CUDA-DTM: Distributed Transactional Memory for GPU Clusters

Samuel Irving[1], Sui Chen[1], Lu Peng[1], Costas Busch[1], Maurice Herlihy[2], and Christopher J Michael[1]

[1] Louisiana State University, Baton Rouge LA 70803, USA
[2] Brown University, Providence RI 02912, USA

Abstract. We present CUDA-DTM, the first ever Distributed Transactional Memory framework written in CUDA for large scale GPU clusters. Transactional Memory has become an attractive auto-coherence scheme for GPU applications with irregular memory access patterns due to its ability to avoid serializing threads while still maintaining programmability. We extend GPU Software Transactional Memory to allow threads across many GPUs to access a coherent distributed shared memory space and propose a scheme for GPU-to-GPU communication using CUDA-Aware MPI. The performance of CUDA-DTM is evaluated using a suite of seven irregular memory access benchmarks with varying degrees of compute intensity, contention, and node-to-node communication frequency. Using a cluster of 256 devices, our experiments show that GPU clusters using CUDA-DTM can be up to 115x faster than CPU clusters.

Keywords: Distributed Transactional Memory · GPU Cluster · CUDA.

1 Introduction

Because todays CPU clock speeds are increasing slowly, if at all, some computationally intensive applications are turning to specialized hardware accelerators such as graphics processing units. Originally developed for graphics applications, GPUs have become more versatile, and are now widely used for increasingly complex scientific and machine learning applications. Though traditional GPU applications required little or no coordination among concurrent threads, GPUs are now routinely used for irregular applications that often require complex synchronization schemes to ensure the integrity of data shared by concurrent threads.

Conventional synchronization approaches typically rely on locking: a coherence strategy in which a thread must acquire an exclusive lock before accessing shared data. Though conceptually simple, locking schemes for irregular memory access applications are notoriously difficult to develop and debug on traditional systems due to well-known pitfalls: Priority Inversion occurs when a lower-priority thread holding a lock is preempted by a higher-priority thread; Convoying occurs when a thread holding a lock is delayed, causing a queue of waiting threads to form; and most importantly, a deadlock, in which overall progress halts indefinitely, can occur if multiple threads attempt to acquire a set of locks in different orders. These pitfalls are especially difficult to avoid in GPU and cluster computing applications, where the degree of parallelism is orders-of-magnitude higher than traditional applications.

Transactional Memory (TM) [8] is an increasingly popular alternative synchronization model in which programmers simply mark the beginning and end of critical sections so the system can treat those regions as "Transactions", which appear to execute atomically with respect to other transactions. At runtime, a complex conflict-detection system, invisible to the programmer, guarantees forward progress and that deadlocks cannot arise. The allure of Transactional Memory is that it commonly achieves performance comparable to that of custom lock-based solutions despite requiring only minimal effort. The programmability advantages of Transactional Memory are magnified in situations where high degrees of parallelism make lock-based solutions difficult to design and debug.

This paper investigates the performance of the first scalable Distributed Transactional Memory (DTM) [9] system for large-scale clusters of GPUs. Individual GPU threads are granted access to a coherent distributed shared memory space and can perform fine-granularity remote memory operations without halting the kernel or halting other threads within the same warp. Inter-node communication is achieved using active support from the host CPU, which sends and receives messages on behalf of the GPU. Coherence is automatically ensured using Transactional Memory, which guarantees lock-freedom, serializability, and forward progress while requiring minimal effort from programmers.

2 Related Work

There exists much prior work on the use of STM for single-device irregular memory access applications on the GPU. Cederman *et al.* [2] first proposed the use of STM on GPUs and evaluate two STM protocols. Xu *et al.* [17] proposed GPUSTM with encounter-time lock sorting to avoid deadlocks. Holey *et al.* [10] propose and evaluate multiple single-device GPU STM protocols. Shen *et al.* [15] propose a priority-rule based STM system for GPUs in which ownership of data objects can be stolen from other threads. Villegas *et al.* [16] propose APUTM, an STM design in which transactions are simultaneously executed on the GPU and host CPU. STM has been also used to maintain NVRAM persistence for GPUs [5].

There also exists much prior work in the hardware acceleration of TM on GPUs. Kilo TM [6] is a hardware-based GPU transactional memory system that supports weakly-isolated transactions in GPU kernel code; this work has been extended many times including by Chen at al. who recently described how to relax read-write conflicts with multi-version memory and Snapshot Isolation [4] and two early conflict resolution schemes [3].

There is much ongoing research in DTM for CPU clusters where it is most commonly implemented using a data-flow model, in which transactions are immobile and shared memory objects are dynamically moved between nodes [9]. DTM has been implemented in many software languages, most notably C++ [12]. There is ongoing research on how best to scale DTM to very large numbers of threads [14].

3 Design

CUDA-DTM provides an API that allows GPU programmers to treat all GPUs as a single unified compute resource and all storage resource as a single unified memory space. Individual GPU threads across all devices are assigned unique global thread IDs and allowed to access shared virtual memory space using unique global virtual memory addresses. CUDA-DTM is designed for clusters with heterogeneous nodes, each containing one or more GPU accelerators that can access the network vicariously through the host processor.

