BifurKTM: Approximately Consistent Distributed Transactional Memory for GPUs

Samuel Irving
Louisiana State University, Baton Rouge, LA, USA

Lu Peng
Louisiana State University, Baton Rouge, LA, USA

Costas Busch
Augusta University, GA, USA

Jih-Kwon Peir
University of Florida, Gainesville, FL, USA

Abstract

We present BifurKTM, the first read-optimized Distributed Transactional Memory system for GPU clusters. The BifurKTM design includes: GPU KoSTM, a new software transactional memory conflict detection scheme that exploits relaxed consistency to increase throughput; and KoDTM, a Distributed Transactional Memory model that combines the Data- and Control- flow models to greatly reduce communication overheads.

Despite the allure of huge speedups, GPUs are limited in use due to their programmability and extreme sensitivity to workload characteristics. These become daunting concerns when considering a distributed GPU cluster, wherein a programmer must design algorithms to hide communication latency by exploiting data regularity, high compute intensity, etc. The BifurKTM design allows GPU programmers to exploit a new workload characteristic: the percentage of the workload that is Read-Only (e.g. reads but does not modify shared memory), even when this percentage is not known in advance. Programmers designate transactions that are suitable for Approximate Consistency, in which transactions "appear" to execute at the most convenient time for preventing conflicts. By leveraging Approximate Consistency for Read-Only transactions, the BifurKTM runtime system offers improved performance, application flexibility, and programmability without introducing any errors into shared memory.

Our experiments show that Approximate Consistency can improve BkTM performance by up to 34x in applications with moderate network communication utilization and a read-intensive workload. Using Approximate Consistency, BkTM can reduce GPU-to-GPU network communication by 99%, reduce the number of aborts by up to 100%, and achieve an average speedup of 18x over a similarly sized CPU cluster while requiring minimal effort from the programmer.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems

Keywords and phrases GPU, Distributed Transactional Memory, Approximate Consistency

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.2

1 Introduction

GPUs have become the device of choice for high performance and scientific computing due to their high computational throughput and high memory bandwidth compared to the CPU. The popularity of the GPU has ushered in the era of "General Purpose GPU Computin" in which Graphics Processors are increasingly used for applications that they were not originally designed for, including applications with recursion, irregular memory access, atomic operations, real-time communication between the GPU and its host processor [15], and even real-time communication between the GPU and remote devices using MPI [12].

12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021).
Editors: João Bispo, Stefano Cherubin, and José Flich; Article No. 2; pp. 2:1–2:15

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Despite its strengths and increasing flexibility, the GPU suffers from poor programmability, as GPU programmers must meticulously balance performance, correctness, and code readability [7]. Programmers must be wary to: avoid critical warp divergence caused by misuses of the GPU's Single Instruction Multiple Data architecture; be wary of the GPUs weak memory model and requirement that programmers manually control several distinct memory spaces; and be careful to ensure data integrity for degrees of parallelism several orders-of-magnitude greater than traditional CPUs. These concerns are magnitude exponentially when scaling GPU applications to the cluster scale.

Transactional Memory (TM) [8] has emerged as a concurrency control strategy that can ensure GPU program correctness while greatly improving programmability. When using TM, programmers simply mark the beginning and end of critical sections that are then converted, at compile time, into architecture-safe implementations that guarantee correctness and dead-lock freedom regardless of application behavior. TM allows programmers to quickly develop performant versions of complex applications without complete understandings of the underlying hardware, workload behavior, or synchronization requirements. These advantages have motivated research into Hardware TM [7, 4, 5], Software TM (STM) [3], and Distributed TM (DTM) [11] for GPUs. Transactional Memory implementations are often modular and highly customizable, allowing programmers to try a variety of performance optimization techniques and hardware configurations without rewriting the program.

We present BifurKTM, the first distributed Transactional Memory system for GPU clusters that uses *Approximate Consistency* [1] to improve the performance. BifurKTM allows programmers to benefit from the GPU's high computational throughput and high memory bandwidth despite the presence of GPU-initiated remote communication and irregular memory accesses in their application. Furthermore, BifurKTM allows GPU programmers to leverage a new application property to improve performance: the percentage of the workload that reads but does not modify shared memory.

The contributions of this paper are as follows:

1. We propose *GPU KoSTM*: a read-optimized conflict detection protocol for GPUs to reduce conflicts using Approximate Consistency.
2. We propose *KoDTM*: a distributed transactional memory model using K-Opacity [1] to reduce node-to-node communication frequency using Approximate Consistency.
3. Based on GPU KoSTM and KoDTM, we design *BifurKTM*: an implementation of GPU DTM that achieves better speedups over CPU clusters using Fine-Grained locking and existing GPU DTM designs.

2 Background and Related Work

Software Transactional Memory (STM) for the GPU must have two key components 1) a strategy for detecting conflicts and 2) a method for transforming the original transaction, written by the programmer, into an implementation that ensures deadlock freedom, correctness, and forward progress despite the addition of loops unanticipated by the programmer (retry after abortion, repeat non-blocking atomic operation, etc.).

Lightweight GPU STM [10] uses a fall-through transformation, in which each transaction is placed inside a while-loop that is repeated until all threads in a warp are successful; threads are "masked off" when a conflict is detected. To allow for long-running remote operations and the pausing and resuming of transactions, a state-machine transformation is used in CUDA DTM; this transformation introduces many additional branching overheads, but allows threads within the same warp to execute the stages of the transaction completely out of sync [11]. GPU KoSTM relies on a combination of the fall-through and state-machine transformations.

