Hardware Transactional Memory for GPU Architectures

Wilson W. L. Fung† Inderpreet Singh† Andrew Brownsword Tor M. Aamodt†
Department of Computer and Electrical Engineering
†University of British Columbia
wwlfung@ece.ubc.ca isingh@ece.ubc.ca
andrew@brownsword.ca aamodt@ece.ubc.ca

ABSTRACT

Graphics processor units (GPUs) are designed to efficiently exploit thread level parallelism (TLP), multiplexing execution of 1000s of concurrent threads on a relatively smaller set of single-instruction, multiple-thread (SIMT) cores to hide various long latency operations. While threads within a CUDA block/OpenCL workgroup can communicate efficiently through an intra-core scratchpad memory, threads in different blocks can only communicate via global memory accesses. Programmers wishing to exploit such communication have to consider data-races that may occur when multiple threads modify the same memory location. Recent GPUs provide a form of inter-block communication through atomic operations for single 32-bit/64-bit words. Although fine-grained locks can be constructed from these atomic operations, synchronization using locks is prone to deadlock. In this paper, we propose to solve these problems by extending GPUs to support transactional memory (TM). Major challenges include supporting 1000s of concurrent transactions and committing non-conflicting transactions in parallel. We propose KILO TM, a novel hardware TM design for GPUs that scales to 1000s of concurrent transactions. Without cache coherency hardware to depend on, it uses word-level, value-based conflict detection to avoid broadcast communication and reduce on-chip storage overhead. It employs speculative validation using a novel bloom filter organization to increase transaction commit parallelism. For a set of TM-enhanced GPU applications, KILO TM captures 59% of the performance of fine-grained locking, and is on average 128× faster than executing all transactions serially, for an estimated hardware area overhead of 0.5% of a commercial GPU.

Categories and Subject Descriptors

C.1.4 [**Computer System Organization**]: Processor Architectures—*Parallel Architectures*; D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Design, Performance

1. INTRODUCTION

Recently, there has been much interest in the use of GPUs to provide cost-effective parallel performance. GPUs have the potential to provide higher computation per unit cost since they devote a larger fraction of their area to functional units rather than scheduling logic. Applications that benefit from running on a GPU tend to contain plenty of data parallelism coupled with regular memory access patterns that ensure efficient use of off-chip memory bandwidth. In this work, we explore a generic manycore accelerator architecture similar to contemporary GPUs from NVIDIA and AMD and focus on the challenge of running applications that require some form of synchronization.

A common problem with using lock-based synchronization is the potential for deadlocks. Manycore accelerators such as GPUs use more threads to deliver more throughput at a lower cost and energy consumption. More threads, and correspondingly larger problem sizes, exacerbate the challenge of lock-based programming by increasing the number of ways in which a deadlock can manifest [4].

One obstacle preventing GPUs from tackling a wider variety of application workloads is the difference between CPU and GPU programming models. This is particularly true for synchronization primitives, which perform poorly and can interact in intricate ways (from a programmer perspective) with the underlying multi-threaded execution hardware on a GPU. We believe it is necessary to simplify the programming of highly parallel workloads that require data synchronization. Enabling transactional memory (TM) [28] on GPUs simplifies synchronization, and provides a powerful programming model that promotes fine grained communication and strong scaling of parallel workloads. This work focuses on enabling TM on a GPU efficiently and evaluates a detailed implementation to understand the trade-offs involved.

The contributions of this paper are:

- It proposes the use of hardware transactional memory (HTM) for GPU computing.
- It proposes KILO TM, a novel, scalable HTM design. The design combines aspects of value-based conflict detection [42, 19], RingSTM [49], and Scalable TCC [16] (Transactional Coherence and Consistency) to support 1000s of concurrent transactions without requiring a cache coherency protocol. It detects conflicts at word-level granularity and employs speculative validation to increase transaction commit parallelism.
- It proposes an extension to SIMT [35] hardware to handle control flow divergence due to transaction aborts.
- It introduces the *recency bloom filter* which incorporates a notion of time and supports implicit, multi-item removal.
- It evaluates the potential of transactional memory on a set of GPU computing workloads that employ transactions.
- It describes an extension to the GPU hardware thread scheduler to control transaction concurrency, and shows that the mechanism benefits high-contention workloads.

Our evaluation shows that KILO TM captures 59% of the performance of fine-grained locking. We find that KILO TM outperforms fine-grained locking for low contention applications that require

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO '11, December 3-7, 2011, Porto Alegre, Brazil
Copyright 2011 ACM 978-1-4503-1053-6/11/12 ...$10.00.

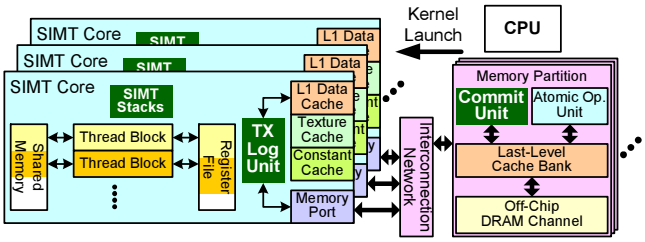


Figure 1: High-Level GPU architecture exposed by the CUDA programming model. TX Log Unit, Commit Unit added for KILO TM. SIMT stack modified to support transactions.

acquiring multiple locks to enter a critical section. On the other hand, we find that TM applications ported from CPU-optimized versions can perform poorly on GPUs regardless of the data synchronization mechanism used (fine-grained locking or TM). Optimizing these applications for GPUs would involve redesigning the algorithm and data structures, possibly requiring data synchronization at a much finer-granularity to expose more parallelism. Doing so without TM would require rewriting much of the application before testing for correctness, a risky investment that might deter wider adoption of GPU computing for these types of applications. Our estimation with CACTI [47] indicates that implementing KILO TM on an NVIDIA Fermi GPU [40] would increase area by only 0.5%, a small overhead for the large increase in programmability.

The rest of this paper is organized as follows: Section 2 briefly reviews relevant background information. Section 3 motivates TM on GPUs, and outlines the challenges in adopting prior HTMs on GPUs. Section 4 describes KILO TM, our TM design for a GPU that supports 1000s of concurrent transactions. Section 5 describes our methodology and benchmark applications, Section 6 presents results, Section 7 discusses related work, and Section 8 concludes.

2. BACKGROUND

2.1 Transactional Memory

Transactional memory [28, 27] simplifies software development for parallel architectures by providing the programmer with the illusion that blocks of code, called *transactions*, execute atomically. With TM, the programmer does not need to write code with locks to ensure mutual exclusion. For example, in the ATM benchmark (Section 5), each scalar thread uses a transaction to specify that funds should be debited from one account and deposited into another account as a single action. If two threads try to access the same account at the same time, one transaction will be restarted. In the absence of such conflicts, all transactions can perform the transfer in parallel.

2.2 Baseline GPU Architecture

Figure 1 shows the high-level organization of our baseline GPU-like manycore accelerator that runs CUDA and OpenCL applications [39, 41, 31]. A CUDA program starts on a CPU and then launches parallel *compute kernels* onto a GPU. Each kernel launch dispatches a hierarchy of threads (a *grid* of *blocks* of *warps* of scalar *threads* running the same compute kernel) onto the accelerator. Blocks are allocated as a single unit of work to a heavily multi-threaded SIMT core. Threads within a block can communicate via an on-chip *shared memory*[1]. The SIMT cores access a distributed, shared, read/writeable last-level (L2) cache and off-chip DRAM via an on-chip network.

With the SIMT execution model, scalar threads are managed as a SIMD execution group called a warp (or wavefront in AMD terminology). Each warp contains 32 scalar threads [41].

[1] 16kB scratchpad memory per SIMT core.

SIMT Stack - Branch Divergence Hardware. Each warp has a SIMT stack that serializes the execution of different subsets of threads that diverge to different control flow paths. We summarize the SIMT stack mechanism in our baseline below [23, 24].

If some threads in a warp have different outcomes when a branch executes, i.e. the branch diverges, new entries are pushed onto the warp's SIMT stack. Each entry contains a reconvergence PC (RPC) which is set to the immediate post-dominator[2] of the branch. Each bit in the active mask indicates whether the corresponding thread follows the control flow path corresponding to the stack entry. The PC of the top-of-stack (TOS) entry indicates the target path of the branch. Reaching the reconvergence point is detected when the next PC equals the RPC at the TOS entry. When this occurs, the top of the stack is popped (current GPUs use special instructions to manage the stack [34, 18, 3]). This switches execution to the next branch target that is to be executed by the other subset of threads. After all threads reach the reconvergence point, the TOS entry will reveal a full active mask with the reconvergence PC of the divergent branch, indicating that the threads have effectively reconverged.

Memory Subsystem. When a warp executes a memory instruction, each scalar thread in the warp sends a scalar memory access to the memory subsystem in the SIMT core. For shared memory accesses, each shared memory bank processes conflicting accesses in successive cycles. For *global* and *local* memory spaces [41], accesses from different threads in the same warp to the same cache line are merged (coalesced) into a single wide access. The L1 data cache services these wide accesses one per cycle.

Each thread has access to a private *local memory space* [41], which is used mainly for register spilling. The local memory is stored in off-chip DRAM and cached in the per-core L1 data cache and the last-level (L2) cache. It is organized such that consecutive 32-bit words are accessed by consecutive scalar threads in a warp. When all the threads in a warp are accessing the same address in their own local memory space, their accesses can be coalesced into a single wide access that can be serviced by the L1 data cache in a single cycle.

