Energy Efficient GPU Transactional Memory via Space-Time Optimizations

Wilson W. L. Fung
Department of Computer and Electrical Engineering
University of British Columbia
wwlfung@ece.ubc.ca

Tor M. Aamodt
Department of Computer and Electrical Engineering
University of British Columbia
aamodt@ece.ubc.ca

ABSTRACT

Many applications with regular parallelism have been shown to benefit from using Graphics Processing Units (GPUs). However, employing GPUs for applications with irregular parallelism tends to be a risky process, involving significant effort from the programmer. One major, non-trivial effort/risk is to expose the available parallelism in the application as 1000s of concurrent threads without introducing data races or deadlocks via fine-grained data synchronization. To reduce this effort, prior work has proposed supporting transactional memory on GPU architectures. One hardware proposal, Kilo TM, can scale to 1000s of concurrent transaction. However, performance and energy overhead of Kilo TM may deter GPU vendors from incorporating it into future designs.

In this paper, we analyze the performance and energy efficiency of Kilo TM and propose two enhancements: (1) Warp-level transaction management allows transactions within a warp to be managed as a group. This aggregates protocol messages to reduce communication overhead and captures spatial locality from multiple transactions to increase memory subsystem utility. (2) Temporal conflict detection uses globally synchronized timers to detect conflicts in read-only transactions with low overhead. Our evaluation shows that combining the two enhancements in combination can improve the overall performance and energy efficiency of Kilo TM by 65% and 34% respectively. Kilo TM with the above two enhancements achieves 66% of the performance of fine-grained locking with 34% energy overhead.

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

Keywords

GPU, Transactional Memory

MICRO-46 December 7-11, 2013, Davis, CA, USA.

http://dx.doi.org/10.1145/2540708.2540743

1. INTRODUCTION

Originally designed as an accelerator for graphics rendering, Graphics Processing Units (GPUs) have evolved into a class of throughput computing processor capable of efficiently exploiting parallelism in applications in the form of thread-level parallelism. Modern GPUs feature hardware-accelerated thread spawning. This allows GPU applications to decompose their workloads into as many threads as possible without introducing significant overhead. The GPU runs thousands of these threads concurrently, interleaving their execution on hardware to tolerate latencies from aggressive pipelining and off-chip memory accesses. This allows GPU architectures to focus hardware resources on the actual computation instead of the assistance mechanisms that keep execution units busy (e.g. branch prediction and speculative execution). By combining this fine-grained multi-threading with wide SIMD hardware, GPUs can provide higher computing throughput while consuming less energy per operation than traditional multi-core processors.

Through the development of compute acceleration programming models such as CUDA [27] and OpenCL [20], GPUs are used to speedup a wide range of applications from scientific simulations to computer vision. Many of these applications have copious amounts of parallelism, and operate on their data with highly regular operations. Many recent works have demonstrated that applications with irregular parallelism can also be accelerated by GPUs [24, 22]. However, the development of these GPU applications tends to be a risky process that involves significant effort from the programmer. The irregular, data-dependent memory accesses in these applications make decomposing the workloads into thousands of threads a tedious, error-prone task. It requires great care from the programmer to avoid introducing deadlocks and/or data races when they use fine-grained locks and non-blocking data structures to eliminate unnecessary serialization among threads. To reduce this risk, software developers may choose to deploy a less efficient algorithm that involves more coarse-grained communication among threads to perform the same computation, and scale the amount of data instead to achieve the same level of parallelism required by the GPU. GPU applications parallelized using this *weak scaling* approach tend to perform sub-optimally due to contention at the memory subsystem. The decreased computational efficiency in turn makes GPUs less appealing to software developers.

Transactional memory (TM) [17] has gained significant interest as a promising way to reduce this development risk in multi-threaded applications. While most research and development effort focuses on supporting TM on multi-core processors [15, 39, 33], there are several recent proposals for supporting TM on GPU architectures. Kilo TM [11] is a hardware TM (HTM) proposal designed to support 1000s of concurrent transactions on GPU architectures. Despite the promising scalability, concerns over perfor-

mance and energy overhead of Kilo TM may deter GPU vendors from incorporating it into future designs. In this paper, we address these concerns by evaluating and analyzing the overhead of Kilo TM. The insights from this analysis leads to two distinct enhancements, warp-level transaction management and temporal conflict detection.

The contributions of this paper are:

- It analyzes the performance and energy overhead of supporting transactional memory on GPUs.
- It proposes *warp-level transaction management*, which manages non-conflicting transactions within a warp as a single entity during validation and commit. This aggregates communication to reduce protocol overhead, and more importantly, captures spatial locality from multiple transactions to increase memory subsystem utility.
- It proposes and evaluates two intra-warp conflict resolution schemes to resolve conflicts within a warp.
- It proposes *temporal conflict detection*, a low overhead mechanism that uses globally synchronized timers to detect conflicts in read-only transactions.

Warp-level transaction management and temporal conflict detection together improve the overall performance of Kilo TM by 65% while reducing the energy consumption by 34%. Kilo TM with the two enhancements achieves 66% performance of fine-grained locking with 34% energy overhead. More importantly, the enhancements allow applications with small, rarely-conflicting transactions to perform equal or better than their fine-grained lock versions. We believe that GPU applications using transactions can be incrementally optimized to reduce memory footprint and transaction conflicts to take advantage of this. Meanwhile the transaction semantics can maintain correctness at every step, providing a low-risk environment for exploring optimizations.

2. BACKGROUND

This section discusses aspects of the baseline GPU architecture that are relevant for this paper. It also summarizes the transactional memory programming model and briefly describes the original design of Kilo TM [11].

2.1 Baseline GPU Architecture

Figure 1 shows the high-level overview of our baseline GPU architecture. A GPU application starts on the CPU and uses a compute acceleration API such as CUDA or OpenCL to launch work onto the GPU [25, 27, 20]. Each launch consists of a hierarchy of *scalar threads* that execute the same *compute kernel*. The thread hierarchy organizes threads as *thread blocks*. Each block is dispatched to one of the heavily multi-threaded SIMT cores as a single unit of work. It stays on the SIMT core until all of its threads have completed execution. Threads within a block can communicate via an on-chip scratchpad memory called *shared memory* (*local memory* in OpenCL), and can synchronize quickly via hardware barriers. The SIMT cores access a distributed, shared, read/writeable last-level (L2) cache and off-chip DRAM via an on-chip network.

The application may launch a thread hierarchy that far exceeds the GPU on-chip capacity. The GPU command unit automatically dispatches as many thread blocks as the GPU on-chip resources can sustain, and dispatches the rest of the thread hierarchy as resources are released by completed thread blocks.

SIMT Execution Model. In the SIMT core, scalar threads are managed as SIMD execution groups called *warps* (*wavefronts* in AMD terminology). In this paper, each warp contains 32 scalar threads [27]. Each warp has a hardware SIMT stack that serializes

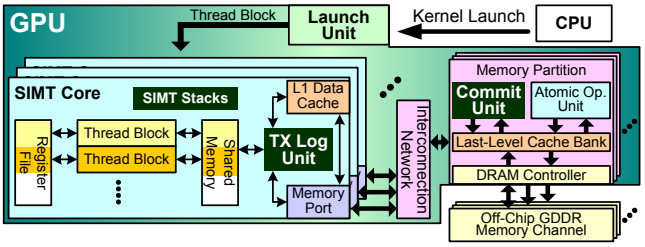

Figure 1: High-Level GPU architecture exposed by the CUDA programming model. TX Log Unit, Commit Unit added for Kilo TM. SIMT stack modified to support transactions [11]. TX Log Unit extended to use shared memory for intra-warp conflict resolution.

the execution of different subsets of threads that diverge to different control flow paths [10].

Memory Subsystem. For each memory instruction, each scalar thread in the warp can generate a scalar memory access. These accesses are served in parallel by the memory subsystem in the SIMT core. Shared memory accesses are served by 32 shared memory banks. Accesses contending for the same bank are serialized. For *global* and *local* memory spaces [27], accesses from different threads in the same warp to the same 128-Byte memory chunk are merged (coalesced) into a single wide access. The memory subsystem services one wide access per cycle.

The L1 data caches in the SIMT cores are not coherent [26]. The applications in this paper store shared data in the *global memory space* that can be updated by threads from different SIMT cores. To avoid access to stale (non-coherent) data, all global memory accesses skip the L1 cache. They are serviced directly by the L2 cache bank at the corresponding memory partition.

Each thread can store thread-private data and spilled registers in a private *local memory space* [27]. The local memory is stored in off-chip DRAM and cached in the per-core L1 data cache and the shared 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 fall into the same cache line in the L1 cache and are serviced in parallel in a single cycle. Kilo TM stores the read and write-logs of each transaction in its local memory space to facilitate L1 caching.

Atomic Operation Units. Current GPUs provide hardware atomic operations for simple single-word read-modify-write operations [27, 20]. The SIMT cores send atomic operation requests to a set of raster operation units in the memory partitions (Atomic Op. Unit in Figure 1) to perform these read-modify-write operations to individual locations atomically within the memory partitions [4]. Programmers can use these atomic operations to implement locks.

2.2 Transactional Memory

Transactional memory [17, 16] simplifies development for parallel software by providing the programmer with the illusion that code regions, called *transactions*, execute atomically in isolation. With TM, the programmer does not need to protect shared data in memory with locks to enforce mutual exclusion. Instead, he/she should place code routines that may access the shared data inside transactions. At runtime, threads may execute transactions in parallel. An underlying TM system monitors data accesses from transactions for data-races (known as *conflicts* in TM). The TM system resolves conflicts between two transactions by restarting one of them, effectively serializing their execution. This automatic se-

rialization ensures that some of the transactions can always make forward progress, avoiding system-wide deadlocks.

