution achieves lower latency than the baseline. We describe in detail the latency benefits of our solution in Section 5.3.

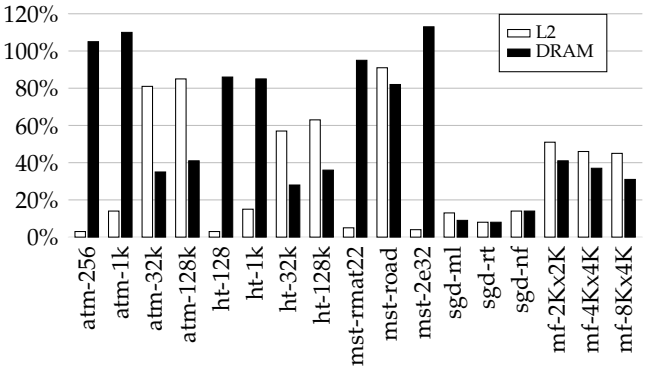

Figure 8. L2 and DRAM traffic of our solution as a percentage (%) of the baseline—The L2 traffic is the total cache-line accesses of global loads and stores and atomics, including misses to DRAM. The DRAM traffic includes both reads and writes.

5.2 L2 and DRAM Efficiency

This section shows that our solution alleviates bandwidth bottleneck by reducing global memory traffic. Figure 8 shows the L2 and the DRAM traffic of our solution as a percentage of the baseline. The traffic includes overhead due to non-coalesced accesses (i.e. unused words in cache lines). In most cases, our solution significantly reduces both L2 and DRAM traffic.

While our scheme generally reduces both DRAM and L2 traffic, in most cases, our DRAM traffic reduction over the baseline is greater than the L2 traffic reduction because our message passing system enjoys locality in the L2 cache. Each receiver buffer has just one *write_idx* and *read_idx*, so access to those pointers will cause L2 traffic but most likely result in a cache hit. Furthermore, for message data writes, when multiple senders make non-coalesced writes to the same buffer (same receiver) at similar times, they are likely to write to adjacent locations of the buffer, since the (circular) buffers are reserved incrementally for writing. So individually, each sender causes non-coalesced L2 accesses, but the cache lines of the buffer are evicted to DRAM and are read by the receiver with coalesced messages.

ht 32k		ht 128k		atm 32k		atm 128k	
L2	DRAM	L2	DRAM	L2	DRAM	L2	DRAM
21.45%	17.75%	23.66%	10.32%	76.1%	26.19%	73.51%	14.00%

Table 2. The net global memory traffic of our solution as percentage (%) of the baseline solution, which uses global fine-grained locks

Figure 8 shows total traffic generated by the entire kernel. The profiler can only measure entire kernel execution. To directly compare message passing against locking for the two microbenchmarks, we replace the critical section that accesses memory with arithmetic loops of similar latency, without fundamentally changing the execution characteristics of the workload. This allows us to extract the net L2 and DRAM traffic, which is shown in Table 2.

5.3 Lock Transfer Latency

To understand our results, we first identify the sensitivity of each benchmark-input combination to lock transfer latency. We conduct an experiment that adds an arithmetic loop to the baseline's critical section and measures the increase in execution time. Thus, this loop simulates increased lock transfer latency because it delays unlock operations without generating additional memory traffic.

Figure 9 shows the empirical results with different loop sizes. A 1× loop has approximately a 1K cycle latency, and this latency increases proportionally with loop size. The arithmetic instructions can overlap with