The L1 data caches in the SIMT cores are not coherent. Similar to NVIDIA Fermi [40], a write hit at a memory location in the *global memory space* [41] evicts the cache line from the L1 and sends the updated contents to the L2 cache bank at the corresponding memory partition. A stale version of the same cache line may be cached in another SIMT core. To ensure that a global memory access always returns coherent data, the CUDA application can configure the GPU hardware to skip the L1 cache for all global memory accesses [41]. These accesses are then serviced directly by the L2 cache.

Atomic Operations. Current GPUs provide hardware atomic operations for simple single-word read-modify-write operations [41, 31]. These are implemented by extending alpha blending hardware [54, 37] (Atomic Op. Unit in Figure 1) to perform these read-modify-write operations to individual locations atomically.

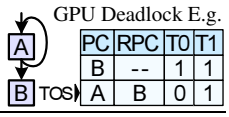
3. TRANSACTIONAL MEMORY ON GPU: OPPORTUNITIES AND CHALLENGES

Atomic operations on current GPUs enable implementation of locks, allowing complex irregular algorithms [12]. Fine-grained locking enables higher concurrency in applications, but requires the application developer to consider all possible interactions between locks to ensure deadlock-free code – a challenging task [33]. With tens of thousands of threads running concurrently on a GPU, the number of possible interactions among these fine-grained locks can

[2] Closest point in the program that all paths leaving the branch must go through before exiting the function.

Example 1 CPU spin-lock code. CAS = compare-and-swap.

```
A: while(CAS(lock,0,1)==1);
B: // Critical Section ...
C: lock = 0;
```



Example 2 Spin-lock implementation on GPU to avoid deadlock due to implicit synchronization in warps [1].

```
A: done = 0;
B: while(!done){
C:   if(CAS(lock,0,1)==0){
D:     // Critical Section ...
E:     lock = 0;
F:     done = 1;
G:   }
H: }
```

be overwhelming in practical applications. This problem is well known to the supercomputing community and has inspired special debugging tools to summarize thread behaviours for deadlock/data-race analysis [4]. Concurrent with this work, IBM has announced support for TM in their upcoming BlueGene/Q supercomputer [5].

In this paper, we propose to increase support for irregular algorithms on GPUs by extending GPU architecture to support TM [28]. While originally proposed for CPUs, we find TM to be a natural extension to the existing GPU/CUDA programming model. From a programmer's perspective, a transaction is executed as an atomic block of code in isolation. A thread in a transaction is never blocked waiting to synchronize with another thread. This is important because a CUDA/OpenCL application can launch many more threads than the GPU hardware can concurrently execute. Like transactions, thread execution sequencing is abstracted away in the CUDA programming model. The hardware thread schedulers on current GPUs can execute transactions with simple extensions.

In addition to the traditional deadlock problem, GPU application developers have to deal with interactions between the SIMT stack and atomic operations. Example 1 shows how a critical section may be guarded by the acquisition and release of a fine-grained lock (line A and C) on the CPU. On a GPU, this code may deadlock [1]. This can happen if the threads in the same warp attempt to acquire the same lock at line A. For example, consider a warp with two threads, T0 and T1, both trying to acquire the same lock. T0 succeeds and exits the loop, but waits at the start of the critical section (line B) for reconvergence, while T1 still spins in the loop (see inset at right in Example 1). T1 will continue spinning and waiting for the lock held by T0 and never exit, forming a deadlock. To remove the deadlock, the program must be modified to use code similar to Example 2. This issue is known among GPU application developers [1] and explored in more detail by Ramamurthy [43]. With TM, the GPU hardware can be designed to handle such interactions between transactions and the SIMT stack (see Section 4.1).

Figure 2 compares the performance of a set of GPU TM applications (described in Section 5) running on an ideal GPU TM system against fine-grained lock versions of the applications. In this ideal TM system, TM overheads related to detecting conflicting transactions are removed. The performance shown is normalized to that obtained by serializing all transaction executions via a single global lock. On average, the applications running on ideal TM achieve 279× speedup over serializing all transactions, which is 24% faster than fine-grained locking.

Table 1 shows the IPC of these applications with the ideal TM system and fine-grained locking. Some of our applications (CL, BH and CC) achieve reasonable performance, while others suffer from GPU performance bottlenecks such as control flow and memory divergence (see Section 6.2 for further discussion). We believe these

Table 1: Raw performance (IPC) of applications described in Section 5 (Peak IPC = 240).

Applications →	HT-H	HT-L	ATM	CL	BH	CC	AP
Ideal TM	6.6	5.9	4.2	9.4	10.5	33.4	0.5
FG Lock	8.1	6.5	4.2	8.8	9.5	51.0	0.5

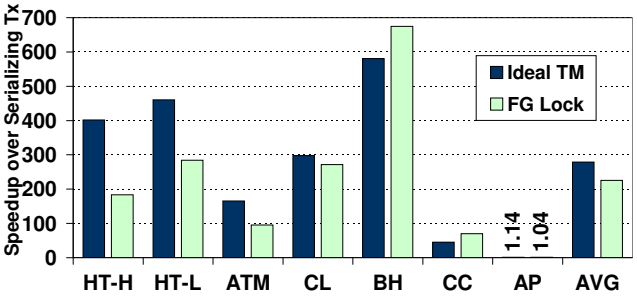


Figure 2: Performance comparison between applications running on an ideal TM system and their respective fine-grained (FG) locking version (applications described in Section 5).

applications can be optimized via performance tuning – identifying bottlenecks and redesigning the applications incrementally to address these bottlenecks one by one. Performance tuning is beyond the scope of this work. Transactional memory arguably provides an easier programming model for performance tuning because it allows GPU application developers to rework algorithms and data structures without concern for deadlock. This work focuses on enabling TM on GPUs efficiently, and with minimum overhead.

3.1 Challenges with Prior HTMs on GPUs

Hardware transactional memory (HTM) has been researched extensively. Many proposed HTMs leverage cache coherence for conflict detection among concurrent transactions, while assuming that each transaction owns a private L1 cache. Even though recent GPUs have caches [40], the caches local to a SIMT core are not coherent and are shared among 100s of threads that execute concurrently on the core. GPUs are designed to exploit fine-grained data parallelism; adjacent memory words are often accessed by different threads. These differences raise many challenges in adopting existing HTMs for GPUs.

An emerging class of manycore accelerators, such as Intel's Larrabee [46], feature fewer concurrent threads per core and coherent caches that can be partitioned per thread. The following challenges may be less severe for this class of manycore accelerators.

3.1.1 Access Granularity and Write Buffering

Each line in the L1 data caches could be extended to identify and isolate speculative data written by individual transactions. However, each transaction might obtain only a few cache lines before the cache overflows because there are fewer L1 cache lines than scalar threads on a SIMT core. Transactions typically lack the spatial locality required to fully use a cache line and make poor use of the few lines they can access. Furthermore, the fine-grained, interleaved accesses among different threads can introduce significant false-sharing and reduce the accuracy of conflict detection.

3.1.2 Transaction Rollback

Many proposed HTMs checkpoint the architectural state of the hardware thread at the start of a transaction for restoration upon rollback. Maintaining copies for 10s of registers at transaction boundaries in a CPU core is relatively cheap. GPUs, however, are designed to execute 1000s of concurrent threads, and spend significant hardware resources on register file storage. NVIDIA Fermi has 2MB of register file storage, which exceeds its aggregate cache capacity [40]. Naively checkpointing this many registers would introduce significant overheads.

3.1.3 Scaling Conflict Detection

A key challenge for scaling TM beyond 1000s of concurrent transactions is designing a conflict detection mechanism that works effectively at this scale. Naive broadcast-based conflict detection scales poorly; T concurrent transactions will broadcast to T-1 other transactions, generating $O(T^2)$ traffic.

Many proposed TMs use global metadata, such as a cache coherence directory, to eliminate unnecessary traffic. Recently, directory based cache coherence protocols supporting up to 1000 cores have been proposed [29, 58, 22]. However, GPUs such as Fermi [40] do not have a private cache for each thread.

Using bloom filters to represent read- and write-sets of a transaction [56, 15, 36] allows each thread to quickly react to incoming requests and enables fine-grained conflict detection. We experimented with an ideal version of a signature-based HTM (lazy conflict detection and lazy version management) with each transaction maintaining both its read- and write-sets in a bloom filter. We used the parallel bloom filters described by Sanchez *et al.* [45]. Each filter contained 4 separate sub-signatures and each sub-signature was indexed by a unique H_3 hash function. We had to use a 1024-bit filter per thread (3.8MB of total storage for 30720 threads) to keep the false conflict rate below 20% for the benchmarks CL, BH and AP. Using 512-bit filters increased the rate to 60%.

3.1.4 Commit Bottleneck

Even if we reduce the bloom filter storage by limiting the number of concurrent transactions (Section 4.6), the bloom filters of all transactions and the directory cannot be modified when one of the transactions is committing. Otherwise, a conflicting access may go undetected when a transaction that has resolved all of its conflicts is updating memory [16]. Scalable TCC [16] solves this issue by locking entries in the directory, but its commit protocol serializes transaction commit at each directory bank. LogTM-SE [56] uses eager version management, writing speculative data directly to global memory, to allow transactions to commit in parallel. However, data isolation of each transaction is maintained by eager conflict detection via a cache coherence protocol. The potential commit bottleneck and the signature storage explosion (Section 3.1.3) issue persuaded us towards exploring alternatives.