For example, in the ATM benchmark (Section 7), each scalar thread uses a transaction to transfer funds from one account into another account atomically. Transactions that try to access the same account simultaneously are automatically serialized, while transfers between disjoint sets of accounts are processed in parallel.

2.3 Kilo TM

Kilo TM [11] is a proposed HTM system designed for GPU architectures. With Kilo TM, programmers specify weakly-isolated transactions within the compute kernels. Each scalar thread can execute an independent transaction, which may conflict with any other transactions in the system. Nested transactions are flattened.

Figure 1 highlights the hardware implementation of Kilo TM: an enhanced SIMT execution hardware to handle control-flow divergence due to transaction aborts within a warp, a transaction log unit in each SIMT core that manages the read- and write-logs for each transaction, and a set of commit units which orchestrate the validation and commit of transactions to increase commit parallelism.

Kilo TM uses value-based conflict detection [8, 28] to avoid expensive tracking of data ownership among thousands of threads while permitting unbounded transactions. Each transaction buffers the value loaded from global memory in a read-log. Prior to commit, the transaction *validates* its read-set – it compares every value in the read-log to the most recent value in the global memory to detect conflicts with previously committed transactions. If all the values match, the transaction is free of conflicts and can writeback the data buffered in its write-log to the global memory. To allow multiple transactions to commit in parallel, each commit unit uses a *last writer history unit* to detect conflicting transactions that try to commit concurrently. These conflicts are called *hazards*. The commit unit serializes transactions with hazards so that values committed from one transaction will be visible to the other one.

Our evaluation shows that GPU TM applications running on a simulated NVIDIA Fermi GPU extended with our baseline Kilo TM only captures 40% of the fine-grained locking performance, and consumes 2× the energy. Notice that this is lower than the 59% relative performance between Kilo TM and fine-grained lock in our previous evaluation [11]. The discrepancy is mainly contributed by the different core to memory ratio between the NVIDIA Fermi architecture and the cache-extended Quadro FX5800 architecture modeled in our previous evaluation [11].

3. WARP-LEVEL TRANSACTION MANAGEMENT (WARPTM)

The GPU memory subsystem is designed to handle accesses with high spatial locality. The L2 cache bank in each memory partition can access a quarter of the cache block (32 Bytes) in a single cycle, and the accessed data is delivered through an interconnection network that can inject 32 bytes per cycle at each port. The use of wide cache ports and wide flit size matches well with the off-chip DRAM architecture, and delivers high bandwidth with relatively low control hardware overhead. GPU uses special coalescing logic to capture the spatial locality among scalar memory accesses from threads in the same warp.

Although Kilo TM executes transactions in the same warp in parallel, each transaction validates and commits individually. This scalar management simplifies the design of the commit units – conflicts between transactions within the same warp are handled just as conflicts between any two transactions in the system. This design simplification results in an inefficient utilization of the memory subsystem. Every cycle, each commit unit can only send one scalar, 4-Byte request (validation or memory writeback of a single 4-Byte word for a single transaction) to the L2 cache bank. This wastes at least 7/8 of the L2 cache bandwidth, creating a major energy overhead for Kilo TM.

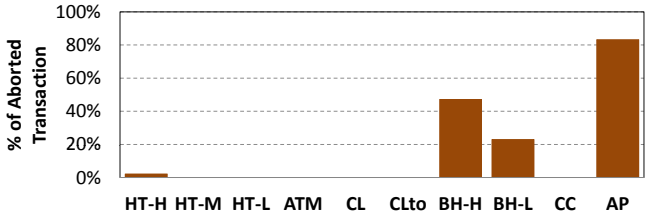

Figure 2: Transaction conflicts within a warp.

The scalar management also introduces many extra protocol messages between the SIMT cores and the commit unit. Each transaction has to generate at least three messages: (1) a done-fill message to indicate that the entire read-set and write-set have arrived at the commit unit, (2) a response from the commit unit to relay the outcome of value-based conflict detection local to the unit, (3) a message broadcasting the overall transaction outcome to each commit unit. These extra protocol messages can significantly increase interconnection traffic.

Even though it is possible to improve the L2 cache bandwidth utility by extending each commit unit with an access combine buffer that opportunistically accumulates multiple scalar accesses and coalesces them into wider accesses, we believe a simpler alternative is to exploit the spatial locality that already exists in a warp. We call this Kilo TM extension *warp-level transaction management* (WarpTM). WarpTM uses a low-overhead *intra-warp conflict resolution* mechanism to detect and resolve all conflicts within a warp before validating and committing the warp via the commit units. The warp that is free of intra-warp conflicts can then be managed as a single entity, allowing various optimizations that boost the performance and efficiency of Kilo TM without introducing complex control logic.

The rest of this section describes the optimizations enabled by warp-level transaction management and the hardware modifications required to realize these optimizations. The implementation of intra-warp conflict resolution will be discussed in Section 4.

3.1 Optimizations Enabled by WarpTM

Without any potential conflicts within the warp, the commit unit can coalesce the validation and commit requests from all transactions within the warp into wider accesses to the L2 cache. It can also aggregate the protocol messages so that it is relaying the validation outcomes of the entire warp. The benefits from WarpTM can be categorized as follows.

Eliminate Futile Validation. By resolving the conflicts within a warp prior to the global commit, transactions that would have failed can abort before sending any traffic out of the SIMT core. This can reduce contention at the commit units for workloads with high contention, improving their performance and energy usage. Figure 2 shows that conflicts between transactions within the same warp, *intra-warp conflicts*, rarely occur in most of our workloads. The exceptions, BH and AP, both feature a high contention period when many transactions are trying to append leaf nodes to a small tree.

Aggregate Control Messages. Figure 3 illustrates the protocol messages that are sent between the SIMT core and the commit units to commit a transaction. Notice that the original proposal of Kilo TM already aggregates the read-set and write-set messages and the

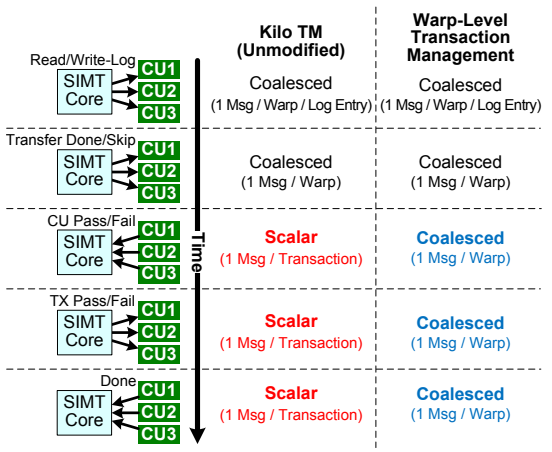

Figure 3: Kilo TM Protocol Messages.

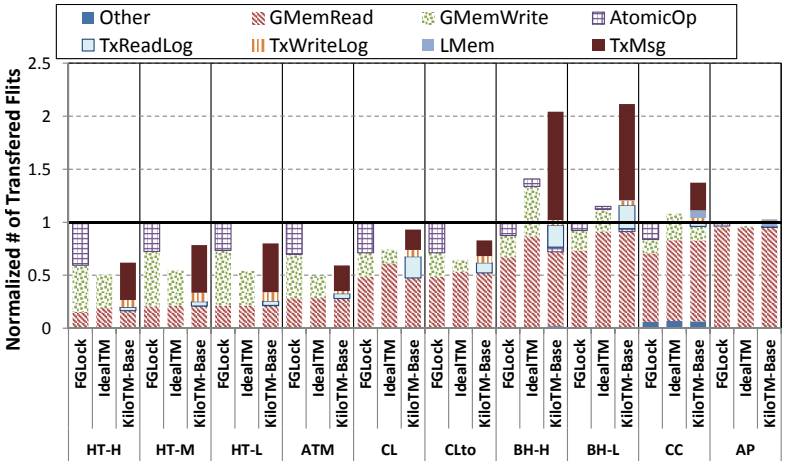

Figure 4: Interconnection Traffic Breakdown.

done-fill messages from multiple transactions in the warp. However, it does not aggregate the remaining protocol messages.

With WarpTM, in the absence of potential intra-warp conflicts, the commit unit can wait until all transactions have finished validation and combine their outcomes (pass/fail) into a single message. After receiving replies from all the commit units, the SIMT core can also combine the final outcomes of the entire warp into a single message that is broadcast to the commit units.

Figure 4 shows the interconnection traffic breakdown for our workload with fine-grained locks, Ideal TM, and Kilo TM. The protocol messages for Kilo TM (TxMsg) on average account for 36% of the interconnection traffic.

Validation and Commit Coalescing. While applications with irregular parallelism tend to exhibit less spatial locality among threads within a warp, coalescing memory accesses from the same warp can still significantly reduce the number of accesses. With the original Kilo TM, the memory accesses performed by threads during transaction execution are already coalesced just as the non-transactional memory accesses. The commit units, however, generate scalar memory accesses for validation and commit of transactions to avoid explicitly handling intra-warp conflicts. This simplifies the design of the commit units. With WarpTM, each commit unit knows *a priori* that all transactions from the same warp are free of intra-warp conflicts. Consequently, the value-comparison outcome for the validation of one transaction will not be changed after another transaction in the same warp has committed. As a result, the commit unit can always merge the scalar memory accesses for the validation of multiple transactions in the same warp into wider accesses. We call this *validation coalescing*. Similar reasoning permits the commit unit to merge scalar memory writeback accesses for the commit as well. We call this *commit coalescing*.