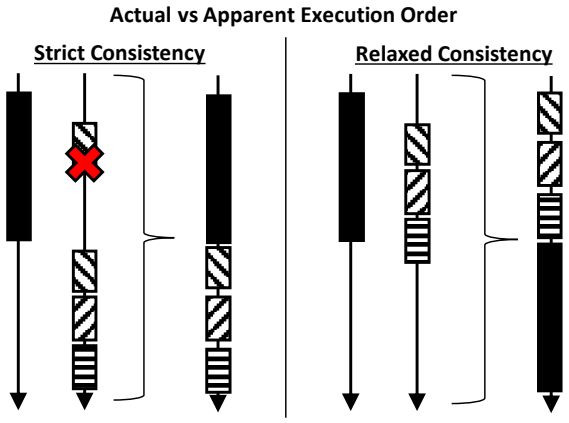


Figure 1 Conflict between Update (solid) and Read-Only transactions (striped) is prevented while maintaining serializability. When using Relaxed Consistency, the Read-Only (striped) transactions "appear" to have executed earlier than they actually did (compared to the solid Update transaction).

Distributed Transactional Memory (DTM) [9] has been implemented for traditional CPU clusters using two models: (1) the Data-Flow model [14], in which objects move between nodes but transaction execution is stationary; and (2) the Control-Flow model [16], in which shared memory is statically mapped but transactions execute across multiple nodes using Remote Procedure Calls (RPCs). DTM for GPU clusters has only been implemented using the Control-Flow model, which avoids the overheads of maintaining a coherent data cache on the GPU which can greatly impair performance [11]. DTM requires support threads that are not explicitly created by the programmer for facilitating remote communication and executing RPCs. Alongside GPU DTM research, there is ongoing research in using Transactional Memory to facilitate cooperation between the CPU and GPU [2], [17], which faces similar challenges due to the requisite communication between different memory spaces, large number of working threads, and high penalties for synchronizing both devices.

Approximate Consistency has been proposed as a method to reduce conflicts by allowing transactions to "appear" as if they executed in a "more convenient" order that would not have resulted in a conflict [1]. The term "approximate" is used in the sense that transactions appear to execute "approximately" when issued by the application; the runtime system is not producing estimates of current values. Fig. 1 shows how relaxed consistency allows three Read-Only transactions to appear to have been executed before a conflict with Update transaction, allowing all transactions to finish earlier while maintaining serializability.

3 Design

BifurKTM (BkTM) is an implementation of Distributed Transactional Memory (DTM) for GPU clusters designed for applications in which workloads typically exhibit characteristics well-suited for the GPU architecture, but irregular memory accesses and atomic operations present major challenges to the programmer. The goal of BkTM is to relax the high barrier of entry required for performant GPU programming by allowing developers to ignore the underlying GPU hardware and network topology while still ensuring that a program runs correctly and to completion. Though some knowledge of the system is required to achieve large speedups over corresponding CPU clusters, The goal of BkTM is to allow developers to create distributed GPU applications in minutes that have historically taken days or months.

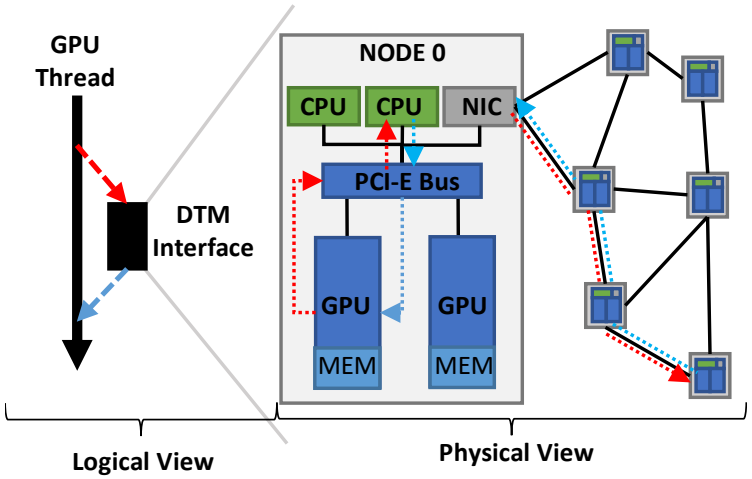


Figure 2 BkTM System Overview showing how the network topology, off-device communication, and distributed memory spaces are "hidden" behind the DTM interface (from the perspective of GPU threads).

When using BkTM, the programmer can achieve correctness and performance while being blind to the underlying GPU architecture and network topology, as shown in Fig. 2. To use BkTM: the programmer has two key responsibilities; 1) marking the beginning and end of critical sections and 2) marking which data is shared between nodes. At compile-time, BkTM will transform transactions into multiple GPU-safe implementations that can be repeated until successful, despite atomic operations and remote communications. Memory accesses within the loop are converted into TM_READs and TM_WRITEs, which direct memory accesses into the STM manager. Finally, a programmer can choose to mark a critical section as "Read-Only", allowing the BkTM runtime system an opportunity to improve performance.

The BkTM runtime system guarantees shared data integrity and strict correctness at all times. Any transaction that might modify shared memory ("Update Transaction") is not allowed to use the Approximate Consistency optimizations; the use of Approximate Consistency is limited exclusively to transactions declare in advance that they will not make changes to shared memory ("Read-Only Transactions"). Correctness and deadlock freedom is guaranteed at the device level by the KoSTM protocol, and at the cluster-level by KoDTM detailed in section 3.2.