4. KILO TRANSACTIONAL MEMORY

In this section, we present KILO Transactional Memory, a TM system scalable to 1000s of concurrent transactions. KILO TM does not leverage a cache coherence protocol for conflict detection among running transactions. Instead, each transaction performs *word-level, value-based conflict detection* against committed transactions by comparing the saved value of its read-set against the value in memory upon its completion [42, 19]. A changed value indicates a conflict. This mechanism offers weak isolation [27]. Each transaction buffers its saved read-set values and memory writes in a read-log and a write-log (in address-value pair) in local memory (lazy version management). When a transaction finishes executing, it sends its read- and write-log to a set of commit units for conflict detection (validation), each of which replies with the outcome (pass/fail) back to the transaction at the core. Each commit unit validates a subset of the transaction's read-set. If all commit units report no conflict detected, the transaction permits the commit units to publish the write-log to memory. To improve commit parallelism for non-conflicting transactions, transactions speculatively validate against committed transactions in parallel, leveraging the deeply pipelined memory subsystem of the GPU. The commit units use an address-based conflict detection mechanism to detect conflicts among these transactions (we call these *hazards* to distinguish them from the conflicts detected via value comparison). A hazard is resolved by revalidating one of the conflicting transactions at a later time. Section 4.5 describes the protocol in detail.

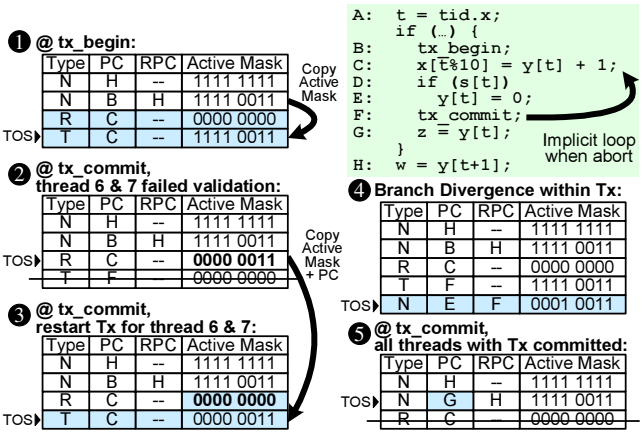


Figure 3: SIMT stack handling divergence due to transaction aborts (validation fail). Thread 6 and 7 have failed validation and are restarted. Stack entry type: Normal (N), Transaction Retry (R), Transaction Top (T). For each scenario, added entries or modified fields are shaded.

In KILO TM, transaction-specific communication (conflict detection and memory updates) occurs only between the commit units and the committing thread (shown in Figure 4a). This restriction permits the communication packets from different threads to be pipelined and interleaved, as long as the end-to-end message order between the SIMT cores and the commit units is maintained. KILO TM restricts each transaction to have a single entry and a single exit, matching '`atomic{}`' semantics in common TM language extensions [27]. Transaction boundaries are conveyed to hardware with `tx_begin` and `tx_commit` instructions in the compute kernel. Nested transactions are flattened [27] into a single transaction.

Figure 1 highlights the changes required to implement KILO TM on our baseline GPU architecture. These include an extension to the SIMT stack, a Transaction Log Unit, and a Commit Unit.

4.1 SIMT Stack Extension

When a warp finishes a transaction, each of its active threads will try to commit. Some of the threads may abort and need to reexecute their transactions, while other threads may pass the validation and commit their transactions. Since this outcome may not be unanimous across the entire warp, a warp may diverge after validation.

Figure 3 shows how the SIMT stack can be extended to handle control flow divergence due to transaction aborts. When a warp enters the transaction (at line B, `tx_begin`), it pushes two special entries onto the SIMT stack (❶). The first entry of type R stores information to restart the transaction. Its active mask is initially empty, and its PC field points to the instruction after `tx_begin`. The second entry of type T tracks the current transaction attempt. At `tx_commit` (line F), any thread that fails validation sets its mask bit in the R entry. The T entry is popped when the warp finishes the commit process (i.e., its active threads have either committed or aborted) (❷). A new T entry will then be pushed onto the stack using the active mask and PC from the R entry to restart the threads that have been aborted. Then, the active mask in the R entry is cleared (❸). If the active mask in the R entry is empty, both T and R entries are popped, revealing the original N entry (❺). Its PC is then modified to point to the instruction right after `tx_commit`, and the warp resumes normal execution. Branch divergence of a warp within a transaction is handled in the same way as non-transactional divergence (❹).

4.2 Scalable Conflict Detection

Section 3.1.3 discussed how signature-based conflict detection is prone to the commit bottleneck and storage explosion when scaled to 1000s of threads. Typical conflict detection used in HTMs checks the existence of conflicts *and* identifies the specific conflicting transactions. One insight Spear *et al.* present with RingSTM [49] is that a committing transaction only needs to detect the existence of conflicts with transactions that have committed. Transactions with detected conflicts can self-abort without interfering with execution of other running transactions. This reduces storage and traffic requirements because the TM system does not need to maintain a set of in-flight sharers/modifiers for each memory location, and each transaction only performs the detection once before it commits. However, in our experiment with RingSTM, we had to use 512-bit write-signatures in the commit record ring to keep the false conflict rate below 40% (1.9MB of total storage for a ring with 30720 records to support 30720 concurrent transactions). Value-based conflict detection [42, 19] exhibits similar traffic requirements as RingSTM. Transactions detect conflicts with other committed transactions, but without using any global metadata – only values from global memory are used. KILO TM combines aspects of RingSTM and value-based conflict detection in hardware, and extends them to permit concurrent validations (Section 4.5).

A transaction is *doomed* if it has observed an inconsistent view of memory (e.g., in between memory reads to two different locations, another transaction has committed and updated both locations). These doomed transactions may enter an infinite loop. To ensure that doomed transactions are eventually aborted, we use a watchdog timer to trigger a validation pass. This satisfies *opacity* [26] with minimum overhead for GPUs.

4.3 Version Management

KILO TM manages global memory accesses in hardware and uses software for version management of registers and local memory space. Section 3.1.2 discussed how blindly checkpointing each transaction is too expensive on GPUs. We observed that the original values in many registers are rarely used when a transaction restarts, and do not need to be restored. A compiler could determine which registers are both read and written within a transaction and insert code to checkpoint and restore them before/after a transaction. We observed that transactions in the BH benchmark require restoring *two* registers on average. Other benchmarks do not require any register restoration upon transaction aborts. Hence, we do not model the register checkpoint overhead as we believe it to be minor compared to validation and commit overheads in our workloads.

Accesses to global memory are buffered in the read/write-log in local memory. A small bloom filter can be used to detect whether a transaction is reading a value in its write-set. A hit in the filter will trigger the transaction log unit to walk the write-log. Since the member set of the filter is constrained to only the memory accesses of a single transaction, a small filter should produce reasonably few false positives. In our evaluations, this detection is perfect.

4.4 Transaction Log Storage and Transfer

The read- and write-logs of transactions in KILO TM are stored as linear buffers in local memory located in off-chip DRAM, cached in the per-core L1 data cache, and mapped to physical addresses such that consecutive 32-bit words are accessed by consecutive scalar threads in a warp. GPU applications can specify the maximum size of local memory to avoid overflow.

When a warp accesses global memory in a transaction, a new entry is appended to the read/write-log for all threads in the warp. Entries for the inactive threads are marked with a special address to void the entry. This organization allows the log accesses to be coalesced. If only part of a warp needs to walk the write-log for

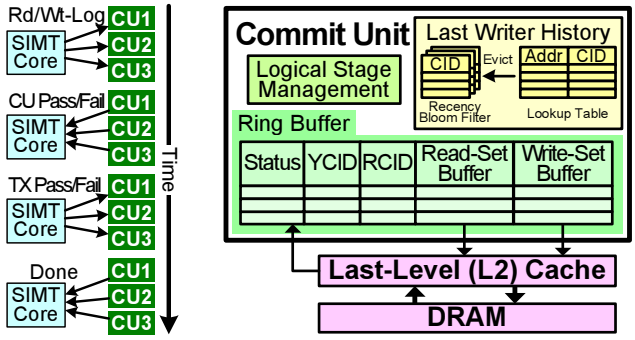


Figure 4: Commit Unit. (a) Communication flow with a SIMT core. (b) Overview of a commit unit.

data, the entire warp will wait for the walk to finish before proceeding to the next instruction. When threads in a warp are ready to validate their transactions (before commit), the transaction log unit walks the read- and write-logs of each thread and sends the address-value pairs to the commit unit in the corresponding memory partition. The individual entries sent to the same memory partition are grouped into a single larger packet to reduce interconnection traffic.

This transaction log design addresses the fact that per-core caches in contemporary GPUs are shared by 100s of threads. GPUs employ a flexible register allocation scheme that balances the number of registers per warp against the number of concurrent warps to avoid spilling registers. Hence, memory reads rarely access data written by the same transaction, reducing the penalty of storing the write-log as a linear buffer.

4.5 Distributed Validation/Commit Pipeline

A naive implementation of value-based conflict detection serializes transaction commits. Memory updates from a transaction (its write-set) are invisible to others until the transaction commits. Two conflicting transactions validating concurrently will observe no changes to their read-sets, and will subsequently update global memory with their contradicting write-sets. While serializing all transaction commits prevents this potential data race, it also prevents non-conflicting transactions from committing in parallel [9].