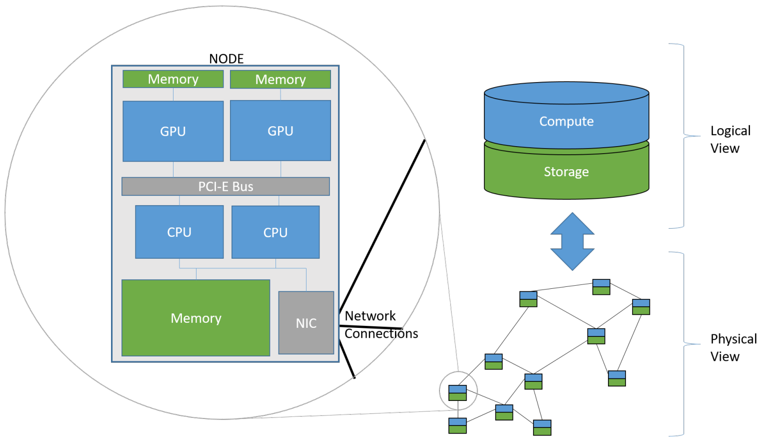


Fig. 1: Cluster-level overview of CUDA-DTM.

A lightweight STM coherence protocol allows programmers to ensure deadlock-free coherence automatically. CUDA-DTM uses custom GPU-to-GPU communication on top of CUDA-Aware MPI. A cluster level overview of CUDA-DTM is shown in Figure 1.

CUDA-DTM is designed for heterogeneous clusters in which nodes are equipped with GPU accelerators, which are the only devices executing transactions, and host CPUs, which facilitate communication between GPUs. As shown in Figure 1, the current CUDA-DTM design assumes only the CPU has direct access to the Network Interface Card (NIC) and thus must be responsible for all network communication. Node-to-Node communication is achieved using MPI. The stages for communication between devices and the network via the host in a CUDA-DTM cluster are shown in Figure 2. Only local threads are allowed to access the local

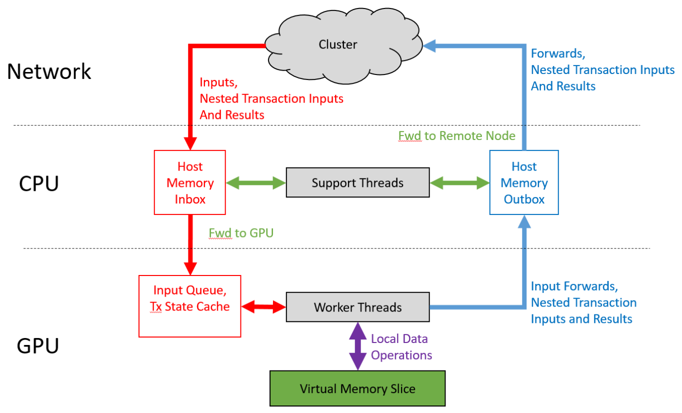

Network
Cluster
Inputs, Nested Transaction Inputs And Results
Forwards, Nested Transaction Inputs And Results
Fwd to Remote Node
CPU
Host Memory Inbox
Support Threads
Host Memory Outbox
Fwd to GPU
Input Queue, Tx State Cache
Worker Threads
Input Forwards, Nested Transaction Inputs and Results
Local Data Operations
GPU
Virtual Memory Slice

Fig. 2: Node-level overview of CUDA-DTM showing the control-flow cycle.

virtual memory slice directly. A system of message passing, shared data structures, and active support from host threads enable transactions to move to the node containing the requisite data.

Slices of the shared memory space are stored in each GPUs memory. GPU worker threads perform local data operations directly on the slice of virtual memory. CUDA-DTM uses a control-flow model, in which objects are immobile and remote procedure calls are used to move work between nodes. When a transaction accesses a virtual memory address that does not resolve locally, a remote procedure call is used to create a new sub-transaction on the remote node, termed a Remote Nested Transaction [13], by passing transaction inputs and an abbreviated execution history; this process is repeated each time transaction execution accesses data outside the local memory slice, resulting in a hierarchy structure in which top-level transactions may be comprised of many nested transactions, each detecting its own conflicts and capable of being aborted and restarted independently described in [13]. The entire hierarchy of nested transactions must be committed simultaneously.

This control-flow Remote Nested Transaction strategy only requires remote communication when transaction execution leaves the local memory slice, thereby avoiding the frequent broadcasts required by some data-flow models [9] and eliminating the need for a global clock, which can also have a significant communication overhead.

In the current design, shared memory is evenly distributed between nodes, and thus the owner of any virtual address can be found using the most-significant 8 bits of the 32-bit virtual address. Remote Nested Transaction creations and forwarded inputs, for which the critical section has not yet started, are sent to remote nodes by support threads on the host CPU. Outgoing messages are first accumulated on device before a ”ready to send” message is passed into pinned host memory. A host support thread then uses CUDA-Aware MPI to send a batch of messages to the correct destination.