Figure 5 shows the amount of L2 cache access from the commit units that can be reduced through validation and commit coalescing. On average, coalescing can reduce the number of validation requests and memory writeback requests by 40% and 39% respectively. Without coalescing, scalar accesses that exhibit spatial locality tend to hit the L2 cache. However, they still waste L2 cache bandwidth supplied by the wide cache ports, and they will consume more L2 cache miss-status holding registers (MSHR) that track accesses waiting for in-flight requests from DRAM.

Validation and Commit coalescing can benefit any GPU TM system as long as the GPU still employs the SIMT execution model, and accesses memory in large contiguous chunks. Even with wide-

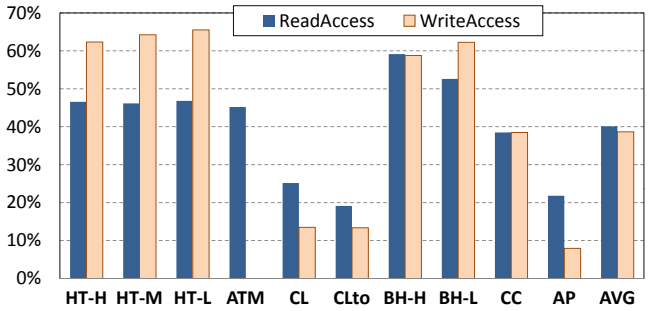

Figure 5: Reduction in L2 cache accesses from the commit units via validation and commit coalescing.

channel 3D DRAMs, there are tangible benefits in amortizing SRAM and DRAM control logic by accessing data in large chunks, as long as the applications contain sufficient memory access spatial locality. Most existing GPU applications do.

3.2 Hardware Modification to Kilo TM

Aside from implementing intra-warp conflict resolution (described in Section 4), WarpTM requires modification to the commit units. Each commit unit contains a small *read/write buffer* that caches the read-logs and write-logs of committing transactions. In the original Kilo TM, the commit unit only accesses one word from the read-log or write-log of one transaction in each cycle. The read/write buffer can supply this bandwidth with a narrow (4-Byte) port. WarpTM requires this read/write buffer to have a wide (64-Byte) port. The wide port allows the read-sets and write-sets from multiple transactions in the same warp to be retrieved in a single cycle. Each commit unit also needs to be extended with a memory coalesce logic unit to merge multiple scalar accesses that head to the same cache block into a wider access.

4. INTRA-WARP CONFLICT RESOLUTION

Developing a low overhead mechanism to detect conflicts among transactions within a warp is the key challenge in enabling WarpTM. Each transaction in Kilo TM stores its read-set and write-set as linear logs in the local memory space. The logs are organized physically such that each transaction may only access one word in its logs per cycle. Detecting conflicts between two transactions naively requires traversing the linear logs repeatedly, once for each

word in the read- and write-log, for a full comparison of the logs. Even if the logs are fully cached in the L1 data cache, a full pair-wise comparison among T transactions still requires $O(T^2 \times N)$ traversals, where N is the combined size of the read- and write-logs of a transaction. The overhead would likely negate any performance and energy benefit from WarpTM.

4.1 Multiplexing Shared Memory for Resolution Metadata

We noticed that many applications that require irregular communication between threads in different SIMT cores make little use of shared memory (the on-chip scratchpad memory). This observation is exploited in NVIDIA Fermi GPUs by allowing part of the shared memory storage to be configured as the L1 data cache [26]. The non-configurable part of the shared memory remains unused in most of these applications. Given that intra-warp conflict resolution only involves communication within a warp, we propose to use this underused storage as temporary buffers for intra-warp conflict resolution. When a warp has finished executing its transactions, it allocates a buffer in the shared memory to perform the intra-warp conflict resolution. The warp then uses this buffer to store metadata for its intra-warp conflict resolution, and releases the buffer after the resolution is done. This allows the buffer to be time-shared by multiple warps – a technique known as shared memory multiplexing [37]. The shared memory storage can also be partitioned into multiple buffers to allow multiple warps to interleave their intra-warp conflict resolution to hide access latency for read/write-log accesses that miss the L1 data cache.

To support applications that use shared memory for other computations, the GPU command unit can be extended to launch fewer thread blocks on each SIMT core according to amount of metadata storage reserved by the programmer.

4.2 Sequential Conflict Resolution with Bloom Filter (SCR)

Using the shared memory to store a bloom filter [39, 7], we have developed a sequential conflict resolution scheme that always prioritizes the transactions executed by the lower lanes in the warp. Threads with lower thread ID are assigned to the lower lanes in the warp. In this scheme, the transaction with the lowest lane in the warp first populates the bloom filter with its write-set. Each subsequent transaction in the warp first checks to see if its read-set or write-set hits in the bloom filter. If so, this transaction conflicts with one of the transactions in the prior lanes, and it is aborted. Otherwise, the transaction adds its write-set to the bloom filter to make its write-set visible to the subsequent transactions in the warp. The accumulative nature of the bloom filter allows each transaction to compare its read and write-set against the write-set of all transactions in the prior lanes. While this scheme does reduce the number of transaction log traversals, its sequential nature makes poor use of the bandwidth provided by the L1 cache and shared memory.

4.3 2-Phase Parallel Conflict Resolution with Ownership Table (2PCR)

In SCR, each transaction in the warp is essentially matching its read-set and write-set with the aggregated write-set of all the transactions in the prior lanes. This matching is inherently parallel if each lane has a pre-constructed record of the aggregated write-set from its prior lanes. Also, the priority among lanes is known in advance, so that multiple conflicting lanes can resolve the conflicts unanimously in parallel without extra communication. From these two insights, we have developed *two-phase parallel intra-warp conflict resolution* (2PCR). First, the transactions in the warp collaboratively construct an *ownership table* in parallel from the write-logs of every transaction in the warp. Each transaction then checks this ownership table for conflicts with another transaction in a prior lane. If such a conflict exists, the transaction aborts itself.

Each entry in the ownership table represents a region in global memory. Its value contains the lane ID (5-bit) of the lowest lane that intends to write to the region and an extra null-bit to indicate if none of the lanes intends to write to the region. Each entry, padded with two used bits, occupies a byte in shared memory. To construct the ownership table, each transaction traverses through its write-log to read out the locations in its write-set. For each location, the transaction calculates the index of the corresponding entry in the ownership table hashing the location's address. It then updates the corresponding entry with its own lane ID if the existing value in the entry has a higher lane ID. The lockstep nature of a warp and the memory pipeline allows this to occur in parallel: At each step, every transaction reads one entry from its write-log, reads the existing value from the corresponding ownership table entry, compares the value against its own lane ID and updates the entry if its lane ID is lower. Special hardware similar to the bank conflict detection for shared memory is added to prevent two transactions from racing to update the same entry at the same step.

After constructing the ownership table, every transaction traverses through its read-log and write-log. For each location in the read-log, the transaction retrieves the lane ID from the corresponding ownership table entry. If the retrieved lane ID is not null and it is lower than the transaction's own lane ID, a conflict exists between this transaction and an earlier transaction. For each location in the write-log, the transaction also retrieves the lane ID from the corresponding ownership table entry. However, a conflict exists if the retrieved lane ID value does not equal the transaction's own lane ID. The different lane ID indicates that another transaction may overwrite the same location as this transaction. Notice that in this case, the retrieved lane ID will always be smaller, because the ownership table construction mandates that every entry contains the lane ID of the lowest lane intending to write to the corresponding region. Every transaction with any detected conflict aborts itself. The remaining transactions in the warp can then proceed and benefit from the optimizations enabled by WarpTM.

Figure 6 contains an example of the two-phase parallel intra-warp conflict resolution. The example consists of four transactions (X_1, X_2, X_3, X_4), each reading and writing to two locations in memory (except X_4, which only reads from one location). Here are the steps in this example:

1. Every entry in the ownership table is initialized to read-only (RO).
2. Every transaction updates the ownership table according to entry 1 in its own write-log.
3. The transactions proceed to entry 2 in their write-log. The ownership entry for 0x08 is not updated to X_4 because it is already owned by X_2, which has a higher priority. Meanwhile, the ownership of 0x10 is updated from X_3 to X_1.
4. The transactions proceed to entry 1 in their read-log for parallel matching. They all check the ownership table in parallel for the first lane that writes to 0x04, which is X_4. None of the transactions aborts at this point since their lane IDs are smaller or equal to X_4.
5. Every transaction proceeds to checking entry 2 in its own read-log. X_3 is aborted since 0x10 is already owned by X_1.
6. Each remaining transaction has ownership to address in entry 1 of its write-log.
7. After checking entry 2 in its write-log, X_4 is aborted due to a WAW conflict with X_2 at 0x08.

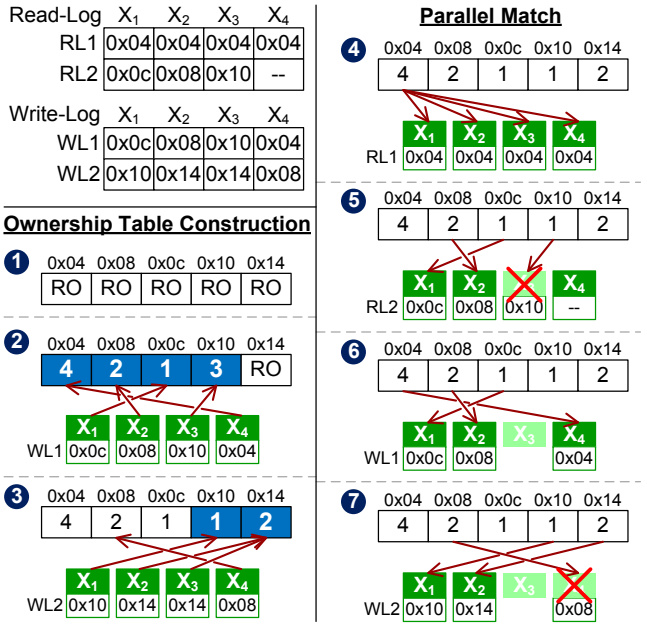

Figure 6: Two-Phase Parallel Intra-Warp Conflict Resolution. Each step shows the content of the ownership table and accesses from the transactions in the warp. RO = Read-Only

Finally, X_1 and X_2 are conflict free and can be validated and committed together via the commit units. Notice that this relatively narrow warp will take 15 steps with SCR. With 32-wide warps and larger transaction footprints in real workloads, the difference is even greater.