BkTM is built on top of a custom Partitioned Global Address Space (PGAS) implementation, though the principles here described can be adapted to any Distributed Shared Memory implementation. A PGAS allows GPU threads to read and write the locally mapped partition with minimal overheads and no additional atomic operations. In BkTM: a single-copy model is used for modifiable data, which is evenly divided between devices. Attempts to modify data in a partition owned by a remote device will incur a series of network communications before the transaction can complete; network communications along the critical path for a transaction can be devastating for overall GPU performance as off-device bandwidth and latency immediately become bottlenecks for transactional throughput. BkTM prevents some remote accesses by introduces a virtual memory hierarchy between the device and the network, as shown in Fig. 3. The Read-Only cache contains copies of remote partitions, and eliminates the need for long-latency remote partition accesses for read only transactions. In

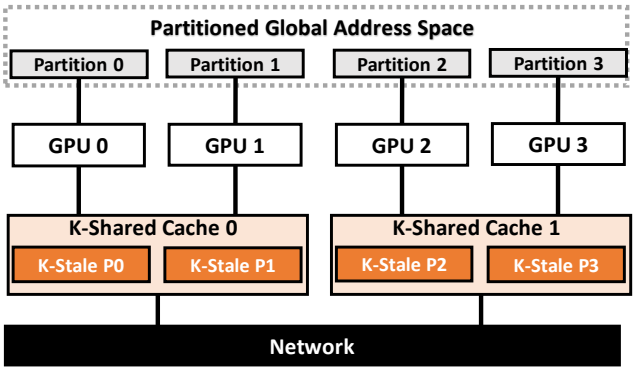


Figure 3 BkTM Virtual Memory Hierarchy showing that each GPU has 1) fast access to a local partition 2) fast read-only access to K-Stale copies of partitions for GPUs within the same sub-cluster, and 3) comparatively slow access to all partitions via the network.

this work we term Read-Only K-Opaque copies of remote partitions as "K-Shared" copies that can be used by "K-Reading" nodes. Modifiable values are statically mapped to a home node, which must track and update K-Shared copies on remote nodes.

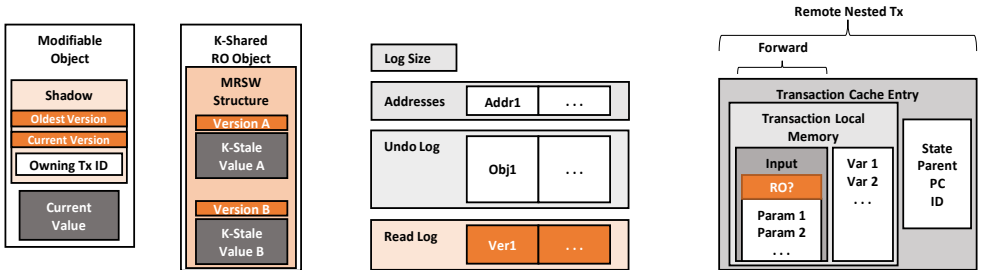
The GPU is a particularly unforgiving device for developers, where the penalties for misunderstanding application behavior can be devastating to performance and expose the system to deadlocks. While BkTM makes the usual TM guarantees for deadlock-freedom, forward progress, etc. the system cannot capitalize on any the GPU advantages (with respect to CPUs) unless they are fundamental to the workload: this can be any of 1) compute intensity, 2) high data access regularity, 3) high parallelizability, and, as introduced in this work, 4) high read intensity. The absence of all of these critical advantages ensures that GPU-to-GPU bandwidth will be the bottleneck of the system, and the high FLOPS and local memory bandwidth of the GPU become irrelevant to performance.

In BkTM, Read-Only transactions are allowed to access "stale" copies of shared data objects, even if the current version is being modified by another thread or if the only modifiable version is owned by another device, provided the stale version is within "K" versions of the current (K-Opaque). Both GPU KoSTM and KoDTM work together to guarantee K-Opacity across all devices. While this strategy cannot introduce any error to the system, whether K-Stale values are acceptable for read-only purposes is at the digression of the programmer.

3.1 GPU K-Opaque Software Transactional Memory (KoSTM)

BkTM uses novel software transactional memory model, GPU K-Opaque Software Transactional Memory (KoSTM), that prevents conflicts between Read-Only and Update transactions, while maintaining performance, data integrity, and minimizing memory overheads. GPU KoSTM allows transactions to read from old versions of shared objects as long as they are within K versions of the current version. In this work, only Read-Only transactions to use Approximately Consistency; Update transactions that might modify shared objects must use precise values.

Update Transactions detect and resolve conflicts eagerly using atomic operations and usage metadata for each shared object, shared memory access, and transaction creation. Read-Only transactions recover a version access history, use lazy conflict detection, and require no



(a) Shadow Entry. **(b)** Access Log. **(c)** Transaction meta-data.

Figure 4 GPU KoSTM Meta-Data. Optimizations for Read-Only Transactions are highlighted in orange. (a) Shadow entries store usage information for each shared memory object; (b) Access Logs are created for each in progress transaction and are used to track changes to Shadow Entries; (c) Transaction Meta-Data contains raw transaction inputs and facilitates the creation of Remote Nested Transactions.

atomic operations. GPU KoSTM guarantees strict correctness for Update transactions, as exclusive locks and strict data management guarantee data integrity. Read-Only transactions do not modify Shadow Entries or shared data values. The meta-data used for GPU KoSTM is shown in Fig. 4 as explained in the following subsections. All KoSTM meta-data is stored in GPU Global Memory. The total memory overhead of the system is the combination of all Shadow Entries, Access Logs, and Transaction Meta-Data.

3.1.1 Shadow Entries

The GPU KoSTM Shadow Entry for modifiable objects has three components: 1) the "Owning Tx ID" integer that is used to track ownership and prevent two transactions from modifying a shared object at the same time and 2) an integer representing the current version of the shared object and 3) an integer containing the version of the oldest K-Shared copy in the system. GPU KoSTM ensures that these versions never differ by more than K-1.

Read-Only transactions are allowed to access K-Shared copies of shared objects, which use a separate Shadow Entry containing a Multiple-Reader Single-Writer (MRSW) structure comprised of two version numbers and two object values. This structure allows the K-Shared object to be duplicated without the use of atomic operations, even while the writer is updating the current version. Including K-Shared objects, Shadow Entry memory overheads range from 200% if shared objects are very large up to 800% if shared objects are very small.