To enable parallel commits, prior STMs with value-based conflict detection [19, 42] use a set of versioned locks, each serializing commit to a memory region. Each transaction checks/acquires the locks of all the memory regions that require protection during validation and commit. Acquiring locks imposes significant overhead.

KILO TM increases commit parallelism by using a set of *commit units* that quickly detect conflicts among a limited set of transactions. GPU memory subsystems are deeply pipelined to support a large number of in-flight accesses to maximize throughput. The commit units leverage this capability. In each commit unit, a subset of transactions are speculatively validated in parallel. This validation only detects conflicts with the already committed transactions. Later, a *hazard detection* mechanism is applied to conservatively detect all potential conflicts. Any hazard is resolved by deferring one of the conflicting transactions and *revalidating* its read-set after the other transaction has updated global memory. Revalidation serializes the validation/commit process among transactions when necessary. This mechanism guarantees forward progress by giving the deferred transaction a second chance to validate and commit in case the earlier transaction failed.

Each memory partition has a commit unit (shown in Figure 4b) that handles validations and commits of TM accesses to that memory partition. Before a transaction starts the validation/commit process, it acquires a commit ID (CID) from a centralized ID vendor (similar to Scalable TCC [16]). This commit ID is associated with

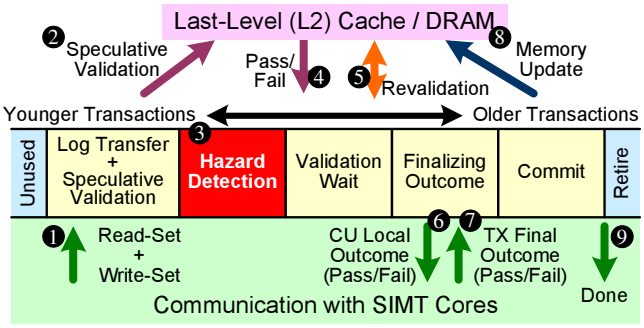


Figure 5: Logical stage organization and communication traffic of the bounded ring buffer stored inside commit unit.

a logical entry in the commit unit at every memory partition, and dictates the relative commit order of this transaction (so that the conflict/hazard resolution is unanimous among all commit units). Each commit unit has a ring buffer of commit entries [49] organized in the logical stages shown in Figure 5. Each entry tracks the state of a committing transaction in this memory partition. The Status field in each commit unit ring buffer entry in Figure 4b indicates the current status of the transaction. The YCID and RCID fields are used for hazard detection. The RCID of a transaction is a pointer to the oldest committing transaction when this transaction started speculative validation. Transactions that committed before this transaction started speculative validation do not trigger a hazard with this transaction. The YCID of a transaction points to the youngest conflicting transaction detected to have a hazard with this transaction. The transaction needs to wait for the conflicting transaction to retire before it can start revalidation. The Read-Set and Write-Set Buffers consist of bounded linear buffers that store, for value comparison, the exact address-value pairs of each transactional access to this memory partition.

The following is an overview of the validation/commit process of a transaction at different logical stages (Figure 5):

Log Transfer + Speculative Validation. The transaction transfers its read- and write-logs to an allocated entry in the commit unit (❶). The incoming read-set is speculatively validated against the current values in global memory (accessing L2 cache/DRAM ❷).

Hazard Detection. Once the read- and write-logs have been transferred to the commit unit, the read-set of the transaction is checked against the *Last Writer History* unit (LWH) for *hazard detection* (❸), detecting conflicts between the transaction and all committing transactions in the later stages. Existence of a hazard indicates that the speculative validation may have accessed stale data in global memory that will be updated before the transaction commits. The hazard is resolved in the Validation Wait stage.

Validation Wait. Each transaction waits for the speculative validation to complete before advancing to later stages (❹). Transactions with hazards will wait until all conflicting transactions have retired to revalidate their read-set with the updated global memory (❺).

Finalizing Outcome. This stage finalizes the outcome of each transaction by replying with the local outcome (pass/fail) of the transaction to the core (❻). After a transaction has received replies from all commit units, it will broadcast the final outcome (pass/fail) to all commit units (❼). Each commit unit entry waits for its final outcome before proceeding to the next stage.

Commit. Each passed transaction updates the global memory at this stage (❽). Failed transactions are skipped.

Retire. The commit unit entry associated with each transaction is deallocated, releasing storage for future transactions. The core is informed so that the thread running the transaction can proceed to the next instruction, or restart the failed transaction (❾).

4.5.1 *Commit Unit Resource Allocation*

When a warp executes `tx_commit`, the transaction log unit acquires credits from a per-core credit pool of commit unit entries before acquiring contiguous commit IDs and proceeding with the commit. Insufficient credits prevent the warp from acquiring the commit IDs until the credits are returned when the validation/commit operation of another warp completes. In this paper, we assume that the commit units always have enough entries to support all in-flight transactions. Section 6.4.2 measures the resources required.

4.5.2 *Hazard Detection, Last Writer History Update*

At the Hazard Detection stage, each transaction checks the integrity of its speculative validation via the *Last Writer History* unit (LWH) in Figure 4b. This unit has an approximate but conservative representation of the write-sets of all older transactions at the later stages that have not yet retired. The LWH unit identifies the youngest conflicting transaction (returns its commit ID) in the later stages that may modify the read-set of the transaction at the hazard detection stage. If this conflicting transaction retired before the current transaction started validating (its CID<RCID of the current transaction), no hazard remains. Otherwise, a hazard is detected. The hazard is resolved in the Validation Wait stage by waiting for this conflicting transaction (now tracked by YCID) to retire, and then revalidating the transaction with the updated memory. After detection, the current transaction updates the LWH unit with its write-set. This mechanism leverages the same intuition described in Section 4.2. The LWH unit can approximately maintain the latest pending writer to each memory location, as a slightly younger false writer only slightly lengthens the wait at Validation Wait stage.

The LWH unit has an address-indexed set-associative lookup table and a *recency bloom filter*. The two structures, in combination, conservatively track the CID of the *youngest* transaction in later stages that may write to a given memory location. The lookup table stores the exact write-sets from recent transactions, whereas the bloom filter stores the approximate write-sets from distant transactions. As write-sets from newer transactions are deposited into the lookup table, entries are updated (replacing the CID if addresses match), or evicted into the bloom filter to free up storage for different addresses. The recency bloom filter has multiple sub-arrays of buckets (each bucket storing a CID) indexed by a hash of the given memory address. Each evicted entry updates a CID bucket in each sub-array according to the hashed written memory address. Due to address aliasing in each sub-array, an older CID writing to an address may be replaced by a younger CID writing to a different address. When the bloom filter is queried with an address, one CID is retrieved from each sub-array. The *oldest* retrieved CID is returned as it is least likely to have been aliased by a younger writer. This oldest CID can also be aliased, causing the LHW unit to report a false writer. The write-set of a retiring transaction is implicitly removed from the LWH unit as its CID can no longer trigger a hazard.

4.5.3 *Unbounded Transactions*

If the commit entry's read-set buffer overflows, the commit unit will continue to speculatively validate the address-value pair of the incoming read-set, but will stop populating the read-set buffer. The commit unit will ask the transaction to resend its read-set from the SIMT core during hazard detection and revalidation. Similarly, if the commit entry's write-set buffer overflows, the commit unit will ask the transaction to resend the write-set during LWH update after hazard detection and memory update at the Commit stage.

4.6 Concurrency Control

While KILO TM can support thousands of concurrent transactions, limiting the number of concurrent transactions can improve

the performance of high-contention applications (with transactions that are likely to abort), and lowers the resource requirement for KILO TM. To limit the number of concurrent transactions within a SIMT core, we use a counter to track the number of warps currently in transactions. We will explore adaptive mechanisms (e.g., [57, 7]) that react to the dynamic contention in applications in the future.

5. METHODOLOGY

We model our proposed hardware changes by extending GPGPU-Sim 3.0 [6]. We evaluate performance of various hardware configurations on the benchmarks listed in Table 2. We add transactions with empty functions `tx_begin()` and `tx_commit()` that are recognized by the simulator as transaction boundaries. The following CUDA/OpenCL applications are used in our evaluations.

Hash Table (HT) is a microbenchmark in which each thread inserts an element into a chained hash table. Each slot in the hash table is a linked list of key-value pairs. We use two table sizes to create high contention (HT-H with 8k entries) and low contention (HT-L with 80k entries) workloads.

Bank Account (ATM) is a microbenchmark with ∼16k concurrent threads accessing an array of structs that represents 1M bank accounts. Each transaction transfers money between two accounts.

Cloth Physics (CL) is based on "RopaDemo", which simulates the cloth physics for a T-shirt [11]. Performance is limited by the Distance Solver kernel, which implements a spring-mass system using a set of constraints between cloth particles. To forbid two constraints concurrently modifying the same particle, the original demo processes these constraints sequentially in octets (i.e., 8 at a time). We modified this kernel to process all ∼4k distance constraints of each T-shirt in parallel transactions.

Barnes Hut (BH) is based on the tree-based n-Body algorithm implemented by Burtscher *et al.* [12] with 30000 bodies. We focus on the iterative tree-building kernel using lightweight locks, which we modified to use transactions. Each thread in this kernel appends a body into the octree, and inserts any branch node required to isolate its body in a unique leaf node. Each level of traversal down the tree and the node insertions are protected by separate transactions.