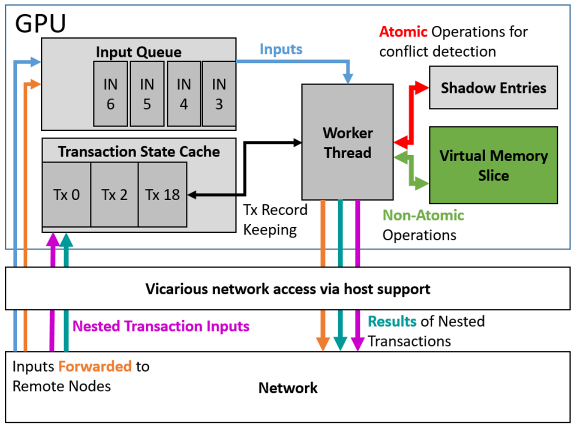

GPU
Input Queue
IN 6
IN 5
IN 4
IN 3
Inputs
Atomic Operations for conflict detection
Shadow Entries
Worker Thread
Virtual Memory Slice
Transaction State Cache
Tx 0
Tx 2
Tx 18
Tx Record Keeping
Non-Atomic Operations
Vicarious network access via host support
Nested Transaction Inputs
Results of Nested Transactions
Inputs Forwarded to Remote Nodes
Network

Fig. 3: Device-level overview of CUDA-DTM showing the two core data structures facilitating transaction control-flow.

Support threads on the host processor ensure that incoming messages accumulate in an ”inbox” in GPU global memory. GPU worker threads pull work assignments out of the inbox and perform work depending on the contents of the message; types of messages in the system and the two data structures that function as an inbox are shown in Figure 3. The inbox consists of 1) an Input Queue, which accumulates the parameters for un-started transactions, and 2) the Transaction State Cache which is used to store the current state and access history of transactions that have entered the critical section on the current node. Host support threads are capable of accessing these structures during kernel execution using asynchronous CUDA memcpys.

Communication between GPUs is facilitated by two structures stored in global memory: the Input Queue and the Transaction State Cache, as shown in Figure 3. The input queue receives blocks of inputs, each containing the parameters for an un-started transaction; the size and usage of each input is application specific. The Transaction State Cache is used to store undo-logs for Transactions that are waiting for the result of a Remote Nested Transactions. Each working thread on the GPU has a Transaction State Cache Set that it is responsible for which is regularly polled when no other work is available.

During the execution of a transaction, a transaction state is created and maintained in local memory; the active transaction state is modified when performing atomic operations to the shadow entries stored locally using the conflict detection rules described in Section 3.1. When execution of a transaction accesses a virtual memory address outside of the current node and must create a Remote Nested Transaction, an entry is created in the Transaction State Cache; the entry contains the unique transaction ID, created using the unique thread ID shifted and then added to a private counter, the largest address accessed so far, the undo-log, and state variable indicating the transaction has not yet been aborted nor committed. Remote Nested Transactions are created directly in the remote Transaction State Cache of the node containing the desired data; Transaction State entries are stored such that they can be copied directly from memory using CUDA-Aware MPI. Similarly, the results of a transaction can be sent directly into the Transaction State cache – overwriting the state variable so working threads can see that a transaction has been committed or aborted. Serialization and deserialization are handled entirely by CUDA when communicating between device and host and entirely by MPI when communicating between nodes.

3.1 Transactional Memory Model

CUDA-DTM detects and resolves conflicts using a modified version of the Pessimistic Software Transactional Memory (PSTM) protocol described in [10] built on top of the virtual memory system described above. The use of a distributed memory space is invisible to the transactional memory protocol as a new remote nested transaction is created each time execution moves between nodes.

Ownership is tracked via 32-bit Shadow Entries that store the unique virtual transaction id number for the transaction that is accessing the corresponding object; shadow entries are all initialized to be 0. This design uses a single-copy model in which there is only one write-able copy of each object in the system; while this forces the serialization of accesses to individual objects, it also minimizes the storage and compute overheads of the system, allowing the working data set size to be very large.

Threads in the same warp are allowed to execute simultaneous transactions using a private state variable, which masks off threads that have been aborted or are waiting for work. Live-locks are prevented using exponential back-off, in which transactions that are aborted multiple times are forced to wait an exponentially increasing length of time before restarting.

PSTM was chosen for our design due to its simplicity, low overheads, and its eager conflict detection which aborts transactions early and can help reduce the number of remote messages.

When a transaction begins execution or is restarted: its local state is set to ACTIVE and its local undo log is cleared, as shown in Algorithm 1. Each transaction maintains a private undo log in local memory which can be used to reverse changes to local shadow entries and shared memory in the event of an abortion. A single transaction may create several Remote Nested Transactions, each with its own private undo log on its respective node.