The accuracy of 2PCR depends on the size of the ownership table. Our evaluation shows that a 4K entry ownership table (requiring 4kB of storage in shared memory) performs comparably to an ownership table with infinite capacity. With a fixed-size ownership table, the accurracy of the intra-warp conflict resolution decreases for transactions with larger read/write-set. The average per-transaction footprint in our workloads spans between 3 to 36 words. Since GPU applications usually decompose larger input data into more threads, we believe that the per-transaction footprints in future GPU TM applications should not grow significantly beyond the footprints found in our workloads.

Notice that 2PCR tends to be less accurate than SCR. Since the ownership table is constructed in parallel assuming that every transaction in the warp will be committed, a transaction may unnecessarily abort due to a conflict with another aborted transaction. Nevertheless, our evaluation shows that the benefit from 2PCR outweighs the overhead from its additional false conflicts.

5. TEMPORAL CONFLICT DETECTION (TCD)

While warp-level transaction management can help reduce the extra interconnect traffic introduced by the Kilo TM protocol and improve L2 cache bandwidth utility, the fundamental overhead for Kilo TM still exists: Even without conflicts among transactions, each transaction has to reread its entire read-set for value comparisons prior to updating memory. To overcome this overhead, we propose *temporal conflict detection*, a low overhead mechanism that uses a set of globally synchronized timers to detect conflicts for read-only transactions.

A read-only transaction can occur dynamically when the transaction only conditionally writes to memory, or it can be explicitly introduced by programmers to ensure that code within the transaction can safely read from a shared data structure which may be updated occasionally by other transactions. A read-only transaction differs from a read-write transaction in that it can commit silently and locally as long as it has observed a consistent memory state – a memory state that does not contain partial memory updates from other committing transactions. Although the original design of Kilo TM can dynamically detect a read-only transaction at commit by observing an empty write-log, it does not exploit this information. Among the workloads we evaluated, read-only transactions account for ∼40% of the transactions in CL/CLto and ∼85% of the transactions in BH-L/BH-H. Being able to commit these read-only transactions silently without rereading their read-set can significantly reduce their energy and performance overhead.

TCD is a form of eager conflict detection that complements Kilo TM. Using a set of globally synchronized timers, it checks the accessed location as a transaction reads from global memory to ensure that the loaded data has not been modified since the transaction first reads from global memory. To do so, the system records when each word in memory was last written. Each transaction maintains the time of its first load, and each subsequent load in the transaction retrieves the time when the loaded word is last written. A retrieved last written time that occurs later than the time of the transaction's first load indicates a potential conflict, because the value at the loaded location have been modified since the first load. If none of the words loaded by the transaction has been written since the first load, the value of every word read by the transaction coexists in a instantaneous snapshot of global memory that existed at the time of the first load. A read-only transaction satisfying this condition has effectively obtained all of its input value from this snapshot, and appears to have executed instantly with respect to other transactions. Therefore, the read-only transaction can commit directly without further validation.

Notice that the instantaneous snapshot observed by the transaction via TCD may occur in the midst of a memory writeback from a committing transaction. This causes the snapshot to contain partial updates from a transaction, which is not a consistent view of the memory. We have not observed this issue in the workloads we evaluated. Nevertheless, it is possible to extend TCD to detect if a transaction is loading from a location in the write-set of another committing transaction. The overhead for such detection mechanism involves a hardware buffer that conservatively records the last transaction that has written to each memory location, and extra protocol messages and hardware for maintaining a conservative set of committing transactions. We leave the exploration of this and other potential solutions as future work.

While timestamp-based conflict detection has been used in existing software TM systems [8, 36], each of these systems uses a global version number that is explicitly updated by software at transaction commit. The globally synchronous timers used by temporal conflict detection increment locally at each hardware component. This eliminates the bottleneck to update and access the global version number. Ruan et al. [31] also proposed extending Orec-based STM systems with synchronous hardware timers. Their approach embeds timestamps in the ownership record of each transaction variable in memory, whereas we uses a small on-chip storage to conservatively record when each word is last written.

5.1 Implementation

Figure 7 shows the hardware modification to implement TCD: A 64-bit globally synchronized timer in each SIMT core and memory partition, a first-read time table in each SIMT core recording when

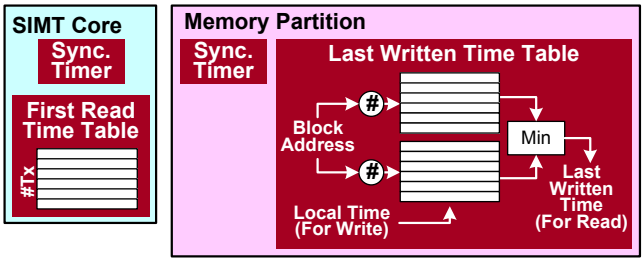

Figure 7: Hardware extensions for temporal conflict detection.

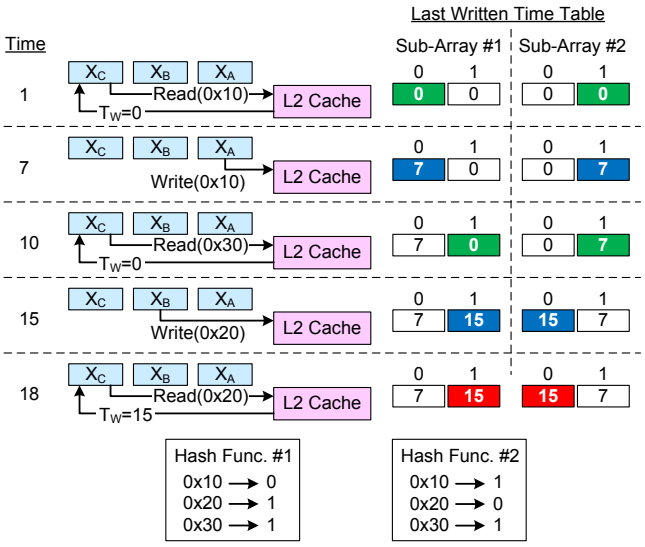

Figure 8: Temporal conflict detection example.

each transaction sends its first load, and a last written time table in each memory partition that conservatively records the last written time of each 128-Byte block in the partition. Existing hardware already implements timers synchronized across components [18, Section 17.12.1] to provide efficient timer services. We implement the last written timetable with a *recency bloom filter* [11]. Each recency bloom filter consists of multiple sub-arrays of timestamps. Each 128-Byte block in the memory partition maps to an entry in each sub-array of the recency bloom filter via a different hash function. Whenever a word in the 128-Byte block is updated by a committing transaction in the L2 cache, the corresponding entries in every sub-array of the recency bloom filter are updated with the value from the synchronized timer. Each transactional load served by the L2 cache retrieves a timestamp from each sub-array in the recency bloom filter and returns the minimum of those timestamps along with the data to the SIMT core. This timestamp is compared against the time of the transaction's first load to detect conflicts.

At 700MHz, 64-bit timers only roll over every few hundred years. In the event that it happens, the TM system can handle the rollover by validating the read-set of all running transactions through value-based conflict detection. For the transactions that remain valid, the TM system resets their first read time to zero. Although not needed for correctness, it should also reset the last written time table so that the table will not report overly conservative last written time.

5.2 Example

Figure 8 walks through how TCD detects an inconsistent view of memory during transaction execution. The example consists of two

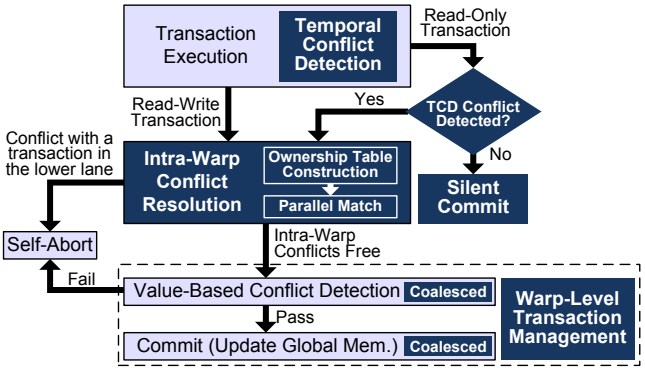

Figure 9: Kilo TM in enhanced with warp-level transaction management and temporal conflict detection.

committing transactions, X_A and X_B, and one read-only transaction X_C. At time T=1, X_C starts execution and issues its first load to location 0x10. X_A then commits and updates the value at 0x10 at time T=7. At the same time, X_A also updates the timestamps corresponding to location 0x10 in the recency bloom filter (consists of 2 sub-arrays with 2 timestamp each). X_C issues its second load to location 0x30 at T=10, and the recency bloom filter returns with T_W[0x30]=0. Notice that even though the original value at 0x10 is overwritten by X_A, the updated value is not visible to X_C and does not constitute a conflict. X_B commits and updates the value at 0x20 at time T=15, and updates the recency bloom filter. X_C issues its third load to location 0x20 at T=18, and the recency bloom filter returns with T_W[0x20]=15, which is later than the first load from X_C. This is a conflict for X_C because the value loaded is not the same value at 0x20 at T=1. The memory state observed by X_C does not correspond to an actual global memory state at any time – an invalid snapshot.