CudaCuts (CC) applies a maxflow/mincut algorithm to segmentation of a 200×150 pixel image [53]. It consists of Push kernels that use atomic operations to push excessive flow from a node to its neighbours, and Relabel kernels that change the height of a node when excessive flow cannot be pushed. We grouped consecutive atomic operations in the Push kernels into transactions.

Data Mining (AP) is based on Apriori in the RMS-TM benchmark suite [2, 30]. We evaluate the `apriori_gen()` function, which was modified [43] to use CUDA, with each thread processing a unique record. As in the CPU TM version, transactions are used to protect candidate insertion and support value counting.

Our modified GPGPU-Sim is configured to model a GPU similar to NVIDIA Quadro FX5800, extended with L1 data caches and a L2 unified cache similar to NVIDIA Fermi [40]. We validated GPGPU-Sim 3.0 with the NVIDIA Quadro FX5800 configuration (no cache extensions and using PTX instead of SASS) against the hardware GPU and observed an IPC correlation of ∼0.93 for a subset of the CUDA SDK benchmarks. GPGPU-Sim incorporates a configurable interconnection network simulator [20]. Traffic in each direction between the SIMT cores and the memory partitions are serviced by two separate crossbars. The crossbars can transfer a 32-byte flit per interconnect cycle to/from each memory partition (∼160GB/s per direction). Each flit takes 5 cycles to traverse the crossbar. The 30 SIMT cores are grouped in 10 clusters. Cores in a cluster share a common port to each crossbar (concentration of three). The memory partition has an out-of-order memory access scheduler. We model detailed GDDR3 timing. Every memory access sent to L2 cache/DRAM has a minimum pipeline latency of 460 cycles (in compute core clocks) to match that observed by microbenchmarks of NVIDIA Quadro FX5800 [55]. The actual latencies of individual accesses can be higher due to delays from memory access scheduling and queuing as DRAM bandwidth saturates. We have an optimistic performance model for atomic operations (used in fine-grained locking). Atomic compare-and-swap operations on GPGPU-Sim have ∼4× higher throughput than on NVIDIA Fermi GPU, while other types of atomic operations on GPGPU-Sim perform roughly the same as Fermi. Table 3 lists the other major configuration parameters.

Table 3: GPGPU-Sim Configuration

# SIMT Cores	30 (10 clusters of 3)
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Branch Divergence Method	PDOM [23]
Warp Scheduling Policy	Loose Round Robin
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	5KB, 32B line, 20-way assoc.
L1 Data Cache / Core	48KB, 128B line, 6-way assoc. (transactional and local memory access only)
L2 Unified Cache	64KB/Memory Partition, 128B line, 8-way assoc.
Interconnect Topology	1 Crossbar/Direction (SIMT Core Concentration=3)
Interconnect BW	32 (Bytes/Cycle) (160GB/dir.)
Interconnect Latency	5 Cycle (Interconnect Clock)
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
# Memory Partitions	8
DRAM Req. Queue Capacity	32
Memory Controller	Out-of-Order (FR-FCFS)
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8 t_{CDLR}=6 t_{WR}=11
Memory Channel BW	8 (Bytes/Cycle)
Min. L2/DRAM Latency	460 Cycle (Compute Core Clock)
KILO TM	
Commit Unit Clock	650 MHz
Validation/Commit BW	1 Word/Cycle/Memory Partition
#Concurrent TX	2 Warps/Core (1920 threads)
Last Writer History Unit Size	5kB (See Section 6.3.2)

We model all interconnection network traffic between the SIMT cores and the commit units. Packets from the transaction log unit are sized according to the payload within the packet, and they contend for the same interconnection port with packets for normal memory accesses. Each short commit protocol message occupies a single flit. Packets containing multiple read/write-log entries (see Section 4.4) may occupy multiple flits, taking multiple cycles to transfer. In our evaluations, KILO TM validates and commits each transaction as directed by the timing simulation. In our simulation, timing of committing transactions affects functional behaviour of the application, and hence any undetected data-race would likely lead to an application error, which we verify does not occur.

6. EXPERIMENTAL RESULTS

6.1 Performance

In this section, we compare the performance of KILO TM against the ideal TM system (Ideal TM) and fine-grained locking (FG Lock) described in Section 3. Figure 6 shows the execution time of each application with KILO TM and fine-gained locking normalized to the execution time of Ideal TM. In our evaluations, KILO TM uses commit units with unlimited capacity.

With unlimited transaction concurrency (Inf. TransWarp), KILO TM is on average 4.1× slower than Ideal TM. HT, CL, and BH are affected the most. These applications have many concurrent transactions with high contention (Figure 7). Although BH's overall

Table 2: Benchmarks. #Inst obtained from Ideal TM version. Other metrics refer to KILO TM with unlimited concurrency.

Name	Abbr.	#Inst	Blk Size	Grid Size	#Blk/ SIMT Core	#Committed TX	#Inst per TX (Avg)	#Aborts per TX Avg	#Aborts per TX Max	Read-Set (#Words) Avg	Read-Set (#Words) Max	Write-Set (#Words) Avg	Write-Set (#Words) Max	Max # Concurrent TX (KILO TM)
Hash Table (CUDA)	HT-H	632k	192	120	4	23040	26	1.39	2	1.0	1	4.0	4	23040
	HT-L	501k	192	120	4	23040	26	0.14	2	1.0	1	4.0	4	23040
Bank Account (CUDA)	ATM	4.1M	192	120	3	122880	8	0.03	3	3.0	3	2.0	2	16131
Cloth Physics [11] (OpenCL)	CL	6.8M	512	118	1	60200	53	1.06	8	11.2	12	4.8	8	22816
Barnes Hut [12] (CUDA)	BH	15M	288	60	1	264106	48	0.15	14	4.3	40	0.82	14	8640
CudaCuts [53] (CUDA)	CC	104M	256	133	1	114677	21	0.004	3	1.4	4	1.4	4	735
Data Mining [2, 30] (CUDA)	AP	39M	64	112	4	4550	89	0.32	6	15.7	174	6.2	109	192

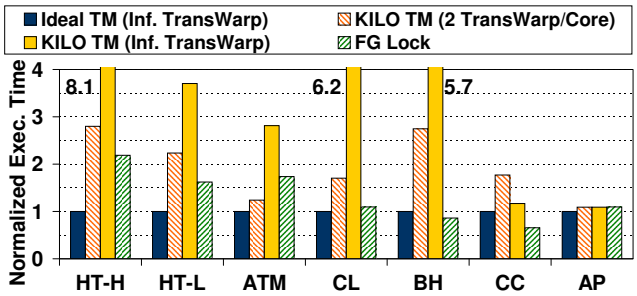


Figure 6: Execution Time. Lower is better.

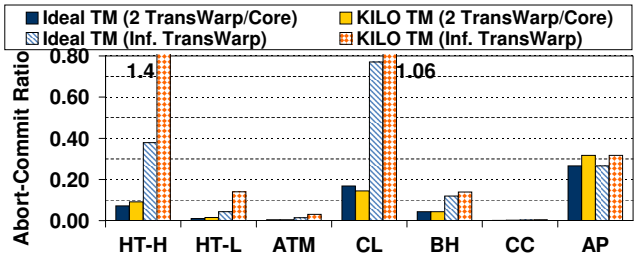


Figure 7: Abort/Commit Ratio. Lower is better. Ratio = 1 if on average transactions abort once.

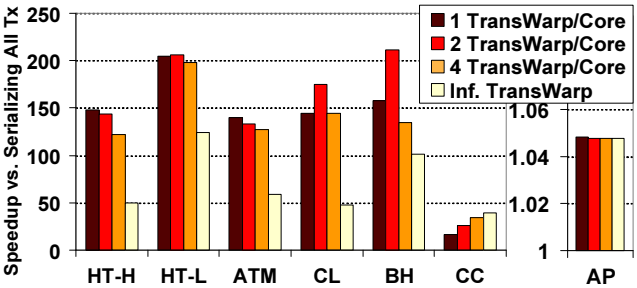


Figure 8: Performance scaling with increasing number of concurrent transactions with KILO TM. Higher is better.

abort-commit ratio is relatively low, it starts with a high-contention period when all transactions are competing to insert nodes near the root of the octree. When conflicting transactions attempt to commit concurrently, the commit unit defers revalidating one transaction. This reduces overall performance. Notice that AP has relatively few concurrent transactions, so its high abort-commit ratio has little impact on performance (Table 2).

Limiting each core to two transaction warps (2 TransWarp/Core in Figure 6, 1920 threads globally) reduces contention in HT-H, CL, and BH and improves their performance with KILO TM by 2-3×. ATM speeds up by 2.3× from improved hazard detection accuracy. The performance of HT-L improves by 66%, while AP is unaffected. CC's performance drops by 34% because of this limit. In CC, warps are typically diverged before entering transactions. CC would not be penalized with thread-level concurrency control. Overall, KILO TM performs significantly better with this concurrency limit, capturing 52% of Ideal TM and 59% of fine-grained locking performance.

Concurrency Control. Figure 8 compares the performance of KILO TM under different concurrency limits versus serializing execution of all transactions. HT-L, ATM, CL and BH achieve the best performance with transaction execution limited to two warps per core (2 TransWarp/Core), while HT-H performs best with transaction execution limited to one warp per core (1 TransWarp/Core). CC prefers unlimited transaction concurrency. AP is insensitive to the limit. Overall, KILO TM performs best with transaction concurrency limited to two warps per core, achieving on average 128× speedup over serially executing each transaction.