Algorithm 1: TX_begin(i)

```
$T_i.state$ = ACTIVE;
$T_i.undoLog \leftarrow \emptyset$;
```

Algorithm 2: TX_validate(i)

```
if $T_i.state = ACTIVE$ then
    TX_commit(i);
else
    TX_abort(i);
end
```

Algorithm 3: TX_commit(i)

```
foreach $u \in T_i.undoLog$ do
    $shadow[u.addr]$ ←0;
    end
    if $T_i.parent$ != null then
        send abort result to
          parent;
    end
```

Algorithm 4: TX_abort(i)

```
foreach $u \in T_i.undoLog$ do
    $data[u.addr]$ ←u.value;
    shadow[u.addr]←0;
    end
    if $T_i.parent$ != null then
        send commit result to
          parent;
    end
```

Algorithm 5: TX_access(i,addr)

```
if $T_i.state$ != ACTIVE then
    return;
end
if addr is not local then
    Create remote nested Tx;
    $T_i.state = WAITING$;
    return;
end
if $addr \notin T_i.undoLog$ then
    if $addr > T_i.maxAddr$
     then
        while(!tryLock(addr,i));
    else
        if !tryLock(addr,i) then
            $T_i.state =$
              $ABORTED$;
            return;
        end
    end
    add (addr,objects[addr]) to
     $T_i.undoLog$;
end
```

Algorithm 6: tryLock(i,addr)

```
if $shadow[addr] \neq 0$ then
    return false;
end
return CAS(shadow[addr],i,0));
```

At validation, any transaction whose state is still ACTIVE is ready to be committed, as shown in Algorithm 2. A thread's state is only set to ABORTED after failing to acquire exclusive control over a specific shared memory address. Setting the state to ABORTED effectively masks off threads when other threads in the same warp are still ACTIVE.

If still ACTIVE at validation time, all changes performed by the transaction must be made permanent by simply releasing all locks acquired during execution, as shown in Algorithm 3. If this transaction is a Remote Nested Transaction that was created by a parent transaction on another node, then a result message must be sent to the parent node.

In the event of an abort, a thread must iterate through the undo log, restore the original object values, and reset ownership of the corresponding shadow entries as shown in Algorithm 4. If this transaction is a Remote Nested Transaction, then a result message must be sent to the parent node indicating that all transactions must be restarted; otherwise, the transaction will resume execution when the thread warp re-executes TX_begin.

For simplicity, we combine TX_read and TX_write into TX_access, as shown in Algorithm 5, because PSTM does not distinguish between read and write operations when detecting conflicts. PSTM pessimistically assumes that any address touched by a transaction will eventually be modified, and thus a transaction should immediately be aborted if it fails to acquire exclusive control over a specific shared memory address.

Although transactions can perform speculative writes to shared memory, other threads cannot read these values until the transaction commits and the corresponding shadow entry is released.

When TX_access is called using a virtual address that is mapped to a different node, execution of the current transaction must be suspended and a Remote Nested Transaction created. Execution of the parent transaction is suspended by first storing the undo-log into the local Transaction State Cache and, if the transaction originated on the current node, assigning it a unique ID. The working thread indicates the target node when creating a Remote Nested Transaction Start message, along with variables required to begin execution on the remote node, and includes the largest address accessed so far. This message is inserted into the appropriate remote GPU Transaction State Cache where a new transaction state is created including a new local-only undo-log and a reference to the originating node that will ultimately receive a message indicating the result of the transaction. The process of creating a Remote Nested Transaction, suspending, and resuming transactions is handled entirely by the CUDA-DTM system and is invisible to the programmer.

To gain ownership of an object, a thread will perform an Atomic Compare-and-Swap operation (CAS) on the objects corresponding shadow entry, as shown in Algorithm 6. This CAS operation attempts to atomically exchange the current shadow entry value with the threads unique, non-zero id. This exchange is only performed if the expected value of 0 is found; otherwise, the function returns the value discovered before the exchange. If the function returns a non-zero value, then the current transaction has failed to gain ownership and may abort. If the exchange is successful, the transaction is allowed to proceed.

Our modified PSTM allows transactions to use blocking atomic operations when accessing addresses in increasing order; this is tracked by storing the max-address-locked-so-far (termed “maxAddr” in Algorithm 5). This strategy reduces the total number of abortions, as a transaction is only aborted when trying and failing to acquire a lock out of order. Transactions can proceed as normal if an out of order lock is successfully acquired on the first try.After successfully accessing a shared memory object, its address and current value are inserted into the undo log so that speculative changes can be reversed in the event of an abortion (referred to as (addr, objects[addr]) in Algorithm 5).

3.2 Communication

GPU worker threads provide virtual memory addresses to the CUDA-DTM API, which performs the necessary communication operations under-the-hood. Operations using virtual addresses that are mapped to local physical addresses resolve quickly because the object and shadow entry are stored in local global memory. However, when a virtual address is mapped to a remote physical address, the API automatically creates a Remote Nested Transaction that continues execution on the remote device that contains the requisite data.