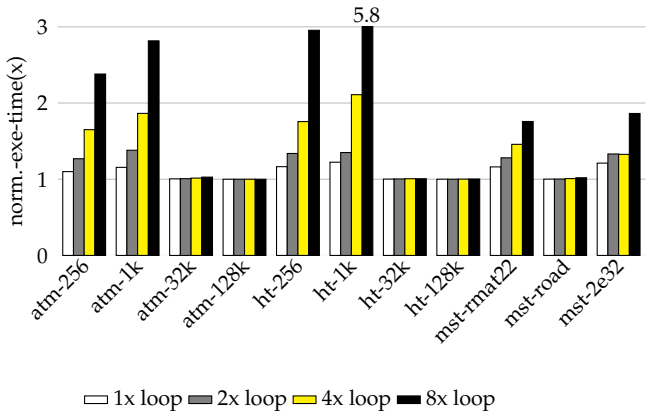

Figure 9. Execution time of the baseline with additional latency by arithmetic loop inside critical section, normalized to no additional latency—this figure demonstrates latency sensitivity

memory accesses, so for small loops, the overall critical section latency has little or no increase. On the other hand, very large loops will increase execution time regardless of the workload being used.

We see that under **moderate** latency increase, ATM-256, ATM-1k, HT-256, HT-1k, MST-ramt, and MST-2e23 noticeably increase execution time, while the others do not.

	Latency (Cycle)	
	Baseline	Our Solution
atm_256	24269	8292
atm_1k	18234	13831
ht_256	2192	248
ht_1k	2679	863
mst_rmat	3601	490
mst_r4	3537	533
	Latency (Norm.)	**Run Time (Norm.)**
	Our Solution	Our Solution
atm_256	0.34	0.31
atm_1k	0.75	0.68
ht_256	0.11	0.13
ht_1k	0.32	0.25
mst_rmat	0.14	0.11
mst_r4	0.15	0.13

Table 3. Latency and Total Execution Time

Table 3 shows that execution time strongly correlates with latency. In particular, the table shows average unlock and reacquisition latency, including measurement overhead, where latency is measured using the %globaltimer register, which is a nanosecond hardware timer that has a consistent time for all SMs.

For HT and MST, our solution handles locks and unlocks entirely through server TBs that access scratchpad memory. By contrast, the baseline must go through higher latency global memory, which is also affected by

memory contention caused by lock retries. Therefore, our solutions have significantly lower latencies. For our ATM solution, global memory is used to send messages to acquire and release locks, but the critical path operations in global memory are not inhibited by lock retries, and some of the lock transfers are handled in scratchpad memories, so our solution has lower latency than the baseline.

5.4 Comparison Against Previous Solutions

In this section, we compare our solution with two hardware solutions for improving the performance of global memory lock operations. *HQL* [46] embeds hardware locks in the L1 and L2 caches, where cache tag entries act as queue locks. A cache coherence-like protocol for lock operations between L1 and L2 is used. *BOWS* [13] is a warp scheduler that reduces retry traffic by deprioritizing warps that are spinning on locks.

Because *HQL* and BOWS are hardware solutions evaluated on simulators, a direct comparison is impractical, but to get a rough sense of how our solution compares, Table 4 shows—for common benchmarks—the speedup of our solution along with those from published results of *HQL* and *BOWS*. *Because of the numerous methodological and implementation differences, these numbers should be taken with a large grain of salt.*

	Speedup over Baselines				
	HT-32	HT-128	HT-512	HT-1K	ATM-1K
HQL	10x	1.6x	1.1x	0.9x	
BOWS				1.3x	1.8x
Ours	18.3x	8.9x	4.0x	3.9x	1.5x

Table 4. Speedup over respective baselines—For HQL, the results are from Figure 12 of the paper [46]; the baseline is a simulated Radeon HD 5870 GPU. For BOWS, the results are from Figure 15 of the paper [13]; the baseline is a simulated GTX 1080ti. The HQL paper only provides results for the HT microbenchmark, and the BOWS paper only provides results for HT-1K and ATM-1K; unavailable results are left blank in the table.

At low lock count, HQL achieves speedup for HT because lock transfers are partially handled in the L1, which decreases latency compared to the baseline; the effect is similar to the use of scratchpad memories in our solution. However, hardware locks are bound to limited cache resources, namely, the cache capacity and the number of tags, so HQL's performance benefit decreases rapidly as the number of locks increases; at 1K, HQL degrades performance. Since our solution is implemented in software, it does not have these limitations, so our solution achieves much higher speedups and does not see performance degradation at high lock counts.