Effects on Abort-Commit Ratio. Figure 7 compares the abort-commit ratios between KILO TM and the ideal TM system. KILO TM and Ideal TM show similar abort-commit ratios with transaction concurrency limited to two warps per core. With unlimited transaction concurrency, contention at the commit unit defers memory updates from older transactions that would have been made visible much earlier with Ideal TM. Younger transactions that were originally reading the updated values in Ideal TM now conflict with the older uncommitted transactions.

6.2 Execution Time Breakdown

To provide further insight, Figure 9 shows a breakdown of the cumulative per-hardware thread cycles, scaled by the overall execution time of each application. At each cycle, a thread can be in a warp stalled by *Concurrency control* (TC), be in a warp committing its transactions (TO), have passed commit and be *Waiting* for other threads in its warp to pass (TW), be executing an eventually *Aborted* (TA) or committed/*Useful* (TU) transaction, be acquiring a lock or performing an *Atomic operation* (AT), be waiting at a *Barrier* (BA), or be performing non-atomic non-transactional work (NL). We compare the thread-state distributions between the fine-grained locking versions of the benchmarks (FGL), and the transactional versions running on Ideal TM (IDEAL), KILO TM with transaction concurrency limited to two warps per core (KL), and KILO TM with unlimited transaction concurrency (KL-UC).

We observe the overheads of lock acquisition (AT) in the lock-based versions to be proportional to the inherent contention in their transactional versions. Transactional HT-H and CL have the largest abort-commit ratios in Figure 7 and their lock-based counterparts have the greatest locking overheads in Figure 9. HT-L, ATM and CC have lowest abort-commit ratios and the smallest locking overheads. Lock-based BH has a significant locking overhead because of the initial high-contention period, as explained in Section 6.1. Lock-based AP shows insignificant locking overhead, despite a high abort-commit ratio in its transactional version, due to limited parallelism in its implementation. For the lock-based benchmarks, the NL cycles include the execution of the critical sections and are therefore greater than in the transactional versions. Detailed analysis (not shown) indicates that lock-based benchmarks suffer from increased branch divergence, further increasing their NL cycles.

Threads running on KILO TM with unlimited transaction concurrency spend much of their time waiting to be committed (TO). This overhead is significantly reduced by limiting transaction exe-

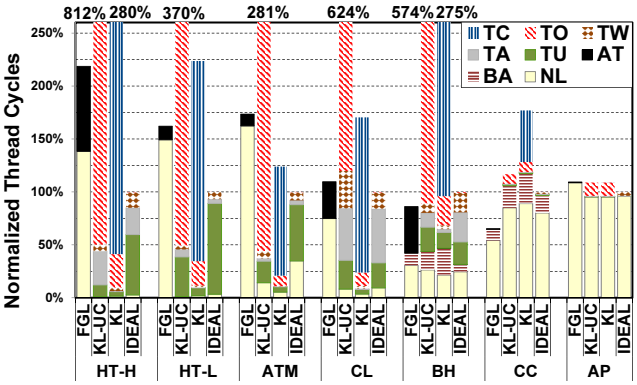


Figure 9: Breakdown of thread execution cycles. Scaled by the overall execution time, normalized to IDEAL TM.

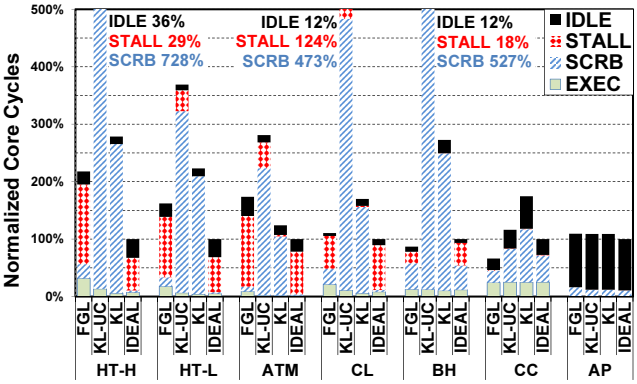


Figure 10: Breakdown of core execution cycles.

cution to two warps per core in exchange for long waits in concurrency control (TC). For most benchmarks this provides an overall gain in performance. CC's performance on KILO TM, however, degrades with concurrency control. This is because CC's originally low commit overhead remains unchanged with reduced concurrency, and because CC benefits from increased transaction concurrency as indicated by its scaling performance in Figure 8. Figure 10 shows the cumulative execution cycle breakdown of each core. At each cycle, a SIMT core may issue a warp (EXEC), be stalled by downstream pipeline stages (STALL), have all warps blocked by the scoreboard due to data hazards, concurrency control, pending commits or any combination thereof (SCRB), or not have any warps ready to issue in the instruction buffer (IDLE). This figure shows that limiting concurrency in KILO TM reduces stalling and waiting at the scoreboard. Stalling is reduced as a result of fewer concurrent transactional memory accesses, while shorter and fewer commits reduce the amount of time spent waiting at the scoreboard.

The amount of time spent on transactional work, indicated by TU and TA in Figure 9, is lower on KL than on KL-UC and IDEAL. This is also due to the reduction in STALL cycles in Figure 10 for KL. Reduced stalling leads to faster transaction completion. BH saw only a small decrease in transaction time when concurrency was reduced. This is because BH contains inherent and limiting memory dependencies that are visible in Figure 10 as SCRB cycles on IDEAL TM. Similar to TU and TA, TW also decreases with reduced concurrency as passed transactions spend less time waiting for failed transactions in their warp to re-execute and commit.

In Figure 9, HT-H, HT-L and ATM spend less time doing useful transactional work (TU) on KL-UC than on IDEAL, even though both have unlimited concurrency. This is because KILO TM caches global memory writes in write-logs stored in the L1 data cache. HT-H benefits most from this buffering during transaction execution as its transactions are dominated by writes (See Table 2). HT-L and ATM's lower data locality negates some of the benefit of write buffering. CL and BH are dominated by reads and gain little benefit from write buffering. The memory write overhead of write buffering is eventually incurred during transaction commit (TO).

CC and AP both suffer from load imbalance as indicated in Figure 10 by the significant portion of IDLE cycles - the portion of the time when the cores run out of warps to execute. The inter-thread load imbalance suffered by CC is exacerbated by transactional overheads. AP suffers from inter-core load imbalance. AP spends most its execution in non-transactional work, but the overhead of KILO TM still impacts performance because of the time involved in transferring logs for the large transactions. AP spends 90% of its core cycles in IDLE. This behaviour contributes to the low absolute performance of AP. We created the CUDA version of AP from its CPU TM version without changing much of the algorithm and data structures. An improved version may redesign the algorithm to spread the workload across more threads.

Overall, even with a significant portion of time spent on executing aborted transactions, the Ideal TM system performs comparably to fine-grained locking. This indicates that the performance penalties of KILO TM may be reduced with future refinements.

6.3 Sensitivity Analysis

6.3.1 *L2 Cache Miss from Validation Access*

We observe that >90% of validation accesses for KILO TM hit in the L2 cache for all benchmarks with transactional execution limited to two warps per core. This also applies to most benchmarks with unlimited concurrent transactions, but for HT-L, ATM and CL, the cache hit rate for validation access is lower (70% for HT-L, 46% for ATM and 62% for CL). These extra accesses are easily handled by the GPU memory subsystem. In a sensitivity study with idealized validation accesses that always hit in the L2 cache, performance of ATM and CL improves only by 11% and 17%, respectively. Other benchmarks (including HT-L) are insensitive to this change. In this study, KILO TM employs LWH units that detect hazards perfectly. About 50% of the validation-induced L2 cache accesses in CL are pending hits. In ATM, the extra L2 cache misses improve the row-hit rate in the open-row, out-of-order DRAM controller, increasing the bandwidth efficiency by 5%. The improved efficiency partly compensates the penalty from validation-induced DRAM accesses. In HT-L, these L2 cache misses increase the DRAM bandwidth utilization by 5% and do not impact performance. This ability to handle extra memory accesses in GPUs shows why value-based conflict detection is a viable solution for supporting TM on GPUs.

6.3.2 *Hazard Detection Sensitivity*

We explored the performance of KILO TM with different hazard detection mechanisms. We compared two versions of the LWH mechanism described in Section 4.5.2 and an additional mechanism based on a bloom filter array. The first 5kB LWH consists of a 512-entry, 4-way set-associative lookup table (3kB) and a 1024-bucket bloom filter (2kB) split into 4 separate sub-arrays, each array indexed by a unique H_3 hash function (similar to the parallel bloom signature described by Sanchez *et al.* and Ceze *et al.* [45, 15]). The second 512B LWH configuration consists of a 64-entry lookup table and a 64-bucket bloom filter. A second detection mechanism, bloom filter array (BF Array), encodes the read-set and write-set of each transaction into two 512-bit signatures in the commit unit. Each signature consists of 4 sub-signatures with each indexed by a unique H_3 hash function. Incoming read/write accesses check for conflicts against all signatures in the commit unit in parallel.