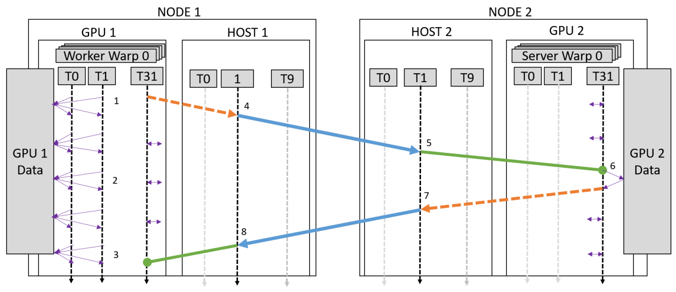


Fig. 4: Timing of the communication protocol stages showing the execution of a Remote Nested Transaction.

The CUDA-DTM communication protocol uses three asynchronous messages passes, as shown in Figure 4: 1) the originating thread writes a message to an outbox in global memory (orange dashed arrow) and then sets a ready Boolean in host memory to true; 2) a support thread on the host detects that the ready Boolean is true for a outbox and sends the message to the correct node using an asynchronous MPI write to remote host memory (thick blue arrow); 3) support threads in the remote nodes host receive the incoming message and place it in the correct threads inbox using an asynchronous cudaMemCpy (green line with circle on the end).

Depending on how aggressively messages are batched, all threads may have a designated inbox in global device memory and a designated outbox in pinned host memory.

After creating a Remote Nested Transaction, worker threads are allowed to begin execution of a new transaction; worker threads cycle between responsibilities when blocked waiting for remote communication by polling the transaction state cache and input queue (purple double-sided arrow).

Figure 4 shows the timing of GPU-to-GPU communication for transactions that have already begun the critical section of a transaction that increments multiple addresses. (1) Warp 0 is initially un-diverged and all threads begin virtual memory increments using different virtual memory addresses. Of the threads shown, only thread 31s virtual memory address is mapped to a physical address on a remote node. Threads 0 and 1 are forced to wait while Thread 31 enters its transaction state into the transaction state cache, builds a Remote Nested Transaction creation message and notifies the host that a message is waiting to send. Finally, the rest of the warp quickly make copies of the desired objects from global memory. (2) In this example, the transaction state cache and input queue have no available work for Thread 31 to begin, so Threads 0 and 1 continue to perform virtual memory operations while Thread 31 is masked off. When other threads in the warp use the CUDA-DTM API, thread 31 polls the input queue and checks the state of its suspended transaction waiting for work. (3) After five memory operations, the warp finally re-converges when thread 31 receives the Nested Transaction Result.

(4) In this example, ten host threads are responsible for supporting the local GPU worker threads. Responsibility for checking outboxes for readiness is evenly divided among host threads, and thus support thread 1 sees outbox 31 is ready, uses the messages address to calculate its destination, and sends the message to node 2 using an asynchronous MPI write operation. (5) On node 2, host support thread 1 checks thread 31s inbox, discovers a new message, and copies the message into device memory using an asynchronous CUDA copy. (6) Thread 31 on Node 2, having been polling its inbox for incoming work, receives the result of the Nested Transaction from Node 1-Thread 31, begins execution of the Nested Transaction on the new node using a fresh-undo log. Here, the desired virtual memory address resolves locally and the increment is completed successfully. Having reached the end of the Nested Transaction, Node 2's thread 31 commits the transaction by releasing ownership of local shadow entries and destroying the corresponding entry in the transaction state cache. Thread 31 creates a new Nested Transaction Result message indicating the transaction is complete and sends it to the originating Node 0. (7) Support thread 1 on host 2 detects an outgoing message is ready and sends the message back to host 1 where (8) support thread 1 on host 1 copies the final transaction result into the inbox of the originating worker thread using CUDA asynchronous copy to device.

CUDA-Aware MPI is used in cases where outgoing messages can be batched together in global memory, all bound for the same destination. In these cases, only the owner of the final message added to the batch is forced to notify the host that the batch is ready to send. The protocol is achieved using single-writer, single-reader arrays when possible, avoiding the need for atomic operations that increase overheads.

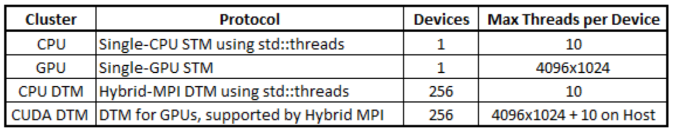
4 Experimental Analysis

For this experiment, we use a set of seven irregular memory access benchmarks commonly used for studying TM; the benchmarks differ in length, composition, contention, and shared data size. A 128-node cluster featuring two CPUs and two GPUs per node is used for this experiment using a 56GB/s Infiniband over-subscribed mesh; each CPU is a 2.8GHz E5-2689v2 Xeon processor with 64GB RAM; each GPU is a NVIDIA Tesla K20x connected via an Intel 82801 PCIe bridge. CUDA-DTM is compiled using CUDA v9.2.148 and MVAPICH2 version 2.2.

Coherence protocols are detailed in Table 1. Transactions are only executed by GPU threads in the GPU and CUDA-DTM configurations.

For this work we use seven benchmarks commonly used to profile TM performance: Histogram (HIST) [1], in which the results of a random number generator are stored in a shared array; two variants of the Hash Table benchmark [7]: one in which each transaction inserts a single element (HASH-S), and one where each transaction inserts four elements simultaneously (HASH-M), as described in [10]; Linked-List (LL) [7], in which elements are inserted into a sorted List; KMeans [11]; and two graph algorithms: Single-Source Shortest Path (SSSP) [1] and Graph-Cut (GCut), which finds the minimum cut of a graph using Karger's algorithm [4].