5.3 Integration with Kilo TM

In this paper, Kilo TM uses TCD to allow read-only transactions to commit silently in the absence of detected conflicts. The recency bloom filter does not perfectly record the time when each word is last written. Aliasing of timestamps in the filter can lead to false positives in TCD. To reduce the penalty of falsely detected conflicts, read-only transactions with detected conflict are given a second chance to commit through the commit units as read-write transactions in Kilo TM. In this way, we can use a relatively small filter with coarse granularity (128-Byte chunk maps to the same entry) to allow most conflict-free read-only transaction to commit silently, and use value-base conflict detection for the situations that require finer granularity detection. Similar hierarchical validation schemes are used in NOrec [8] and the software GPU TM system by Xu et al. [36].

6. PUTTING IT ALL TOGETHER

Our two proposed enhancements to Kilo TM, warp-level transaction management and temporal conflict detection, can work together to further improve performance. Figure 9 shows the overall design of Kilo TM with both enhancements enabled. In this enhanced Kilo TM, each transaction uses TCD to eagerly detect conflicts for each global memory read during its execution. Writes to global memory are buffered in the write-log as in the original Kilo TM. After the transaction has completed execution, if it is a read-only transaction (i.e. containing an empty write-log) and TCD has not detected any conflict, it can commit silently. The remain-

Table 1: GPGPU-Sim Configuration

# SIMT Cores	15
Warp Size	32
SIMD Pipeline Width	16×2
# Threads / Core	1536
# Registers / Core	32768
Branch Divergence Method	PDOM [10]
Warp Scheduling Policy	Greedy-then-oldest [30]
Shared Memory / Core	16KB
L1 Data Cache / Core	48KB, 128B line, 6-way assoc. (transactional+local mem. access only)
L2 Unified Cache	128KB/Memory Partition, 128B line, 8-way assoc.
Interconnect Topology	1 Crossbar/Direction
Interconnect BW	32 (Bytes/Cycle) (288GB/s/Dir.)
Interconnect Latency	5 Cycle (Interconnect Clock)
Compute Core Clock	1400 MHz
Interconnect Clock	1400 MHz
Memory Clock	924 MHz
# Memory Partitions	6
DRAM Req. Queue	32 Requests
Memory Controller	Out-of-Order (FR-FCFS)
GDDR5 Memory Timing	Hynix H5GQ1H24AFR
Total DRAM BW	177GB/s
Min. L2 Latency	330 Cycle (Compute Core Clock)
DRAM Scheduler Latency	200 Cycle (Compute Core Clock)
Kilo TM	
Commit Unit Clock	700 MHz
Validation/Commit BW	1 Word/Cycle/Memory Partition
# Concurrent TX	1, 2, 4, 8 Warps/Core or No Limit (480, 960, 1920, 3840 or Unlimited # TX Globally)
Last Writer History Unit	5kB
Intra-Warp Conflict Resolution	
Shared Memory Metadata	4kB/Warp (3 Concur. Resolution/Core)
Default Mechanism	2-Phase Parallel Conflict Resolution
Temporal Conflict Detection	
Last Written Time Table	16kB (2048 Entries in 4 Sub-Arrays)
Detection Granularity	128-Byte

Table 2: GPU TM Workloads.

Name	Abbr.	Description
Hash Table (CUDA)	HT-H	Populate an 8000-entry hash table.
	HT-M	Populate an 80000-entry hash table.
	HT-L	Populate an 800000-entry hash table.
Bank Account (CUDA)	ATM	Parallel transfer between 1M accounts.
Cloth Physics [3] (OpenCL)	CL	Cloth physics simulation of 60K edges.
	CLto	Optimized version of CL.
Barnes Hut [5] (CUDA)	BH-H	Build an octree with 30K bodies.
	BH-L	Build an octree with 300K bodies.
CudaCuts [35] (CUDA)	CC	Segmentation of a 200×150 pixel image.
Data Mining [1, 19] (CUDA)	AP	Data mining 4000 records.

Table 3: Power component breakdown for the added hardware specific to Kilo TM, warp-level transaction management, and temporal conflict detection.

Commit Unit			
	Size	Area (mm^2)	Power (mW)
Last Writer History - Look Up Table	3kB	0.010	6.3
Last Writer History - Recency Bloom Filter	2kB	0.010	6.6
Commit Entry Array	19kB	0.094	57.5
Read-Write Buffer	32kB	0.128	82.5
Per-Unit Total		0.242	153
All Units Total		1.454	918
Commit Unit (Warp-Level Transaction Management)			
Read-Write Buffer (Warp-Level TM)	32kB	0.731	260
Per-Unit Total		0.846	419
All Units Total		5.074	2512
Temporal Conflict Detection			
	Size	Area (mm^2)	Power (mW)
First Read Timetable (One per SIMT core)	12kB	0.034	25.5
Last Written Time Buffer (One per Mem. Part.)	16kB	0.078	52.3
All Units Total		0.979	696

ing transactions take part in the intra-warp conflict resolution to resolve all conflicts within the same warp. WarpTM then processes the still-active transactions in the warp with the assurance that they do not have conflict among each other. Our evaluation in Section 8 compares the performance and energy consumption of this combined TM system to the original Kilo TM.

7. METHODOLOGY

For our evaluation, we started with the version of GPGPU-Sim [2] from our previous work [11, 12]. It extends GPGPU-Sim version 3.1.2 with support for transactional memory, and includes the performance model for Kilo TM. We incorporated GPUWattch [21] into this version of GPGPU-Sim, and modified it to model the timing and power of our proposed enhancements. We configured the modified GPGPU-Sim to simulate a GPU similar to Geforce GTX 480 (Fermi), with 16kB of shared memory storage per SIMT core. Table 1 lists the major microarchitecture configurations.

We used the workloads from our previous evaluation [11, 13] to evaluate the proposed improvements to Kilo TM. In addition to the original input, we also added new inputs that varies the amount of contention for BH and HT. We have created an optimized version of CL (CLto). In this version, each thread loads the read-only data into its register file before entering the transaction to reduce the read-set of the transaction. Table 2 summarizes each of our workloads.

7.1 Power Model

We modeled the power overhead of Kilo TM by estimating the access energy of the various major structures in the commit units implemented in the 40nm process with CACTI 6.5 [23]. We multiplied the access energies with the operating frequency, conservatively assuming that the structures are accessed every cycle, to estimate their power overhead. For small memory arrays, CACTI provides a conservative area and energy estimates as it automatically partitions the array into sub-arrays when a single array is sufficient [32]. Similarly, we modeled the power overhead of TCD with the full activity power to the last written time buffer in each memory partition and the first read timetable in each SIMT core. Table 3 shows the estimated power for each component in Kilo TM and temporal conflict detection. The Kilo TM specific hardware consumes 0.9W in total, and extending it to support WarpTM increases the consumption to 2.5W. The power increase is introduced by having a wider port to the read-write buffer (See Section 3.2). The hardware that implements TCD consumes 0.7W. Kilo TM with WarpTM and TCD consumes a total of 3.2W.

For parts of the GPU microarchitecture not specific to Kilo TM, we used GPUWattch [21] to estimate the average dynamic power consumed by each workload with the different synchronization mechanisms. This captures the difference in microarchitecture activity between fine-grained locks and Kilo TM (with and without the proposed enhancements). This includes extra L1 cache accesses for the transaction logs, extra L2 accesses for value-based conflict detection, extra interconnection traffic for Kilo TM protocol messages, and accesses to shared memory for intra-warp conflict resolution. We added 59W of leakage power, 9.8W of constant clock power and the dynamic power of the Kilo TM specific hardware to the average dynamic power reported by GPUWattch to obtain the total power. Finally, we multiplied this total power by the execution time to obtain the total energy required to execute each workload.

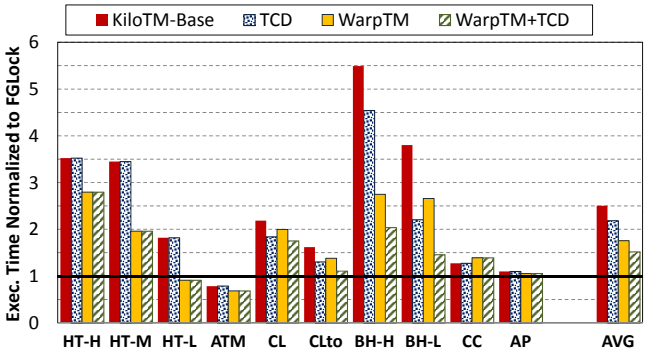

Figure 10: Execution Time. Lower is better.

Table 4: Performance-Optimal Concurrent Transaction Limit and Abort-Commit Ratio. Base = KiloTM-Base. NL = No Limit.

	Concurrent Transaction Limit (#Trans. Warps/SIMT Core)				Aborts per 1000 Committed Trans.			
	Base	TCD	WarpTM	TCD+WarpTM	Base	TCD	WarpTM	TCD+WarpTM
HT-H	2	–	2	–	50	–	107	–
HT-M	2	–	8	–	7	–	84	–
HT-L	4	–	8	–	2	–	63	–
ATM	1	–	4	–	1	–	27	–
CL	2	2	2	2	84	55	149	97
CLto	2	2	2	4	85	49	102	99
BH-H	2	2	4	4	23	23	53	56
BH-L	8	4	8	8	7	5	17	20
CC	NL	–	NL	–	6	–	6	–
AP	1	–	1	–	264	–	318	–

8. EXPERIMENTAL RESULTS

In this section, we evaluate the performance and energy efficiency of our proposed enhancements to Kilo TM: warp-level transaction management (WarpTM) and temporal conflict detection (TCD). We also analyze the benefit of each optimization that is enabled by WarpTM, compare the two intra-warp conflict resolution approaches, and investigate the sensitivity of TCD to the the available hardware resource. Finally, we study the performance impact of L2 cache port width and number of SIMT cores on both fine-grained locks and Kilo TM.