BOWS improves performance by reducing lock retries. However, global synchronization is still handled in L2 and DRAM, which limits the performance gain (particularly for HT) compared to our solution, which implements locks in scratchpad memories.

6 Conclusions

A common research trend is to add hardware support to make GPUs more efficient and effective for irregular computations. In this paper, we have shown that in one respect, GPUs are already much more efficient than has been commonly recognized: With the right programming model, existing GPU hardware can support efficient fine-grained synchronization.

The main idea is to greatly reduce the use of slow global memory by distributing work to the faster local scratchpad memories. In particular, our solution uses global memory to distribute work to server TBs, each associated with a single scratchpad memory. Lock retries are then handled at the scratchpad memories, which are more efficient than global memory, particularly for non-coalesced memory accesses. To support this solution, we have implemented an efficient software message passing system built on top of global memory.

This new software architecture is straightforward for programmers. The main task is to decompose the critical sections from the rest of the code. For example, instead of writing a single kernel with a critical section, programmers implement two kernels, one representing client threads that execute the non-critical sections and make non-blocking procedure calls to the servers, and the other representing server threads that execute the critical sections on behalf of the clients.

We have evaluated our solution on five irregular benchmarks, each with three different inputs. On Nvidia GTX 1080 ti GPUs, our solutions are on average 3.6× faster than the previous best state-of-the-art solutions for each problem.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work was funded in part by DARPA contract # FA8750-16-2-0004, NSF Grant CCF-1823546, and gifts from Intel Corporation and Huawei Technologies.

References

[1] 2017. MovieLens 10M network dataset – KONECT. http://konect.uni-koblenz.de/networks/movielens-10m_rating

[2] 2017. Netflix network dataset – KONECT. http://konect.uni-koblenz.de/networks/netflix

[3] 2017. Reuters network dataset – KONECT. http://konect.uni-koblenz.de/networks/reuters

[4] J. L. Abelln, J. Fernndez, and M. E. Acacio. 2011. GLocks: Efficient Support for Highly-Contended Locks in Many-Core CMPs. In *2011 IEEE International Parallel Distributed Processing Symposium*. 893–905.

[5] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy Release Consistency for GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 26, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195669

[6] James Bennett and Stan Lanning. 2007. The Netflix Prize. In *Proc. KDD Cup*. 3–6.

[7] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '12)*. IEEE Computer Society, Washington, DC, USA, 141–151. https://doi.org/10.1109/IISWC.2012.6402918

[8] Martin Burtscher and Keshav Pingali. 2011. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, Chapter 6, An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm, 75–92.

[9] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (OPODIS 2013)*. Springer-Verlag, Berlin, Heidelberg, 83–97. https://doi.org/10.1007/978-3-319-03850-6_7

[10] S. Chen and L. Peng. 2016. Efficient GPU hardware transactional memory through early conflict resolution. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 274–284. https://doi.org/10.1109/HPCA.2016.7446071

[11] S. Chen, L. Peng, and S. Irving. 2017. Accelerating GPU hardware transactional memory with snapshot isolation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 282–294. https://doi.org/10.1145/3079856.3080204

[12] A. ElTantawy and T. M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2016.7783714

[13] A. ElTantawy and T. M. Aamodt. 2018. Warp Scheduling for Fine-Grained Synchronization. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 375–388. https://doi.org/10.1109/HPCA.2018.00040

[14] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849

[15] W. W. L. Fung and T. M. Aamodt. 2013. Energy efficient GPU transactional memory via space-time optimizations. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 408–420.

[16] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 296–307. https://doi.org/10.1145/2155620.2155655

[17] GroupLens Research. 2006. MovieLens Data Sets. http://www.grouplens.org/node/73.