Table 1: Coherence protocols



Cluster	Protocol	Devices	Max Threads per Device
CPU	Single-CPU STM using std::threads	1	10
GPU	Single-GPU STM	1	4096x1024
CPU DTM	Hybrid-MPI DTM using std::threads	256	10
CUDA DTM	DTM for GPUs, supported by Hybrid MPI	256	4096x1024 + 10 on Host

Using 128 nodes, CUDA-DTM achieves a harmonic mean speedup of 1,748x over the single-node, multi-threaded CPU baseline across the 10 benchmarks used in this study, as shown in Figure 5. Similarly, CUDA-DTM achieves a harmonic mean speedup of 6.9x over a CPU cluster of the same size due to the performance advantages of the GPU architecture. CPU DTM achieves slightly less than a 256x speedup over a single CPU due to the high parallelizability of all seven benchmarks and long run times hiding network latencies. The near-ideal speedup of CPU DTM suggests that the 56 Gb/s bandwidth of the network is never saturated with messages.

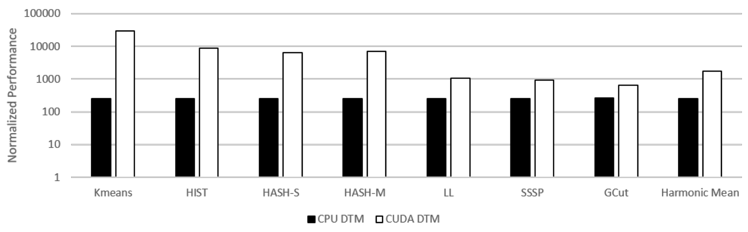


Fig. 5: The performances of CUDA-DTM and CPU DTM on a 128 node cluster normalized by single-node CPU performance.

The speedups achieved by CUDA-DTM are best explained by the execution time breakdown shown in Figure 6. Using Figures 5 and 6, we see that CUDA-

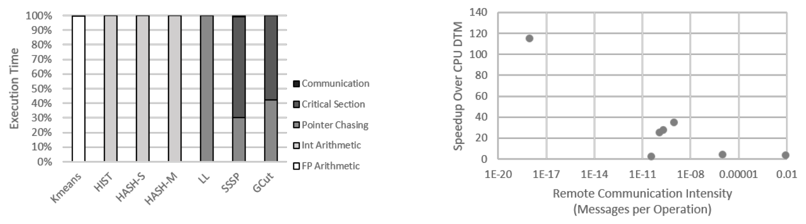
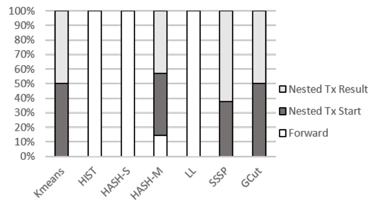


Fig. 6: (Left) CUDA-DTM execution time breakdown and (Right) CUDA-DTM Speedup over CPU DTM vs. remote communication intensity.

DTM achieves a speedup of 25 to 115x over the CPU for compute intensive benchmarks, in which execution time is dominated by arithmetic operations, consistent with the ∼70x higher theoretical peak throughput of the GPU. Similarly, we see CUDA-DTM achieves a speedup of 2.5 to 4.2x for memory intensive benchmarks, in which execution time is mostly spent chasing pointers through shared memory, similar to the ∼4.2x higher theoretical bandwidth of the GPU (250 GB/s vs 59.7 GB/s). Finally, we see the smallest speedup for benchmarks with high contention, as the advantages of the massive number of GPU threads is limited by blocking atomic operations during the critical section. Remote communication is only a very small percentage of the execution time despite varying degrees of remote-communication intensity.

CUDA-DTM's sensitivity to the remote-communication intensity of the workload is visualized in Figure 6. Here we see benchmarks with the most infrequent remote communications generally show the largest speedup over the CPU, though the magnitude of the speedup is heavily impacted by the type of operations used between remote communication. Benchmarks with the highest communication intensity are also memory-intensive, limiting the potential speedup to the ∼4.2x higher memory bandwidth of the GPU. The best performing benchmark, KMeans, is very FLOP intensive, benefiting from both the high volume of operations between remote messages and the ∼70x higher computational throughput of the GPU. CUDA-DTM's speedup will converge on 1x as the remote intensity increases, because the GPU has no communication advantages over the CPU.