8.1 Performance and Energy Efficiency

Figure 10 compares the execution time of the original Kilo TM (KiloTM-Base), Kilo TM with TCD enabled (TCD), Kilo TM with WarpTM (WarpTM) and a configuration with both enhancements enabled (WarpTM+TCD). We evaluate the performance of each configuration with different limits on the number of concurrent transactions, and select a limit for each workload that yields the optimal performance (See Table 4). The performance of each configuration with this optimal limit is normalized to the execution time of an alternative version of the application using fine-grained locking (FGLock) to illustrate their overhead with respect to a pure software effort. Figure 11 breaks down the energy consumption of the same set of Kilo TM configurations. Each breakdown is normalized to the total energy used by the fine-grained locking version of the same workload.

Without WarpTM and TCD, Kilo TM performs 2.5× slower than FGLock on average. The performance of BH-H is particularly poor. Our detailed investigation shows that the major slowdown occurs near the start of the octree-building kernel, where every thread tries to append its node to only a few branches in the octree. This behavior also stresses one memory partition in the GPU, working against the distributed design of Kilo TM. This effect slowly disappears as the octree grows to a point that conflicts become rare. As a result, BH-L, which scales the number of nodes in the octree by 10×, exhibits less slowdown. A similar attempt to reduce transaction contention does not work as well between HT-H and HT-M. The energy-per-operation penalty of Kilo TM is relatively lower (2× energy used vs. 2.5× performance slowdown) due to its lower activity from the poor performance.

Enabling temporal conflict detection for Kilo TM improves the performance of workloads that contain read-only transactions (CL, CLto, BH-H and BH-L). By allowing non-conflicting read-only transactions to commit silently, TCD reduces contention at the commit units and the memory subsystem. This performance improvement translates directly to energy savings as well. Across workloads with read-only transactions, TCD improves performance of Kilo TM by 37% while reducing energy per operation by 30%.

The optimizations enabled by warp-level transaction management (WarpTM) apply to a broader class of applications. In particular, coalescing of memory accesses from the commit units alleviates the bottlenecks at the L2 cache banks. Without this bottleneck, the reduction in transaction contention from HT-H to HT-M now leads to performance improvement for Kilo TM. Section 8.3 analyzes how each optimization from WarpTM contributes to speeding up Kilo TM in different applications. WarpTM does slow down CC, due to overhead from intra-warp conflict resolution (See Section 8.4). Overall, WarpTM speeds up Kilo TM by 42% and reduces energy per operation by 27%.

Kilo TM with both enhancements enabled captures the benefits from both TCD and WarpTM. The combined benefits, together with software optimizations, allow CLto to perform within 90% of the FGLock version while using about the same amount of energy. The original version of this cloth simulation workload starts out performing 2.2× worse than the FGLock version on Kilo TM. This kind of performance improvement indicates that a well designed TM system can complement optimization efforts from the software developer to produce efficient TM applications comparable to FGLock. Overall, Kilo TM with both TCD and WarpTM enabled achieves 66% of the performance of FGLock with only 34% energy overhead.

Concurrency Limit. Table 4 shows the limit on number of concurrent transactions that yields the best performance for each workload with the different Kilo TM configurations. Enabling WarpTM generally increases the optimal limit for each workload, as a result of reduced congestion at the memory subsystem and interconnection network compared to KiloTM-Base. The exceptions include high-contention workloads (HT-H), workloads that do not benefit from WarpTM (CC, AP), and workloads that may overflow the L1 cache with higher limits (CL). We find that enabling TCD has little effect on the optimal concurrency limit for our workload. Enabling TCD increases the limit for CLto, but the actual speedup from the increased limit is <5%.

Impact on Abort-Commit Ratio. Table 4 also shows the number of aborts per 1000 committed transactions for the different Kilo TM configurations. Enabling TCD reduces the amount of aborted transactions for CL and CLto by shortening the execution time span for each read-only transaction. This lowers the probability of another transaction overwriting a location in the read-set of the read-only transaction. Enabling WarpTM introduces significantly more aborted transactions due to false conflicts from intra-warp conflict resolution. Nevertheless, our evaluation has shown that WarpTM leads to an overall speedup and a net energy saving.

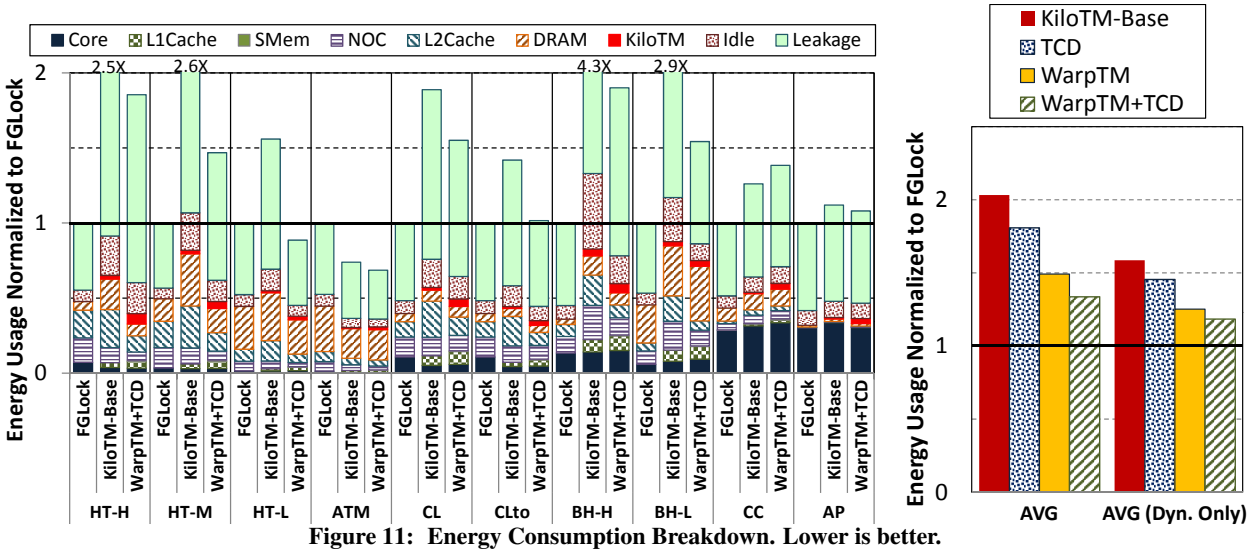

Figure 11: Energy Consumption Breakdown. Lower is better.

8.2 Energy Usage Breakdown

The energy usage breakdown in Figure 11 illustrates the relative contributions from different overheads of Kilo TM to its overall energy usage. Across our workloads, leakage and idle power contributes to >50% of the total energy consumption. Both leakage power and idle power (consists mostly of clock distribution power) persist throughout the program execution, so their contributions increases as execution time lengthens. Removing the contribution from leakage reduces the overall energy overhead of KiloTM-Base from 103% to 59%. Similarly, the pure dynamic energy overhead of KiloTM with WarpTM and TCD enabled is only 18% (versus 34% with leakage).

Aside from leakage and idle power, the memory subsystem (L2 Cache and DRAM) and the interconnection network (NoC) dominate the remaining portion of the dynamic energy usage. On average, the two combined contributes to ∼70% of the dynamic energy. For some workloads, the L2cache energy with KiloTM-Base is >2× that of FGLock. WarpTM and TCD have essentially eliminated this overhead, and on average, reduce the combined energy of L2 Cache, DRAM and NoC by 29%. Energy consumed by the SIMT cores only contributes to ∼25% of the dynamic energy usage. With Kilo TM, energy consumption by the core is lower than FGLock. This illustrates how an effective transaction concurrency control mechanism can substantially cut down the energy overhead for transaction re-execution. L1 accesses for transaction logs contribute to 5% of the dynamic energy usage. Adding intra-warp conflict resolution to support WarpTM increases this overhead by 23% (i.e. <2% increase to the overall energy usage). Finally, Kilo TM-specific hardware only contributes to 3% of the dynamic energy usage. Inclusion of hardware to support TCD and WarpTM only increases this to 8%.

8.3 WarpTM Optimizations Breakdown

Figure 12 breaks down the performance impact of each optimization introduced by WarpTM by enabling the optimizations one by one. In this analysis, we have used an ideal version of intra-warp conflict resolution that has perfect accuracy and no overhead. This isolates our analysis from performance issues that may arise from the particular conflict resolution scheme.

With only intra-warp conflict resolution enabled, WarpTM only impacts performance for applications that exhibit intra-warp transaction conflicts. While detecting such conflicts and resolving them

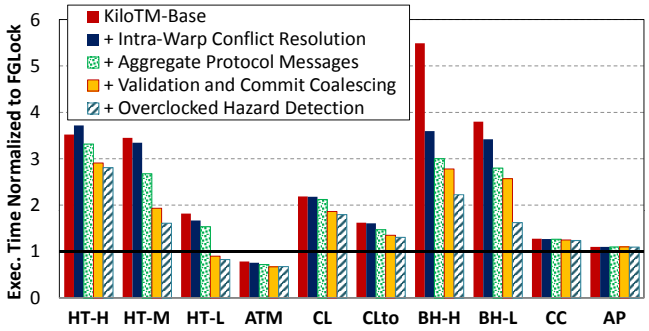

Figure 12: Performance impact from different optimizations enabled by WarpTM.

within the warp benefits BH, it slows down HT-H. In resolving a conflict within a warp, it is possible that a transaction that could eventually commit is aborted while the conflicting transaction is in turn aborted in the global phase of the commit.