[18] Z. He and B. Hong. 2010. Dynamically tuned push-relabel algorithm for the maximum flow problem on CPU-GPU-Hybrid platforms. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–10. https://doi.org/10.1109/IPDPS.2010.5470401

[19] Blake A. Hechtman and Daniel J. Sorin. 2013. Exploring Memory Consistency for Massively-threaded Throughput-oriented Processors. *SIGARCH Comput. Archit. News* 41, 3 (June 2013), 201–212.

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[21] David S. Johnson and Catherine C. McGeoch (Eds.). 1993. *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, Boston, MA, USA.

[22] Rashid Kaleem, Sreepathi Pai, and Keshav Pingali. 2015. Stochastic Gradient Descent on GPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU-8)*. ACM, New York, NY, USA, 81–89. https://doi.org/10.1145/2716282.2716289

[23] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. 2004. RCV1: A New Benchmark Collection for Text Categorization Research. *J. Machine Learning Research* 5 (2004), 361–397.

[24] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. 2015. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 109–118. https://doi.org/10.1145/2751205.2751232

[25] C. K. Liang and M. Prvulovic. 2015. MiSAR: Minimalistic synchronization accelerator with resource overflow management. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 414–426. https://doi.org/10.1145/2749469.2750396

[26] L. Liu, M. Liu, C. J. Wang, and J. Wang. 2016. Compile-Time Automatic Synchronization Insertion and Redundant Synchronization Elimination for GPU Kernels. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. 826–834. https://doi.org/10.1109/ICPADS.2016.0112

[27] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=2342821.2342827

[28] NVIDIA. 2016. GeForce GTX 1080 Whitepaper. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[29] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[30] NVIDIA. 2018. Nvidia Profiler User Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[31] NVIDIA. 2018. Tuning CUDA Applications for Pascal. https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html

[32] Y. Oyama, K. Taura, and A. Yonezawa. [n. d.]. EXECUTING PARALLEL PROGRAMS WITH SYNCHRONIZATION BOTTLENECKS EFFICIENTLY.

[33] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/2983990.2984015

[34] Darko Petrović, Thomas Ropars, and André Schiper. 2016. Leveraging Hardware Message Passing for Efficient Thread Synchronization. *ACM Trans. Parallel Comput.* 2, 4, Article 24 (Jan. 2016), 26 pages. https://doi.org/10.1145/2858652

[35] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management*

of Data (SIGMOD '16). ACM, New York, NY, USA, 1583–1598. https://doi.org/10.1145/2882903.2882958

[36] X. Ren and M. Lis. 2017. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 625–636.

[37] X. Ren and M. Lis. 2018. High-Performance GPU Transactional Memory via Eager Conflict Detection. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 235–246. https://doi.org/10.1109/HPCA.2018.00029

[38] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 647–659. https://doi.org/10.1145/2830772.2830821

[39] Abhayendra Singh, Shaizeen Aga, and Satish Narayanasamy. 2015. Efficiently Enforcing Strong Memory Ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 699–712. https://doi.org/10.1145/2830772.2830778

[40] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 578–590.

[41] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. *SIGPLAN Not.* 44, 3 (March 2009), 253–264. https://doi.org/10.1145/1508284.1508274

[42] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero. 2010. Architectural Support for Fair Reader-Writer Locking. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 275–286.

[43] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro* 27, 5 (Sept 2007), 15–31. https://doi.org/10.1109/MM.2007.4378780

[44] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. 2016. Lock-based Synchronization for GPU Architectures. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 205–213. https://doi.org/10.1145/2903150.2903155

[45] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. 2014. Software Transactional Memory for GPU Architectures. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2544137.2544139

[46] A. Yilmazer and D. Kaeli. 2013. HQL: A Scalable Synchronization Mechanism for GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 475–486.

[47] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. 2007. Synchronization State Buffer: Supporting Efficient Fine-grain Synchronization on Many-core Architectures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 35–45. https://doi.org/10.1145/1250662.1250668