When an object is modified, Update transactions must update the oldest value in the K-Shared shadow and update the version number, before finalizing and releasing exclusive ownership of the shared object. Read-Only transactions with only access to K-Shared versions of the object can always retrieve a valid K-Opaque copy of the object value by 1) storing the largest version (max(a,b)), 2) copying the object value into local memory, 3) performance a CUDA threadFence, and 4) confirming that the largest version has not changed, ensuring that the copied value was not modified by any other thread.

3.1.2 Access Log

A history of all successful data accesses is stored in an undo log for Update transactions and a Read Log for Read-Only transactions. Transactions only ever use either the Undo Log OR the Read Log; the Read Log is kept separate to avoid type casting overheads.

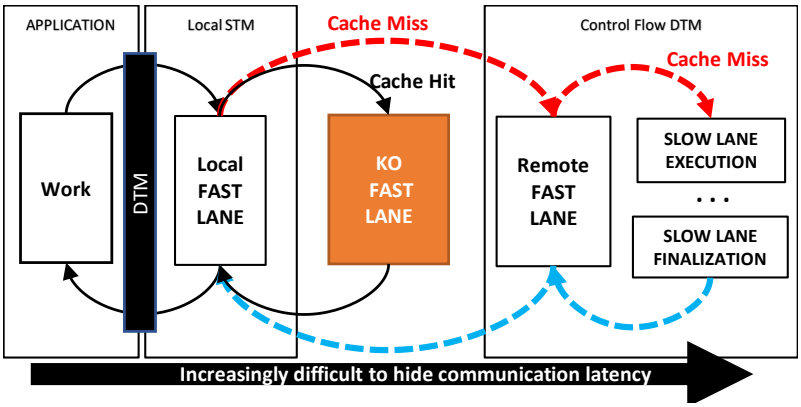


Figure 5 The Control-Flow Model is used for all Transactions. Memory accesses that resolve to the local memory partition can be completed "quickly" without remote communication ("Local Fast Lane"); local-partition misses require either a single Remoted-Nested Transaction ("Remote Fast Lane") or potentially a chain of nested transactions ("Slow Lane Execution"); BFKTM allows some Read-Only transactions to avoid created Remote-Nested Transactions by hitting the K-Shared cache ("KO Fast Lane").

At first access: Update Transactions perform an atomic compare-and-swap on the Owning Transaction ID stored in the object's Shadow Entry. If the object has no current owner, then the transaction is allowed to continue. The transaction must then check that incrementing the version counter at commit-time would not result in a KO violation by ensuring that the Shadow Entry indicates that the current version is at most K-2 versions ahead of the oldest version. If this is not the case, then the transaction is aborted.

Update transactions store the address, a copy of the current value, and increment the log size by 1. Read-Only transactions perform the same record keeping, but instead store the address and version number that was read. Depending on the size of the shared object, recording a version number can be significantly faster than creating a copy of the object.

Update transactions are eagerly aborted at access-time, while Read-Only transactions cannot be aborted until validation. This allows read-only transactions to avoid atomic operations while still guaranteeing K-Opacity.

3.1.3 Transaction Life-Cycle

The initialization and validation of Update transactions are unchanged in this work; any Update transaction still in the ACTIVE state at the time of validation has been successful. Successful update transactions increment the current version counter by 1.

Read-Only transactions must re-check all versions in the Read Log to ensure that each version used is within K versions of the current version. Accesses to the local partition are compared against the current version in the modifiable object Shadow Entry; accesses to K-Shared objects must be greater than or equal to the lowest version number in the K-Shared Shadow Entry.

3.2 Distributed TM Model: KoDTM

BkTM's DTM model uses a combination of the Control- and Data-Flow models. Control-Flow, in which Transaction execution moves between nodes to reach statically-mapped data, is used for Update transactions and Read-Only transactions when the K-Shared caching

is unsuccessful. The Data-Flow model is used exclusively for Read-Only transactions and works to maintain K-Opaque copies of remote partitions and allow Read-Only transactions to bypass remote communications.

BkTM uses a control flow model to guarantee strict correctness for Update transactions in Fig. 5 and for Read-Only transactions that miss the K-Shared cache. Transaction execution is divided into a "Fast Lane", which optimistically assumes there will be no off-device communication on the critical execution path for the transaction. When this assumption fails (i.e. in the event of a partition-miss), execution is halted and moved to the "Slow Lane", either by sending the input variables alone (as a Forward) or sending transaction-local variables to create a remote nested transaction. BkTM aims to keep execution in the Fast Lanes by allowing partition-misses to use K-Opaque copies of remote transactions as if they were the current version. Fig. 5 refers to Read-Only transactions that have avoided remote communication by exploiting K-opacity as being in the "KO Fast lane".

Fig. 6 shows the components used in the KoDTM data flow model. K-Opaque copies of remote partitions are stored on-device in the K-Opaque Cache. System-wide enforcement of the k-consistency is controlled by the Broadcaster thread on the host CPU, which scatters values to remote devices without halting GPU execution in six step process:

1) The broadcaster thread copies the local version shadow entries into host memory and then 2) copies the current local partition values. This ensures that the values copied are newer than or equal to the version numbers copied. The MRSW structure in Fig. 4a allow this copy to occur entirely asynchronously without disrupting GPU transactions. 3) Versions and values are combined into a single message that is MPI Scattered to all nodes in possession of a K-Shared copy. 4) KoDTM listener threads receive the new values, CUDA asynchronous memcpy them into device memory, and then use a stream-synchronize to ensure the values are visible. 5) The original broadcasting thread waits at an MPI Barrier until all KoDTM listener threads have copied the updated values into device memory and arrive at the barrier, indicating all threads in the system with access to the K-Shared copies now see the scattered versions. 6) The versions copied into host memory in step 1 are now copied back into device memory, now indicating the oldest existing K-Shared versions. Compared to Transaction execution, this is long latency process and update transactions may be pessimistically aborted if they are expected to cause KO violations if allowed to proceed. However, this strategy allows remote communications to start as soon as possible rather than waiting until they are initiated by the GPU. Broadcaster threads repeatedly loop through these steps for each shared partition during application execution.