Figure 7 shows the average number of messages generated per committed transaction for each benchmark. GCut generates the fewest messages per transaction while showing the smallest speedup over the CPU while HIST, HASH-S, and HASH-M all show largest speedups despite delivering at least one message per transaction. LL generates over 100 messages per transaction while searching the shared List for the proper data insertion point; we use this graph to suggest that the bottleneck of each benchmark is not the inter-node bandwidth, as the GPU has no inter-node bandwidth advantages over the CPU. KMeans generates very few messages, as centroids are only globally averaged after long spans of intra-node averaging. Similarly, GCut runs isolated instances of Karger's algo-



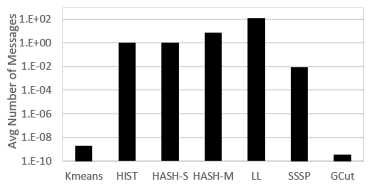


Fig. 7: (Left)Avg number of remote messages generated per transaction showing varying degrees of network intensity. (Right) Breakdown of remote message types.

rithm on each node, only generating messages when a new lowest-min-cut-so-far is discovered.

The types of remote messages generated by each transaction are profiled in Figure 7. HIST, HASH-S, and LL almost never have critical sections that span multiple nodes; nearly all messages are Forwards. HASH-M is similar to its -S counterpart, except the critical section almost always spans multiple nodes; in HASH-M threads will likely perform many non-atomic operations after locking shadow entries but since the critical section has started the transaction must always created Remote Nested Transactions. The remaining benchmarks generate Remote Nested Transaction Start- and Result- messages in nearly equal number, due to low abortion rates and only using the network during the critical section.

Compute intensive workloads have the potential for the largest speedup on GPU clusters due to the $\sim$70x higher theoretical computational throughput. Figure 8 shows that KMeans, HIST, and both HASH benchmarks have a much higher compute intensity than the remaining benchmarks.

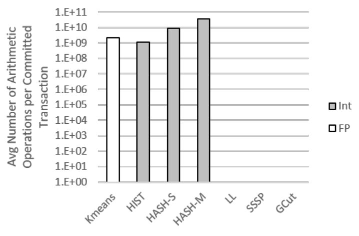


Fig. 8: Average number of arithmetic operations per committed transaction.

The KMeans benchmark exhibits nearly ideal behavior for the GPU and thus show the best performance improvements over the CPU in our experiments, as shown in Figure 8. In these benchmarks, each transaction performs a long series of distance calculations before acquiring a single lock for a brief critical section. The computation intensity, and thus the magnitude of the GPU advantage, of the benchmark is proportional to the number of dimensions for each data point.

Remote communication is minimal, as each node effectively runs in isolation before using a binary-tree style reduction and time between these synchronizations is long. KMeans achieves more than the expected ∼70x speedup, and closer to the ideal ∼140x higher FLOPS reported in the K20x specifications due to the very infrequent usage of remote communication and shared memory and comparatively higher FLOP density.

The Histogram, HASH-S, and HASH-M benchmarks show large improvements over the CPU in our experiments, though not as large as KMeans, as shown in Figure 8. These benchmarks perform a long series of shift and XOR operations on integers to produce random keys to be inserted into a shared data structure using an Xorshift random number generating algorithm. Performance is again compute-bound, this time dominated by shift and XOR operations, and thus the GPU has a large advantage. The large volume of integer operations is again sufficient to hide the time spent searching for the linked-list insertion points in both HASH benchmarks and the remote memory access resulting from each transaction. Similarly, the increased contention of the HASH-M benchmark has little impact on performance due to the compute intensity of the random key calculations. Histogram outperforms HASH-S and HASH-M because it requires no memory operations outside of the critical section; HASH-S and HASH-M require long searches through linked lists, though HASH-M benefits from requiring 4x as many integer operations as HASH-S.

We profile the number of non-atomic virtual memory operations per committed transaction and show the results in Figure 9. Memory intensive applications can benefit from the ∼4.2x higher bandwidth of GPU global memory and the increased parallelism of cluster computing. We observe the LL benchmark has the largest volume of memory accesses and recall from Figure 6 that execution time is overwhelmingly spent performing memory accesses.

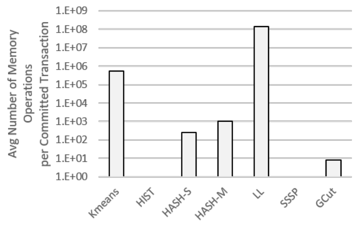


Fig. 9: Average number of local memory accesses per transaction.

Figure 9 shows benchmarks that still benefit from the GPU's higher global memory bandwidth, despite the remote communication overheads. CUDA-DTM shows a 4.2x speedup over the CPU DTM baseline, though performance is limited by irregular memory access patterns, the overheads of transaction record keeping, and warp divergence. Execution time is dominated by long searches through memory, which hides the large average number of messages sent per transaction. The expected speedup for memory-intensive applications is calculated using the

CPUs reported 59.7 GB/s max memory bandwidth and the GPUs reported 250 GB/s global memory bandwidth, as the much faster GPU shared memory cannot be used for atomic operations nor is it sufficiently large to store the virtual memory slice.

We measure the contention of each benchmark using the average number of shadow entries modified per transaction and the average time required to gain data ownership. KMeans and HIST require a single lock, as their critical sections make changes to one shared object.

Benchmarks that require changes to dynamic data structures require two locks per insertion: one for allocating a new object and one for updating the pointer on an existing object; as such, HASH-S and LL require exactly two locks for each transaction and HASH-M, which inserts four objects simultaneously, requires exactly eight. Contention in these benchmarks is low because changes are diluted in a very large number of shared objects.