Aggregating protocol messages speeds up applications by a varying amount. Without further optimizations, the aggregation simply exposes the memory subsystem bottleneck. Nevertheless, the average performance of Kilo TM is improved by 30% and the energy consumption is reduced by 17%.

Coalescing the memory accesses from the commit units allows the L2 cache bandwidth to be better utilized. This reduces the stress on the memory subsystem and improves performance for most of the applications. This optimizes Kilo TM by another 14% on average. BH-H and BH-L do not benefit from this optimization. This is surprising given our measurement has shown that validation and commit coalescing can reduce the amount of L2 cache accesses from the commit units by >50% for both workloads (See Figure 5). The reason is that the serial harzard detection in each commit unit becomes a bottleneck. If the hazard detection hardware were running at twice the clock frequency, BH-H and BH-L can finally benefit from validation and commit coalescing. With all the optimizations enabled, WarpTM with ideal intra-warp conflict resolution can speed up Kilo TM by 49%.

While we could parallelize the hazard detection in the commit units with additional hardware, we find that temporal conflict detection has removed this bottleneck entirely in BH-H and BH-L by allowing most of their read-only transactions to commit silently.

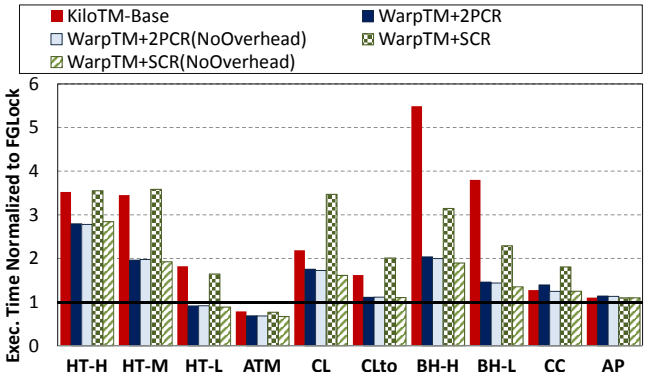

Figure 13: Comparison between different intra-warp conflict resolution. SCR = Sequential Conflict Resolution. 2PCR = 2-Phase Parallel Conflict Resolution.

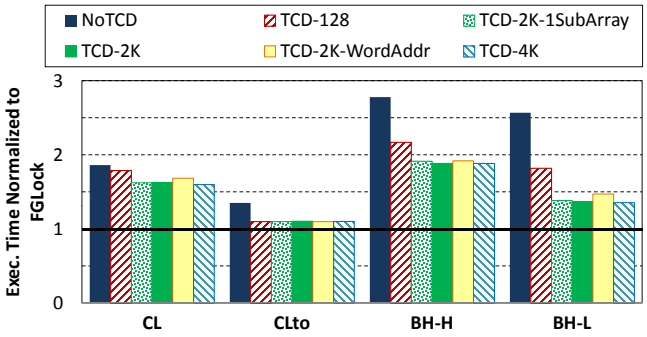

Figure 14: Performance of temporal conflict detection with different last written timetable organizations. Lower is better.

8.4 Intra-Warp Conflict Resolution Overhead

Figure 13 compares the performance between the two intra-warp conflict resolution mechanisms proposed in Section 4: sequential conflict resolution (SCR) in Section 4.2 and 2-phase parallel conflict resolution (2PCR) in Section 4.3. We evaluate the performance of each mechanism with and without modeling the overhead of the conflict resolution. Without modeling the overhead (NoOverhead), each warp finishes the intra-warp conflict resolution instantaneously, and does not generate traffic to the memory pipeline in the SIMT core when it traverses its logs or when it accesses the metadata in shared memory. This allows us to discern between two sources of performance overhead: transaction aborts due to inaccurate conflict resolution and the extra operations that implement the resolution itself. The no-overhead configuration of both SCR and 2PCR performs almost identically across all of our workloads, indicating that the accuracy of both mechanisms are roughly equivalent. However, the serial nature of SCR introduces significant overhead (an average 60% slowdown) to WarpTM, to the extent that most of its performance benefits are negated. 2PCR, on the other hand, can deliver similar accuracy as SCR with a much lower overhead (∼2% on average). The overhead in most cases is minor compared to the benefits from WarpTM, except for CC, where it causes 11% slowdown.

8.5 Temporal Conflict Detection Resource Sensitivity

Figure 14 shows the performance sensitivity of TCD to different last written time table organizations. While our default configuration uses 2048 entries in each memory partition (TCD-2K), a lower cost organization using 128 entries (TCD-128) can capture a significant portion of the performance benefit from TCD-2K. Doubling the size of the last written time table (TCD-4K) shows no further improvement over TCD-2K, indicating that the 2048-entry table is sufficient. Even though the data shows that a single 2048-entry sub-array (TCD-2K-1SubArray) performs comparably to our default organization, we do notice having multiple sub-arrays can reduce the effect of aliasing as transaction concurrency increases. Finally, contrary to our intuition, we notice that reducing the detection granularity of TCD from 128-Byte blocks to 4-Byte words (TCD-2K-WordAddr) decreases performance. We believe that reducing the granularity causes more entries in the recency bloom filter to be populated and the aliasing effect dominates.

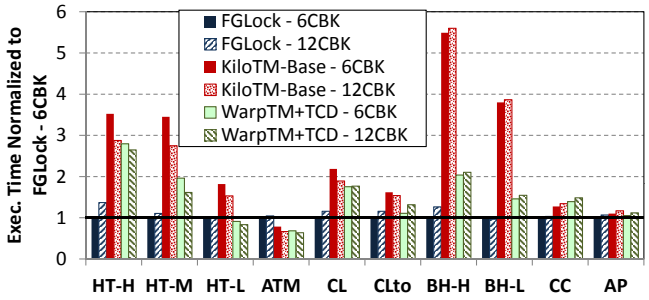

Figure 15: Performance impact with different L2 cache port widths. 6CBK = 6 L2 cache banks with 64-Byte ports. 12CBK = 12 L2 cache banks with 32-Byte ports. Lower is better.

8.6 Sensitivity to L2 Cache Port Width

In this section, we study the performance impact of L2 cache port width on FGLock, the original KiloTM (KiloTM-Base), and the enhanced KiloTM with both WarpTM and TCD (WarpTM+TCD). The L2 cache in our baseline GPU architecture is partitioned into 6 banks, each with a 64-Byte port (6CBK). In this study, we further divide each L2 cache bank into two subbanks with a 32-Byte port, resulting in 12 L2 cache banks across the system (12CBK). Other parts of the GPU architecture, including the total L2 cache bandwidth, remain identical between the two configurations. Figure 15 compares the performance between these two configurations.

Overall, the FGLock workloads with 32-Byte ports run 12% slower than with 64-Byte ports. HT-H and BH-H suffers a higher degree of load imbalancing among the L2 cache banks. The more congested L2 cache bank forms a tighter bottleneck with 32-Byte ports than with 64-Byte ports. While we did not observe this imbalance for CL and CLto, we did notice more atomic accesses from increased lock acquisition failures. The extra failures are caused by higher memory latency due to lower DRAM efficiency.

In this study, we also increased the number of commit units with the number of L2 cache banks due to their tightly-coupled design. The increased number of commit units can increase interconnection network traffic for Kilo TM, because each transaction needs to communicate with more commit units for validation and commit. We notice this extra traffic impacting the performance of Kilo TM for BH-H, BH-L, CC, and AP.

In other workloads, switching to 32-Byte ports improves performance for the original Kilo TM, because the commit units can use the port bandwidth more effectively with just scalar (4-Byte wide) accesses. In turn, the narrower L2 cache ports reduces the benefit provided by validation and commit coalescing. This lowers the average speedup of WarpTM over the original KiloTM from 43% to 40%. In particular, WarpTM does not speedup CLto at all.

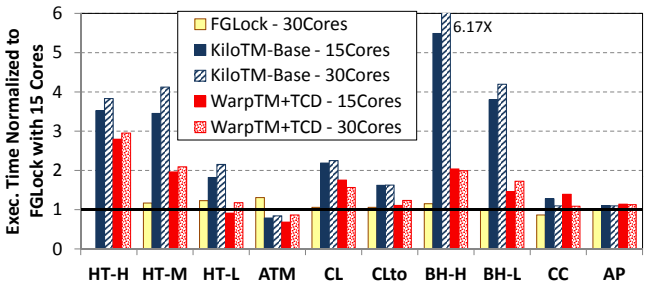

Figure 16: Performance impact from doubling the number of SIMT cores. Lower is better.

Nevertheless, in a GPU architecture with narrower L2 cache ports, KiloTM-Base obtains 49% of the FGLock performance. Kilo TM with WarpTM and TCD captures 76% of the FGLock performance, up from 66% with the wider L2 cache ports.