To limit the negative impact on Update transactions, we allow the programmer to limit the maximum number of K-Shared copies using the "K-Share limit" parameter which is explored in our experiments.

3.3 Strict Correctness with Approximate Consistency

GPU KoSTM and KoDTM both exploit relaxed consistency by allowing Read-Only transactions to execute as if they completed at a more "convenient" time. That is to say: K-consistent values are never approximate nor are they invalid, but they may be outdated at the time of commit. Furthermore, approximate consistency does not guarantee that there was a specific "instant" where all read versions existed at the same time, rather that there was an instance where all read versions were within K modifications from the current version. KoSTM and KoDTM do not actually modify the transaction execution order histories, but instead allows certain types of conflicting transactions to proceed uninterrupted. Relaxed consistency always results in an execution order and shared memory state that are strictly correct.

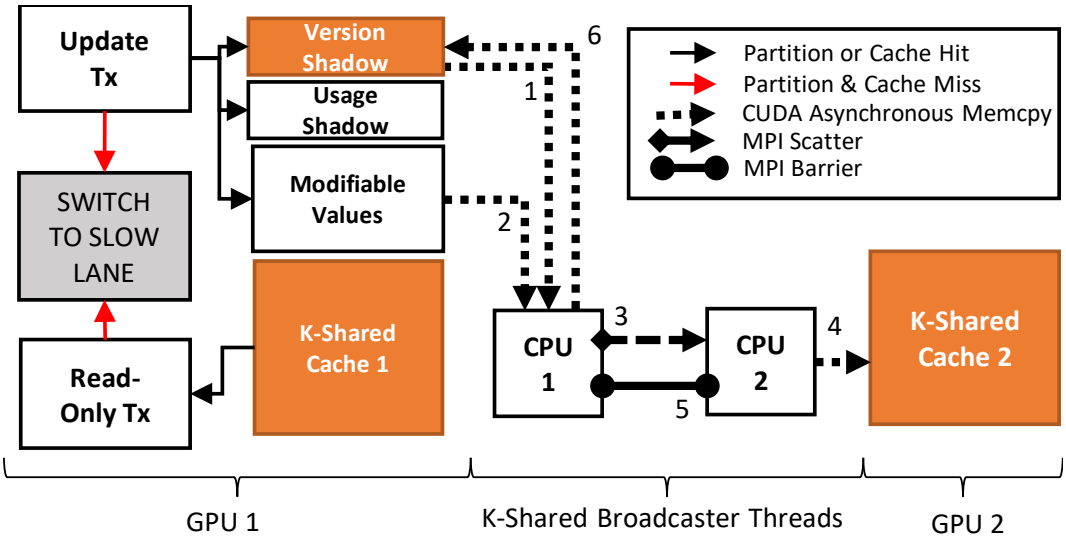


Figure 6 The Data-Flow Model is used to guarantee K-Stale values in the K-Shared cache that is used only by Read-Only transactions. The Host CPU broadcasts the local Version Shadow, and Modifiable Values to remote nodes. These broadcasts are received by remote CPUs and copied into the local K-Shared Cache. No transaction ever initiates data movement; this process is performed automatically by the system.

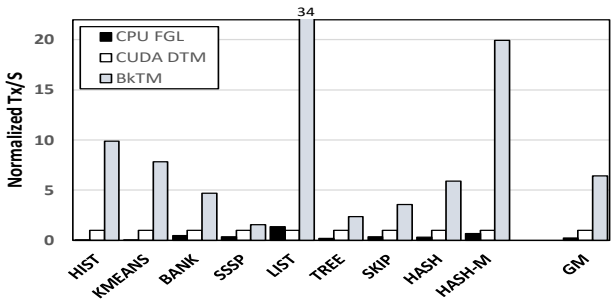
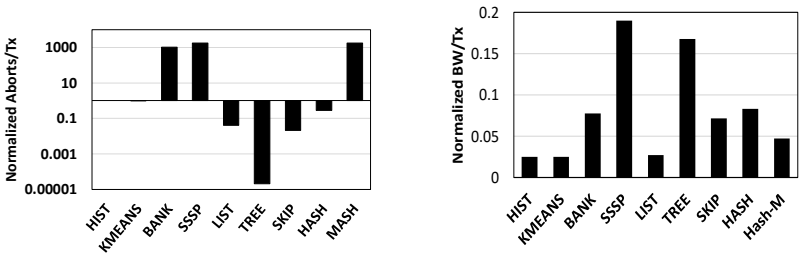


Figure 7 Performance comparisons for workloads with a 75% read-intensity (K=8, 32 Devices, No K-share limit).

Note that no "error" is ever introduced into shared memory through the use of Approximate Consistency. Any transaction that might modify a shared object is required to maintain strict atomicity, consistency, correctness, etc. The programmer must explicitly mark regions where utilization of approximate consistency is acceptable. Whether or not approximate consistency is acceptable for a given application can be subjective and requires a deep understanding of the underlying thread behavior. Programmers can only use approximate consistency in scenarios where stale values cannot be used to introduce error into shared memory.

4 Experimental Analysis

To evaluate the performance of BifurKTM we use a 16 node cluster, where each node contains two CPUs, 2.8GHz E5-2689v2 Xeon processors, and two GPUs, NVIDIA k20x. Nodes are connected using a 56 Gigabit/sec Infiniband network. Benchmarks are written in CUDA



(a) Normalized Abort rate.