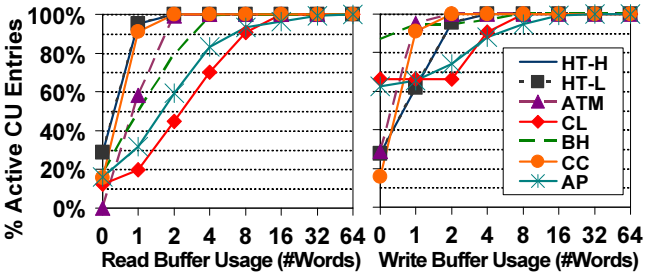


Figure 11: Buffer usage in active commit unit (CU) entries.

KILO TM with 5kB LWH unit performs almost identically to perfect hazard detection. The 512B LWH reduces the storage by 10× but increases execution time by only 36% on average. Despite taking 24× more storage than the LWH unit (120kB vs. 5kB per commit unit), BF Array slows down KILO TM by up to 7.8× (4.3× on average). The lookup table plus bloom filter design in a LWH unit dedicates extra resources to ensure that an unnecessarily revalidating transaction and its false writers are far apart in the commit unit pipeline, minimizing the stalling at the Validation Wait stage.

6.4 Implementation Complexity of KILO TM

In each SIMT core, KILO TM implementation involves extending the SIMT stack to support transactions, employing concurrency control, and adding a transaction log unit. Even though each transaction log unit manages 1000s of transactions, most of the bookkeeping is amortized across the warp. For example, threads in the same warp have the same read-log and write-log sizes, and they always have consecutive commit IDs. The L1 data cache stores the read-/write-logs. Evicted entries are written back to L2/DRAM. We believe the area overhead of a transaction log unit is negligible.

The area overhead of a commit unit consists of the storage required for the LWH unit, the entries in the ring buffer, and the read- and write-set storage buffers for each entry. Section 6.3.2 showed that a 5kB LWH unit is sufficient. Each commit unit ring buffer entry occupies 10 bytes for the status, RCID and YCID fields, and pointers to a shared pool of the read- and write-set buffers. The area required for the read- and write-set buffers is a product of the size of each buffer and the number of buffers present. Section 6.4.1 examines how large each fixed-size buffer should be to limit buffer overflow. Section 6.4.2 examines how many of these fixed-size buffers are required concurrently.

6.4.1 *Read-Set/Write-Set Buffer Capacity*

Figure 11 shows the cumulative distribution of the read- and write-set buffer usage for the active ring buffer entries in the commit units. The distributions show that an 8-word (64 Bytes) read-set buffer and an 8-word write-set buffer can serve >90% of the commit unit ring buffer entries. If the read-set or the write-set buffer overflows (a rare event), the penalty involves resending the read- and write-log. We leave performance evaluation with finite-sized read- and write-set buffers as future work.

6.4.2 *Commit Unit Capacity*

As not all commit unit ring buffer entries will need read- and write-set buffers, dynamically allocating these buffers from a shared pool reduces the area overhead of buffers. Figure 12 shows the average and maximum number of read- and write-set buffers required for 4 different buffer allocation schemes. The *All* and *Accessed* allocation schemes allocate fixed-size read- and write- buffers for a commit unit ring buffer entry when it is created, and deallocate the buffers when the entry retires. *All* allocates the two buffers for all active entries in the ring buffer, while *Accessed* only allocates buffers for the entries whose transaction has accessed this memory partition. The number of ring buffer entries in the *All* and *Accessed*

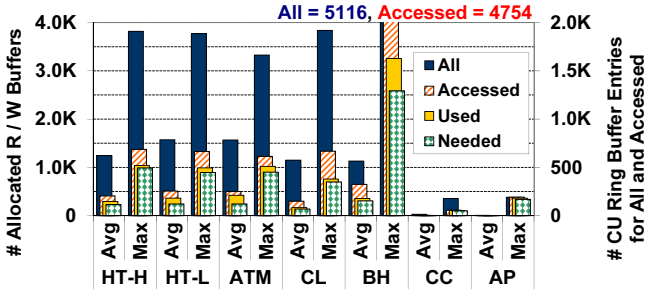


Figure 12: Number of in-flight, allocated read and write buffers for different buffer allocation schemes. All ⊇ Accessed ⊇ Used ⊇ Needed. Y-axis on right shows number of commit unit ring buffer entries with buffers allocated for All and Accessed.

schemes is given by the right Y-axis in Figure 12. The *Used* allocation scheme improves upon *Accessed* by allowing a single fixed-size read- or write-set buffer to be allocated if the transaction has an empty write-set or read-set buffer for this memory partition, respectively. *Needed* further improves upon *Used* by allowing buffers to be deallocated before the ring buffer entry retires. For example, buffers are deallocated as soon as a transaction fails. In Figure 12, the number of concurrent transactions is limited to two warps per core (1920 in total) via concurrency control, and there is no capacity limit on the number of ring buffer entries.

HT-H, HT-L, ATM, CL and BH use ∼700 ring buffer entries on average (*All*). BH's maximum number of entries exceeds the concurrent transaction limit because it has many read-only transactions. The SIMT core considers a read-only transaction to be done when it receives the local outcome reply from commit units, allowing the next waiting transaction to proceed before the corresponding commit unit entries are retired. We observed that the number of in-flight entries exceeded the concurrent transaction limit of 1920 for only <1% of the execution time for BH. CC and AP have significantly fewer concurrent transactions and therefore require fewer entries. A commit unit with 1920 entries with two 64B buffers for all entries (*All*) would require 240kB for read- and write-set storage, and 19kB for the ring buffer storage (10B per ring buffer entry). The *Accessed*, *Used* and *Needed* optimizations reduce the storage requirement for buffers. To serve most of the *Needed* buffer usage, ∼500 buffers per commit unit (32kB per unit, 256kB for the whole GPU) are enough. The rare worst case can be handled by deferring validation/commit via the credit-based allocation mechanism described in Section 4.5.1. We leave the performance evaluation of this allocation mechanism as future work.

6.4.3 *Area Estimation*

We assume that both the 32kB read- and write-set buffer pool and the 19kB ring buffer have 4 banks of SRAM arrays. The 5kB last writer history unit (Section 6.3.2) consists of a 3kB lookup table and a 2kB bloom filter, each an SRAM array. Using CACTI 5.3 [47], we estimate the area of each commit unit (the aggregate area of these arrays) to be $0.40mm^2$ in a 40nm technology. NVIDIA Fermi GPU features 6 memory partitions [40], so implementing the commit units on Fermi architecture requires an area of $2.41mm^2$. This is just 0.5% of Fermi's $520mm^2$ die area.

7. RELATED WORK

Cache Coherence Protocol Based HTMs. Many existing hardware and hybrid TM systems focus on leveraging the sharer/modifier information maintained by cache coherency hardware for conflict detection and using thread-private caches for version management. Some of the HTMs propose to extend the cache coherence protocol with new coherence states for conflict detection and ver-

sion management [16, 52, 17, 21]. Other HTMs monitor the cache coherency traffic (of an existing protocol) for conflict detection, using per-transaction metadata stored in signatures [36, 56, 13], or extra bits added to the cache line [38, 10, 44].

Cache Coherence Protocol for ManyCore Architecture. Several cache coherence directories scalable to 1000 cores have been proposed [29, 58, 22]. Tarjan *et al.* propose a sharing tracker that tracks cache lines in the private cache of each GPU core (a SIMT core) imprecisely [51]. It only maintains a subset of sharers for a cache line, insufficient for correct conflict detection.

Signature-Based Conflict Detection. In BulkTM [15], a committing transaction broadcasts its write-set in a bloom filter based signature, which is compared for conflicts against the signature and the L1 cache tags at each recipient transaction. SigTM [36] and LogTM-SE [56] eagerly detect conflicts with signatures by monitoring the address of each incoming cache coherence request. A software conflict resolution handler is invoked when the address hits the signatures. Other HTMs (TMACC [13], FlexTM [48]) use signatures to handle unbounded transactions.

Software transactional memory (STM) systems also use signatures for conflict detection. RingSTM [49] uses a ring of commit records that hold the write-signatures of recently committed transactions. Before each transactional load, the transaction compares its read-signature against the write-signatures of newly committed transactions added to the commit records since the transaction's last load. A match indicates a new conflict and the transaction is aborted. In InvalSTM [25], a committing transaction compares its write-signature against the read- and write-signatures of other running transactions. A contention manager is invoked upon a match.

To handle unbounded transactions in hazard detection, each commit unit in KILO TM uses a last writer history unit that tracks the write-sets of all committing transactions via a novel recency bloom filter. This bloom filter has commit IDs in its buckets and is implicitly cleared as the associated transactions retire. The recency bloom filter has some similarities to a time-out bloom filter [32]. The time-out bloom filter was proposed for use in network packet sampling, where the key result is the existence of a similar packet occurring within a given window. In contrast, the recency bloom filter provides greater detail. In the presence of a conflict it returns the identity of the conflict rather than simply a pass/fail result. The identity it returns is that of the most recent item inserted into the bloom filter that can possibly conflict.

KILO TM and RingSTM [49] are conceptually similar in that both use a ring to order transaction commits, and detect conflicts between read-set of a transaction and write-sets of committed transactions. KILO TM uses value-based conflict detection to eliminate storage for committed write-sets, and uses a LWH unit to detect hazards (conflicts) among committing transactions. Each transaction in KILO TM stores its read-set in exact address-value pairs. It also features multiple commit units, each with a separate ring, and maintains consistent commit order via a protocol similar to Scalable TCC [16]. KILO TM enforces opacity via a watchdog timer, removing the need to validate before every transactional read.