GCut requires exactly two locks to merge two lists together by updating a pointer, though contention increases during execution as vertices are merged and the number of shared objects decreases; as result, the amount of time required to acquire each lock increases as shown in Figure 10. SSSP is the only benchmarks in this study which require a variable number of locks, though the average in each is low. The average and maximum transaction length, 32 in each case, is determined by the topology of the graph. The minimum, only one in each benchmark, is used when the propagation rules do not require visiting any neighbors.

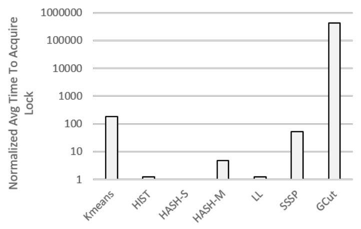


Fig. 10: Normalized wait time per lock.

Figure 10 shows the average amount of time required to successfully complete a CAS operation on a single shadow entry. Times are normalized by that of the HASH-S benchmark, in which contention is the lowest due to the large number of shared objects and short amount of time spent in the critical section. GCut has the longest wait time by far, due to the decreasing shared data size and thus the increasing contention. Despite KMeans high performance, the time spent acquiring locks is second highest due to very small shared memory size and the large number of threads; KMeans performance is still dominated by FLOPs and the impact of the high contention is hidden. However, SSSP and GCut are unable to hide lock-acquisition latency using global memory accesses or arithmetic operations, and their performance suffers as shown in Figure 10, in which they achieve only a fraction of the theoretical ~4.2x speedup from higher bandwidth.

SSSP and Min-Cut are both graph based benchmarks where a subset of the graph must be locked and modified by each transaction; performance is limited by longer transactions (2 to 32 shadow entries each) resulting in high contention (35x higher than the average of all benchmarks), which limits the advantages of the GPUs high parallelism.

5 Conclusion

We propose CUDA-DTM, the first implementation of a coherent distributed shared memory system for GPU clusters using Distributed Transactional Memory. This paper demonstrates that a GPU cluster can outperform a CPU cluster in non-network intensive workloads despite irregular memory accesses and the overheads of accessing virtual memory. We also demonstrate that the strengths of the GPU, namely the high arithmetic operation throughput and higher memory bandwidth, offer large performance advantages over the CPU despite the large number of moving pieces required to support irregular distributed memory access. Our design allows programmers to use coherent remote memory operations without worrying about deadlocks from thread-divergence or lock competition.

References

1. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on gpus. In: 2012 IEEE International Symposium on Workload Characterization (IISWC). pp. 141–151. IEEE (2012)
2. Cederman, D., Tsigas, P., Chaudhry, M.T.: Towards a software transactional memory for graphics processors. In: EGPGV. pp. 121–129 (2010)
3. Chen, S., Peng, L.: Efficient gpu hardware transactional memory through early conflict resolution. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 274–284. IEEE (2016)
4. Chen, S., Peng, L., Irving, S.: Accelerating gpu hardware transactional memory with snapshot isolation. In: Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on. pp. 282–294. IEEE (2017)
5. Chen, S., Zhang, F., Liu, L., Peng, L.: Efficient gpu nvram persistent with helper warps. In: ACM/IEEE International Conference on Design Automation (DAC). ACM/IEEE (2019)
6. Fung, W.W., Singh, I., Brownsword, A., Aamodt, T.M.: Hardware transactional memory for gpu architectures. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 296–307. ACM (2011)
7. Gramoli, V.: More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In: ACM SIGPLAN Notices. vol. 50, pp. 1–10. ACM (2015)
8. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures, vol. 21. ACM (1993)
9. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing **20**(3), 195–208 (2007)

10. Holey, A., Zhai, A.: Lightweight software transactions on gpus. In: Parallel Processing (ICPP), 2014 43rd International Conference on. pp. 461–470. IEEE (2014)
11. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: 2008 IEEE International Symposium on Workload Characterization. pp. 35–46. IEEE (2008)
12. Mishra, S., Turcu, A., Palmieri, R., Ravindran, B.: Hyflowcpp: A distributed transactional memory framework for c++. In: Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on. pp. 219–226. IEEE (2013)
13. Moss, J.E.B.: Nested transactions: An approach to reliable distributed computing. Tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE (1981)
14. Sharma, G., Busch, C.: Distributed transactional memory for general networks. Distributed computing **27**(5), 329–362 (2014)
15. Shen, Q., Sharp, C., Blewitt, W., Ushaw, G., Morgan, G.: Pr-stm: priority rule based software transactions for the gpu. In: European Conference on Parallel Processing. pp. 361–372. Springer (2015)
16. Villegas, A., Navarro, A., Asenjo, R., Plata, O.: Toward a software transactional memory for heterogeneous cpu–gpu processors. The Journal of Supercomputing pp. 1–16 (2017)
17. Xu, Y., Wang, R., Goswami, N., Li, T., Gao, L., Qian, D.: Software transactional memory for gpu architectures. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. p. 1. ACM (2014)