(b) Normalized GPU-to-GPU Communication Overhead.

Figure 8 BkTM greatly reduces the GPU-to-GPU communication overheads, but has inconsistent effect on abort rate. Values are normalized against CUDA DTM.

and then compiled using CUDA 9.2 and MPICH 3.1.1 in Red Hat Enterprise Linux 6. We use 9 irregular memory access and dynamic data structure benchmarks used in prior works [6, 13, 10, 11]. BifurKTM is implemented in CUDA, using MPI for communication, and uses pre-processor directives to make multiple transformations of each transaction at compile-time.

The benchmarks used in this work, their workload configurations, and shared memory sizes are shown in Table 1. We consider benchmarks with behaviors favorable to the GPU (compared to the CPU) despite the presence of irregular memory accesses requiring atomic operations. Inputs are randomly generated in place to simulate the behavior of a server receiving random requests from clients. Update transactions are unchanged from their implementations in prior works. We run each benchmark in each configuration three times and report the average.

Read-Only transactions are typically short and involve retrieving a single value from an array or a dynamic data structure: HIST randomly generates a number and reads the number of times it has been generated previously; KMEANS reads the location of the cluster that is closest to a given point; BANK reads the current balance for a given bank account; SSSP, LIST, TREE, and SKIP search a dynamic data structure for a given key and return the associated value if it is found; and finally, HASH+S and HASH+M read one and 20 values from a shared hash table. Hash+M has a larger read/write set in an effort to configure the benchmark for antagonistic behavior. Note that the Read-Only transactions require atomic operations due to the presence of Update transactions that could change any value at any time.

In our first experiment, we compare the performance of BkTM to CPU FGL, where work is only done by the CPUs in the cluster and correctness is assured using Fine-Grained locking, and to CUDA DTM using Pessimistic Software Transactional Memory using the cluster configurations shown in Table 2. Performance values are normalized by the performance of the CUDA DTM. Fig. 7 shows that BkTM+K8 achieves a geometric mean speed up of 6.5x over CUDA DTM, and a speedup of 18x over CPU FGL, when using an K=8 and having no limit on the number of K-Shared read-only copies. BkTM's speedups are primarily attributed to significant reductions in GPU-to-GPU communication overheads despite some increases in the abort rate as shown in Fig. 8. Note that aborting local Update Transactions to ensure cluster-wide K-Opacity requires no communication and thus these aborts are much cheaper than Remote Nested Transaction aborts. BifurKTM eliminates some expensive Remote-Nested aborts in favor of many cheap local aborts, making aborts cheaper on average compared to CUDA DTM. Compared to CPU FGL, BkTM additionally benefits from the GPU's 100x higher computational throughput and 5x higher memory bandwidth.

Table 1 Benchmark Configurations.

Benchmark	Read-Only	Update		Shared Data	
	Tx Reads	Tx Reads	Tx Writes	Object Size (B)	Total Size
HIST	1	1	1	4	70MB
KMEANS	1	1	1	4K	70MB
BANK	1	2	2	4	500MB
SSSP	1	2	1	8	150MB
LIST	1	1	2	8	150MB
TREE	1	1	2	12	100MB
HASH-S	1	1	2	8	150MB
SKIP	1	8	9	36	150MB
HASH-M	20	20	20	4	500MB

HIST and KMEANS perform very long computations before performing very short transactions with low conflict rates. We observe that BkTM has no noticeable impact on the abort rate in Fig. 8a, and reduces communication overheads by 98%, yielding a 7x speedup over CUDA DTM.

BANK and SSSP have very short critical sections with very little time spent outside of the transaction. These behaviors are very disadvantages for GPU clusters unless the workloads exhibit high partition-locality, which limits the usefulness of the K-Shared cache. Having short transactions and high transactional through-puts, the BANK and SSSP both incur massive increases in the abort rate, as update transactions must be pessimistically aborted to ensure K-consistency between K-shared broadcasts. Both benchmarks achieve a modest speedup owing to the 80% reduction in GPU-to-GPU communication overheads.

Dynamic data structure benchmarks show strong speedups proportional to the average search time of the structure. LIST, TREE, SKIP, and HASH all see large decreases in the abort rate as well as the communication overheads owing to the compounding advantages of reduced local contention from 1) GPU KoSTM preventing conflicts between Read-Only and Update transactions, 2) KoDTM allowing at least 80% of remote communication to be avoided, 3) fewer changes to the shared data structure prevents non atomic insertion/retrieval location searches from being repeated. TREE achieves the lowest speedup of 2.4x owing to the very short O(logN) search times and corresponding high update frequency; while list, with a significantly longer O(N) search time, achieves the largest speedup of 34x.

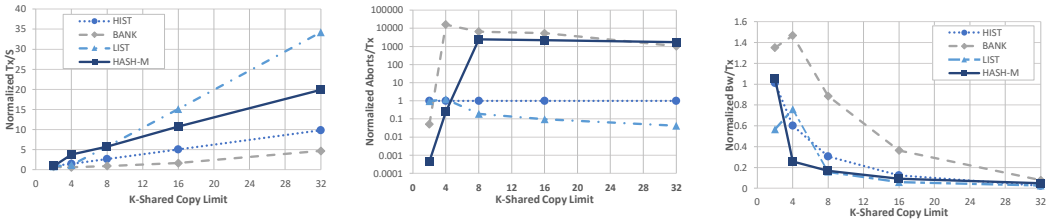
Finally, we observe a 20x speedup for the HASH-M benchmark, which in our experiments performance 20 simultaneous insertions into a shared hash table backed by linked lists. HASH-M's abort rate is greatly increased, as 20 modifications per Update transaction forces abortions to prevent KO violations. The large speedup is attributed to the 95% reduction in GPU-to-GPU communication overheads as shown in Fig. 8b, as all Read-Only transactions hit the K-Share cache. Note that the communication overheads discussed in this work do not include CPU-initiated communications which are used for broadcasting K-Shared values. These broadcasts do compete for communication bandwidth, but they are not along the critical path for any particular transaction and thus do not directly impact performance like a GPU-initiated communication does.