8.7 Sensitivity to Core Scaling

In this section, we explore the impact of increasing the number of SIMT cores on the overhead of Kilo TM. Specifically, we evaluate the performance of our workloads on a scaled up GPU architecture with 30 SIMT cores, doubled from our baseline configuration. Figure 16 compares the performance overhead of Kilo TM (and WarpTM+TCD) over FGLock with 15 and 30 SIMT cores. While doubling the SIMT cores increases concurrency in the GPU architecture, it slows down our FGLock applications by 9% on average. For this study, we have not scaled the memory subsystem with the core counts. The increased concurrency generate extra memory-level parallelism, but these extra concurrent memory accesses introduce more the L2 cache misses, resulting in a net performance loss [30]. We also noticed more atomic accesses from increased lock acquisition failure in HT-M, CL, and BH-H. Only CC can take advantage of the extra cores for a 13% speedup. Moreover, the FGLock version of HT-H runs into a livelock, so its performance is not shown in Figure 16. In comparison, the Kilo TM version is only slowed down by 6-9% with 30 cores.

Aside from HT-H, Kilo TM and WarpTM+TCD mostly follow the performance trends of FGLock with the scaled up GPU architecture. For HT-M, HT-L and ATM, the overhead of Kilo TM and WarpTM over FGLock remains approximately the same with more cores. WarpTM works more effectively for CL with 30 cores because the extra cores provide extra L1 cache capacity for transaction logs, allowing extra transaction concurrency without the L1 cache overflow penalty. WarpTM+TCD works less effectively for CLto and BH-L with 30 cores, whereas enabling each enhancement alone with 30 cores is just as effective as with 15 cores. A detailed investigation reveals that enabling both enhancements with Kilo TM boosts the transaction concurrency in these workloads, generating significantly more transaction conflicts. CC and AP are not affected by the increased concurrency limit, and hence, the extra cores do not impact the overhead of Kilo TM and WarpTM+TCD for these two workloads.

9. RELATED WORK

GPU Transactional Memory. Other than Kilo TM, there are other proposals to support transactional memory on GPU architectures. Cederman et al [6] proposed a GPU software TM system that uses per-object version locks to detect conflicts. As each transaction executes, it records the version of the object it access. These version numbers are later checked during commit to detect conflicts with committed transactions. Xu et al. [36] also proposed a GPU software TM system, but similar to Kilo TM, it uses value-based conflict detection. For each transaction, it has a hash table to store the set of memory locations that needs to be locked during commit. The memory locations stored in this table are sorted, so that every transaction acquires its lock in the same order to prevent deadlocks.

Energy Analysis for Transaction Memory. Ferri et al. [9] analyzed the energy and performance of SoC-TM, their TM system proposal for embedded multi-core SoC. Their analysis shows that for workloads that scale to multiple threads, SoC-TM performs better than locking while consuming less energy. We perform similar analysis for Kilo TM, a TM proposal for GPU architectures.

Intra-Warp Conflict Resolution. Qian et al. [29] described a method to detect and resolve conflicts among threads running on SMT CPU cores, which usually have far fewer threads per core in comparison to GPU cores. The relatively small number of threads allows their design to dedicate explicit storage to record the dependency between transactions and extend each cache line to record read-sharer information. Such storage is impractical for GPU cores which have more than 1000 threads on each core sharing the L1 cache. This work focuses on detecting and resolving conflicts among transactions within a warp on GPU.

Yang et al. [37] proposed multiplexing shared memory storage among multiple concurrently running thread blocks by dynamically allocating the storage to each thread block for temporary use and freeing it immediately after. Our proposed intra-warp conflict resolution employs the same strategy to allow each warp to use a ownership table larger than the capacity possible with static allocation.

Nasre et al. [24] proposed a probabilistic 3-phase conflict resolution that uses parallel passes to resolve conflicts among multiple threads. Similar to our proposed intra-warp 2-phase parallel conflict resolution, their approach uses thread ID to prioritize among different threads. However, their approach focuses on obtaining exclusive access to modify shared data and does not permit read-sharing, which is key to TM system performance.

Globally Synchronized Timers. Temporal Coherence [34] proposed by Singh et al., is a cache coherence framework for GPU architectures that uses a set of globally synchronized timers to eliminate invalidation messages. It uses timestamps to determine when cache blocks in local data cache will expire. Temporal conflict detection uses timestamps to detect if all the values read by a transaction can exist as a global memory snapshot. Ruan et al. [31] also proposed extending ORec-based STM systems with synchronous hardware timers found on existing CMP systems. Their approach embeds timestamps in the ownership record of each transaction variable in the main memory, whereas TCD uses a set of small on-chip buffers to conservatively record when each word in the global memory space was last written.

10. CONCLUSION

In this paper, we proposed two enhancements to Kilo TM, an existing hardware TM proposal for GPU architectures. *Warp-level transaction management* exploits the spatial locality among transactions within a warp to enable a set of optimizations. These optimizations allows Kilo TM to exploit the wide memory subsystem in GPU architectures. *Temporal conflict detection* complements WarpTM by allowing read-only transactions to commit silently in the absence of a conflict. Our evaluation shows that these two enhancements can improve Kilo TM performance by 65% while reducing its energy per operation by 34%. Kilo TM with the two enhancements can achieve 66% of the performance of fine-grained locking, while only requiring 34% more energy per operation. Moreover, software optimizations that reduce transaction footprints and contentions can further close this gap.

While this paper presents WarpTM and TCD as enhancements to Kilo TM, the insights behind these mechanisms extend well beyond GPU TM systems. WarpTM demonstrates the effectiveness of aggregating multiple transactions to amortize their management overheads in a TM system. This principle applies to other novel data synchronization/inter-thread communication mechanisms [38, 14]. The 2-phase parallel conflict resolution that enables WarpTM illustrates how transactions with predetermined order can resolves conflict in parallel with low overhead. This insight may be readily applied to thread-level speculation on multi-core systems. We believe that TCD's ability to cheaply verify that a thread has observed from an instantaneous global memory snapshot has wider uses beyond TM. For example, one may use TCD to accelerate runtime data-race detection on parallel computing systems without relying on any cache coherence protocol.

Finally, as newer commodity CMP systems start to add hardware support for TM, more software developers will start using transactions in their applications. GPU that supports TM will have higher interoperability with these future software applications. This will be an important design consideration for future heterogeneous processors with tightly integrated CMP and GPU.

11. ACKNOWLEDGEMENTS

We thank Henry Wong, Andrew Boktor and the anonymous reviewers for their valuable comments. This work was supported by an NVIDIA Graduate Fellowship and the Natural Sciences and Engineering Research Council of Canada.

12. REFERENCES

[1] R. Agrawal et al. Advances in Knowledge Discovery and Data Mining. chapter Fast Discovery of Association Rules. American Association for Artificial Intelligence, 1996.

[2] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[3] A. Brownsword. Cloth in OpenCL, 2009.

[4] I. A. Buck et al. United States Patent #7,627,723: Atomic Memory Operators in a Parallel Processor (Assignee NVIDIA Corp.), December 2009.

[5] 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.

[6] D. Cederman et al. Towards a Software Transactional Memory for Graphics Processors. In *EGPGV*, 2010.

[7] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, 2006.

[8] L. Dalessandro et al. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, 2010.

[9] C. Ferri et al. SoC-TM: Integrated HW/SW Support for Trasnactional Memory Programming on Embedded MPSoCs. In *CODES+ISSS*, 2011.

[10] W. W. L. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.

[11] W. W. L. Fung et al. Hardware Transactional Memory for GPU Architectures. In *MICRO*, 2011.

[12] W. W. L. Fung et al. `http://www.ece.ubc.ca/~wwlfung/code/kilotm-gpgpu_sim.tgz`, 2013.

[13] W. W. L. Fung et al. `http://www.ece.ubc.ca/~wwlfung/code/gpu-tm-tests.tgz`, 2013.

[14] A. Gharaibeh et al. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *PACT*, 2012.

[15] L. Hammond et al. Transactional Memory Coherence and Consistency. In *ISCA*, 2004.

[16] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool, second edition, 2010.

[17] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, 1993.

[18] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2012.

[19] G. Kestor et al. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *ICPE '11*, 2011.

[20] Khronos Group. OpenCL. `http://www.khronos.org/opencl/`.

[21] J. Leng et al. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.

[22] D. Merrill et al. Scalable GPU Graph Traversal. In *PPoPP*, 2012.

[23] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations with Wiring Alternatives for Large Caches with CACTI 6.5. In *MICRO*, 2007.

[24] R. Nasre et al. Morph Algorithms on GPUs. In *PPoPP*, 2013.

[25] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar.-Apr. 2008.

[26] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[27] NVIDIA Corp. *NVIDIA CUDA Programming Guide v3.1*, 2010.

[28] M. Olszewski et al. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT*, 2007.

[29] X. Qian et al. BulkSMT: Designing SMT Processors for Atomic-Block Execution. In *HPCA*, 2012.

[30] T. G. Rogers et al. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.

[31] W. Ruan et al. Boosting Timestamp-based Tranasctional Memory by Exploiting Hardware Cycle Counters. In *TRANSACT*, 2013.

[32] T. A. Shah. FabMem: A Multiported RAM and CAM Compiler for Superscalar Design Space Exploration. Master's thesis, North Carolina State University, 2010.

[33] A. Shriraman et al. Flexible Decoupled Transactional Memory Support. In *ISCA*, 2008.

[34] I. Singh et al. Cache Coherence for GPU Architectures. In *HPCA*, 2013.

[35] V. Vineet and P. Narayanan. CudaCuts: Fast Graph Cuts on the GPU. In *CVPRW '08*, 2008.

[36] Y. Xu et al. Software Transactional Memory for GPU Architectures. In *Computer Architecture Letters*, volume PP, 2013.

[37] Y. Yang et al. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput. In *PACT*, 2012.

[38] K. Yelick. Antisocial Parallelism: Avoiding, Hiding and Managing Communication. 2013. Keynote at HPCA-2013.

[39] L. Yen et al. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, 2007.