Value-Based Conflict Detection. JudoSTM [42] allows parallel commits with a set of versioned locks each guarding a memory region. Each transaction checks/acquires the locks of all the regions that requires protection during validation and commit. NORec [19] uses a single global versioned lock to offer fast checking with a small number of concurrent threads. DPTM [50] is a cache coherence protocol based HTM. It uses value-based conflict detection to mitigate the false conflicts that are caused by false sharing of cache lines between transactions. KILO TM uses value-based conflict detection, but the motivation is to eliminate global metadata for conflict detection with 1000s of concurrent transactions. It uses special hardware to allow non-conflicting transactions to validate and commit in parallel.

Transaction Scheduling. In this paper, we describe an extension to the GPU hardware thread scheduler to control the number of concurrent transactions. It is effective for high-contention workloads. There exist other transaction schedulers that dynamically adjust concurrency according to predicted contention [57, 7] or the detected transaction footprint [8]. We leave the exploration of such adaptive transaction schedulers on GPUs as future work.

STM for GPUs. Cederman *et al.* have proposed two versioned lock-based STMs (blocking/non-blocking) on GPUs [14]. In their evaluations, while STM-based data structures scale well, they perform $\sim$10$\times$ slower than lock-free data structures. KILO TM, our GPU hardware extension to support TM, uses value-based conflict detection to remove the storage overhead for versioned locks. With dedicated hardware to increase commit parallelism and limit concurrency, KILO TM performs much closer to fine-grained locking, which should perform on-par with lock-free data structures.

8. CONCLUSION

This paper proposes the use of transactional memory for GPU computing. Transactions can simplify parallel programming by making it easier to reason about parallelism. This becomes more important as the number of threads increases and as more software is ported to take advantage of GPUs' better peak performance and power efficiency. Compared to lock-based programming, TM simplifies the porting/creation of applications that require data synchronization on GPUs. Specifically, TM is a better fit to the current GPU programming models. The isolation property of TM is similar to how GPU threads are exposed in the programming model. The application specifies as many transactions as it can, the TM system attempts to execute them in parallel, but transactions can run in isolation. The GPU hardware can be designed to automatically handle interactions between data synchronization and the SIMT stack, solving an obstacle that prevented fine-grained data synchronization from being widely used in GPU applications. Furthermore, TM frees the programmer from deadlock concerns as they rework the algorithms and data structures to optimize the performance of their application.

This paper proposes KILO TM, a novel HTM system scalable to 1000s of concurrent transactions. It uses value-based conflict detection to offer weak isolation, avoid the need for coherence, reduce metadata overheads, support unbounded transactions, and detect conflicts at the granularity of individual words. We describe a scalable parallel commit protocol and the changes to a SIMT hardware organization required to support transactions. KILO TM uses a novel speculative validation mechanism to improve the validation and commit parallelism for non-conflicting transactions. By design, it favors applications with low contention transactions. We have evaluated KILO TM with a set of TM-enhanced GPU applications with various degrees of exposed parallelism (the granularity of the decomposition of work into threads) and contention. We find that applications with low exposed parallelism (*e.g.* AP) perform poorly on the GPU regardless of the data synchronization mechanism used. We argue that these applications can be further parallelized more easily with TM. Our evaluation suggests that KILO TM performs well (relative to fine-grained locking) on applications with low contention and high exposed parallelism (HT-L, ATM, CC). KILO TM performs poorly (relative to fine-grained locking) on applications with high contention and high exposed parallelism (HT-H, CL, BH). For these applications, limiting transaction concurrency lowers contention, and improves their performance with KILO TM. The programmer can lower contention in their appli-

cation via performance tuning, identifying transactions with high contention and reworking the code to reduce contention [59]. Applications with contention varying during execution (*e.g.* BH) may benefit from more dynamic mechanisms that control the transaction concurrency according to the current level of contention [57, 7].

Overall, our evaluation shows KILO TM captures 59% of fine-grained locking performance and is 128× faster than executing transactions serially on the GPU. Our evaluation with an idealized TM system indicates that TM on GPU can perform as well as fine-grained locking. These results motivate the need for TM on GPUs and the need for novel TM systems, like KILO TM, that better address the challenges in this new domain.

9. ACKNOWLEDGMENTS

We thank Doug Carmean, Mike O'Connor, Arrvindh Shriraman, Andrew Boktor, Ali Bakhoda, Henry Wong and the anonymous reviewers for their valuable comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

10. REFERENCES

[1] NVIDIA Forums - atomicCAS does NOT seem to work. http://forums.nvidia.com/index.php?showtopic=98444.
[2] R. Agrawal et al. Advances in Knowledge Discovery and Data Mining. chapter Fast Discovery of Association Rules. American Association for Artificial Intelligence, 1996.
[3] AMD. *R700-Family Instruction Set Architecture*, March 2009.
[4] D. Arnold et al. Stack Trace Analysis for Large Scale Debugging. In *IPDPS*, 2007.
[5] Ars Technica. IBM's new transactional memory: make-or-break time for multithreaded revolution, 2011.
[6] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.
[7] G. Blake, R. G. Dreslinski, and T. Mudge. Bloom Filter Guided Transaction Scheduling. In *HPCA*, 2011.
[8] C. Blundell et al. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *ISCA*, 2007.
[9] J. Bobba et al. Performance Pathologies in Hardware Transactional Memory. In *ISCA*, 2007.
[10] J. Bobba et al. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, 2008.
[11] A. Brownsword. Cloth in OpenCL, 2009.
[12] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. *Chapter 6 in GPU Computing Gems Emerald Edition*, 2011.
[13] J. Casper et al. Hardware Acceleration of Transactional Memory on Commodity Systems. In *ASPLOS*, 2011.
[14] D. Cederman et al. Towards a Software Transactional Memory for Graphics Processors. In *EGPGV*, 2010.
[15] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, 2006.
[16] H. Chafi et al. A Scalable, Non-blocking Approach to Transactional Memory. In *HPCA*, 2007.
[17] J. Chung et al. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *MICRO*, 2010.
[18] B. W. Coon et al. United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.), April 2008.
[19] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, 2010.
[20] W. J. Dally and B. Towles. *Interconnection Networks*. Morgan Kaufmann, 2004.
[21] D. Dice et al. Early Experience With a Commercial Hardware Transactional Memory Implementation. In *ASPLOS*, 2009.
[22] M. Ferdman et al. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *HPCA*, 2011.
[23] W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.
[24] W. Fung et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM TACO*, 6(2), 2009.
[25] J. E. Gottschlich et al. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *CGO*, 2010.
[26] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *PPoPP*, 2008.
[27] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. 2010.
[28] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.
[29] J. H. Kelm et al. WAYPOINT: Scaling Coherence to Thousand-Core Architectures. In *PACT*, 2010.
[30] G. Kestor et al. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *ICPE '11*, 2011.
[31] Khronos Group. OpenCL. http://www.khronos.org/opencl/.
[32] S. Kong et al. Time-Out Bloom Filter: A New Sampling Method for Recording More Flows. In *ICOIN*, 2006.
[33] E. A. Lee. The Problem with Threads. *Computer*, 39, May 2006.
[34] A. Levinthal and T. Porter. Chap - A SIMD Graphics Processor. In *SIGGRAPH*, 1984.
[35] E. Lindholm et al. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 2008.
[36] C. C. Minh et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *ISCA*, 2007.
[37] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *SIGGRAPH*, 1992.
[38] K. Moore et al. LogTM: Log-Based Transactional Memory. In *HPCA*, 2006.
[39] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar.-Apr. 2008.
[40] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, October 2009.
[41] NVIDIA Corp. *NVIDIA CUDA Programming Guide v3.1*, 2010.
[42] M. Olszewski et al. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT*, 2007.
[43] A. Ramamurthy. Towards Scalar Synchronization in SIMT Architectures. Master's thesis, University of British Columbia, 2011.
[44] B. Saha et al. Architectural Support for Software Transactional Memory. In *MICRO*, 2006.
[45] D. Sanchez et al. Implementing Signatures for Transactional Memory. In *MICRO*, 2007.
[46] L. Seiler et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *SIGGRAPH*, 2008.
[47] P. Shivakumar and N. Jouppi. *CACTI 5.0. Technical Report HPL-2007-167*. HP Laboratories, 2007.
[48] A. Shriraman et al. Flexible Decoupled Transactional Memory Support. In *ISCA*, 2008.
[49] M. F. Spear et al. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *SPAA*, 2008.
[50] F. Tabba et al. Transactional Conflict Decoupling and Value Prediction. ICS '11, 2011.
[51] D. Tarjan and K. Skadron. The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches. SC '10, 2010.
[52] S. Tomić et al. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *MICRO*, 2009.
[53] V. Vineet and P. Narayanan. CudaCuts: Fast Graph Cuts on the GPU. In *CVPRW '08*, 2008.
[54] B. A. Wallace. Merging and Transformation of Raster Images for Cartoon Animation. In *SIGGRAPH*, 1981.
[55] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, 2010.
[56] L. Yen et al. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, 2007.
[57] R. M. Yoo and H.-H. S. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *SPAA*, 2008.
[58] H. Zhao et al. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *PACT*, 2010.
[59] F. Zyulkyarov et al. Discovering and understanding performance bottlenecks in transactional applications. In *PACT*, 2010.