In our experiments, we observe large performance improvements despite huge increases in the abort rate due to the cost asymmetry between a local abort-and-retry, which requires no off-device communication, and the creation of a remote-nested transaction when reading data outside of the local partition which can have a six order-of-magnitudes longer latency caused by remote communication on the critical path. At larger cluster sizes or shared data sizes, the bandwidth between devices will become a critical bottleneck preventing the usage of a read-only copy on every remote node.

Table 2 Cluster Configurations.

Name	Description	Devices	Max Threads per device
CPU FGL	CPU cluster using Locking	64	20
CUDA DTM	GPU cluster using DTM	64 + 64	4096x1024 + 20 on Host
BkTM+KX	GPU cluster usingX-Opaque BkTM	64 + 64	4096x1024 + 20 on Host

In this event, the programmer can either 1) restructure data accesses to decrease partition-miss-rate, or 2) relax K until bandwidth no longer limits update frequency (as shown in Fig. 10b).

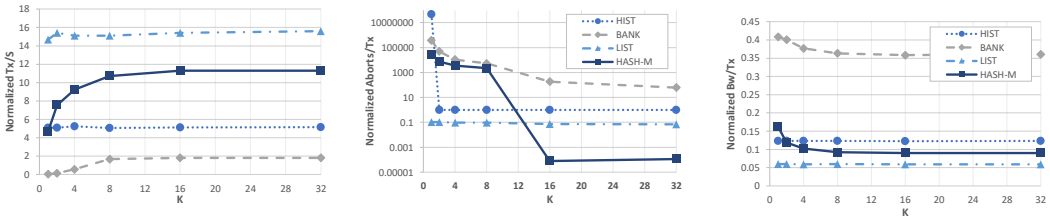


(a) Speedup over BkTM with strict consistency.

(b) Aborts per Committed Transaction.

(c) Communication overhead.

Figure 9 Impact of limiting the number of K-Shared copies on BkTM performance (K=8).



(a) Speedup over BkTM with strict consistency.

(b) Aborts per Committed Transaction.

(c) Communication overheads.

Figure 10 Impact of relaxed consistency on BkTM performance (K-Share limit of 16).

4.1 Balancing Staleness and Throughput

In this section, we study the effect of relaxing consistency and limiting the number of K-Opaque read-only copies of each partition for workloads with a read-intensity of 75%. These two levers allow the programmer to balance two key trade-offs of the BkTM design. We use the shorthand "BkTM+KX" to refer to BkTM configurations with different degrees of approximate consistency, where X=0 refers to "strict consistency".

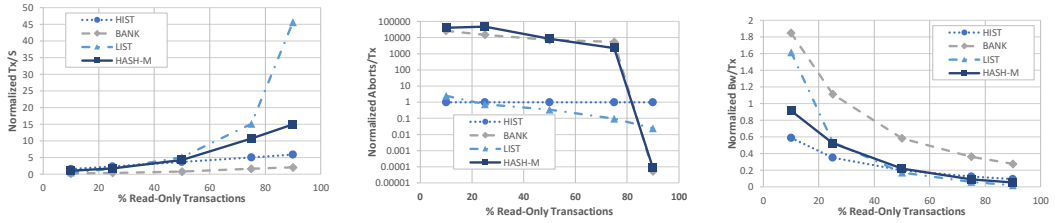
In Fig. 9 we observe that BkTM+K8 achieves a geometric mean speedup of 13.3x over BkTM+K0 due to a 96% reduction in communication overheads. To enforce cluster-wide K-Opacity while using K-Shared data copies, BkTM+K8 incurs a massive increase in the abort-rate. The abort-rate can be mitigated allowing RO transactions to read staler versions by increasing K. We observe that the best performing version of all benchmarks has an unlimited number of K-Shared partition copies, despite the large increases in abort rate, while still enforcing a modest staleness limit of K=8.

As the system allows more K-Shared copies of each partition, the time required to update those copies increases and the limit on the rate at which shared objects can change is lowered. In Fig. 9b we observe a sharp increase in the abort rate for the HASH-M benchmark at a K-Share limit of 2: the time required to update 2 remote partition copies becomes a bottleneck for local update transaction performance. Similarly, maintaining even one K-Shared remote copy hurts BANK update transaction performance. For both benchmarks, the increased throughput of read-only transactions offsets the negative impact on update transactions.

On the other hand, in Fig. 9c we observe spikes in the communication overheads for the BANK and LIST benchmarks as the costs of KO-violation abort & retry are not offset by the RO advantages. We observe that maintaining the K-Shared cache hurts BkTM performance by 24% when allowing only one K-Shared copy of each partition. BkTM+K8 does not show a speedup for the BANK benchmark until 8 devices are allowed to have K-Shared copies of remote partitions.

In Fig. 10, we observe that increasing the degree of Opacity, K, allows update transactions to make more modifications between partition broadcasts to remote readers which can either 1) reduce the number of local aborts caused to ensure global K-Opacity; OR 2) increasing the number of K-Readers, which requires larger, slower broadcasts. Increasing K has a limited impact on RO transactions on K-Sharing nodes as the local copy is guaranteed to be K-Opaque on first access. We observe that increasing K slightly reduces communication overheads in Fig. 10c due to the significantly reduced abort rate.

Increasing K will have a binary effect on update transaction throughput: either K is greater than the number of updates that occur between K-Reader broadcasts or it is not. In Fig. 10b we observe that HIST's update frequency is low and thus increasing K has no impact on the abort rate. In contrast, BANK has a very high frequency and even K=32 is not enough to prevent aborts between K-Reader updates. We see HIST and HASH-M cross break-even points at K=2 and K=16 respectively, where we see 99.9% reductions in their abort-rates. Whether or not a possible staleness of 15 is worth the increased update-throughput is at the discretion of the programmer.



(a) Speedup over BkTM without relaxed consistency.

(b) Aborts Rate.

(c) Communication Overheads.

Figure 11 Impact of workload composition on BkTM performance (K=8, 32 Read-Only Copies).

4.2 Workload Sensitivity

In this study we study the performance of BkTM, normalized by BkTM without relaxed consistency, as the read-intensity of the workload changes. We here define read-intensity as the percentage of all committed transactions that anticipated that no changes would be made to shared memory before the transaction began. The STM system is informed of anticipated read-only execution using information provided by the programmer. Here we use K=8 and allow all 32 nodes to store K-Opaque read-only copies of the full PGAS.

In Fig. 11, we observe large speedups driven primarily by huge reductions in communication overheads for very read-intensive workloads, despite large increases in the abort rate in some cases. With a read-intensity of 10% we observe that BkTM+K8 achieves only 70% of the performance of BkTM+K0 as the majority-update workload does not benefit from the read-optimizations. Poor performance can be attributed to the maintenance of remote read-only partition copies, which can only be K versions behind the local version. These overheads can be hidden with a read-intensity of 25% where BkTM+K8 gains a 1.3x speedup.

At the maximum read-intensity test, 90%, BkTM+K8 shows a geometric mean speedup of 9.6x over BkTM+K0. The LIST benchmark shows particularly large performance improvements owing to the compounding effects of low update frequency, more reliable pre-transaction insertion-point searches, greatly reduced communication overheads, and a greatly reduced abort rate from prevent conflicts between read-only transactions.

In Fig. 11b all benchmarks except HIST show exponentially decreasing abort rates as the read-intensity increases; the abort rate and update frequency of HIST are both so low that the abort rate is unaffected by varying the read-intensity. HASH-M and BANK show greatly increased caused by the high shared-object update frequency causing update transactions to abort and restart while waiting for remote partition copies to be updated. With a read-intensity of 90%, the update frequency is finally low enough that the broadcaster no longer causes aborts to ensure K-Opacity. LIST crosses a similar breaking point at read-intensity of 25%. The exact value for this update frequency is different for each benchmark and is determined by the bandwidth between devices, shared object sizes, and how easily messages can be batched and sent to the same destination node.

5 Conclusion

In this work, we explore the performance of BifurKTM (BkTM), which allows programmers to improve GPU cluster performance by allowing Read-Only transactions to execute at a more convenient time. By rearranging the apparent execution order, BkTM can greatly reduce the conflict rate between transactions and nearly eliminate the communication overheads. We demonstrate that a GPU cluster using BkTM, even workloads with a low read-intensity, can greatly outperform a CPU cluster despite irregular memory accesses and the overheads of accessing distributed shared memory.

Though GPU clusters using BKTM remain sensitive to workload composition, we increase GPU cluster flexibility and achieve a greater high performance range using relaxed consistency. This work relies on the programmer to recognize such opportunities by understanding the underlying application behavior. Finally, the BkTM incurs at least a 200% memory overhead due to the requirement for storing multiple copies of shared objects. We expect future technologies to further relax these constraints by increasing GPU on-device memory, off-chip bandwidth, and increasing the overall diversity of GPU applications.

References

1 Basem Assiri and Costas Busch. Approximate consistency in transactional memory. *International Journal of Networking and Computing*, 8(1):93–123, 2018.

2 Daniel Castro, Paolo Romano, Aleksandar Ilic, and Amin M Khan. Hetm: Transactional memory for heterogeneous systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 232–244. IEEE, 2019.

3 Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a software transactional memory for graphics processors. In *EGPGV*, pages 121–129, 2010.

4 Sui Chen and Lu Peng. Efficient gpu hardware transactional memory through early conflict resolution. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 274–284. IEEE, 2016.

5 Sui Chen, Lu Peng, and Samuel Irving. Accelerating gpu hardware transactional memory with snapshot isolation. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 282–294. IEEE, 2017.

6 Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

7 Wilson WL Fung, Inderpreet Singh, Andrew Brownsword, and Tor M Aamodt. Hardware transactional memory for gpu architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 296–307. ACM, 2011.

8 Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

9 Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20:195–208, 2007.

10 Anup Holey and Antonia Zhai. Lightweight software transactions on gpus. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 461–470. IEEE, 2014.

11 Samuel Irving, Sui Chen, Lu Peng, Costas Busch, Maurice Herlihy, and Christopher Michael. Cuda-dtm: Distributed transactional memory for gpu clusters. In *Proceedings of the 7th International Conference on Networked Systems*, 2019.

12 Jiri Kraus. An introduction to cuda-aware mpi. *Weblog entry]. PARALLEL FORALL*, 2013.

13 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.

14 Sudhanshu Mishra, Alexandru Turcu, Roberto Palmieri, and Binoy Ravindran. Hyflowcpp: A distributed transactional memory framework for c++. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 219–226. IEEE, 2013.

15 John Nickolls, Ian Buck, and Michael Garland. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 40–53. IEEE, 2008.

16 Mohamed M Saad and Binoy Ravindran. Snake: control flow distributed software transactional memory. In *Symposium on Self-Stabilizing Systems*, pages 238–252. Springer, 2011.

17 Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata. Toward a software transactional memory for heterogeneous cpu–gpu processors. *The Journal of Supercomputing*, pages 1–16, 2017.